Mining and Learning With Graphs and Tensors

Namyong Park

CMU-CS-22-103
May 2022

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Christos Faloutsos (Chair)
Tom Mitchell
Leman Akoglu
Xin Luna Dong (Meta)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was sponsored by the Bloomberg Data Science Ph.D. Fellowship, the ILJU Foundation Ph.D. Fellowship, and by Carnegie Mellon University CyLab, with generous support from Microsoft. This work was also supported by the National Science Foundation under Grants No. IIS-1247489 and IIS-1408924; by the Army Research Laboratory under Cooperative Agreement Number W911NF-09-2-00S3; by the Pennsylvania Infrastructure Technology Alliance; and by gifts from Innovu and the PNC Center. Additional support was provided by the project AIDA - Adaptive, Intelligent and Distributed Assurance Platform (reference POCI-01-0247-FEDER-045907). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation, or other funding parties.

Keywords: graphs, tensors, dynamic networks, multi-aspect networks, unsupervised learning, semi-supervised learning, knowledge graphs, graph neural networks, graph representation learning, node importance estimation, explainable recommendation, recommendation justification, tensor factorization, distributed algorithms, model selection, algorithm selection, meta-learning, dynamic system modeling, model revision, prior knowledge incorporation, evolutionary algorithms, temporal knowledge graphs, reasoning over temporal knowledge graphs, temporal point processes, deep graph clustering, temporal graph clustering, contrastive learning, community detection and tracking

To my family.

Abstract

Data generated in diverse contexts can be modeled as graphs. Examples are numerous, from citation and social networks to the World Wide Web. Many real-world networks are multi-aspect, where multiple types of entities interact with each other via various relations. Also, many of them are dynamic, modeling relationships among entities and their features that evolve over time. These real-world networks with rich side information (e.g., node and edge types, and edge timestamps) are naturally modeled as tensors (i.e., multi-dimensional arrays).

Given graphs and tensors, how can we understand them, and utilize them for downstream tasks? Specifically, how can we analyze and model large real-world networks, and gain a better understanding of how they form and evolve? Also, how can we design algorithms that leverage graphs and tensors for important applications such as recommendation and ranking? This thesis focuses on these fundamental problems by developing effective and efficient methods for mining and learning with graphs and tensors.

In the first part of the thesis, we focus on addressing important mining and learning tasks for static graphs and tensors. We first propose novel graph-regularized semi-supervised algorithms for estimating node importance in a knowledge graph, which achieve up to *25% higher accuracy* than the best baseline. Then we develop distributed frameworks for large-scale tensor factorization, which decompose and summarize large tensors up to *180× faster* than existing methods, with near-linear scalability. We also design a meta-learning based approach for automatic graph learning model selection, which is up to *15× more accurate* than using popular methods consistently. In addition, we develop a method that explains product recommendations, up to *21% more accurately* than the best baseline, by performing personalized inference over a product graph.

In the second part of the thesis, we focus on modeling and reasoning with dynamic graphs and tensors, which represent various types of time-evolving networks and dynamic real-world phenomena. We propose a framework to learn differential equations (DEs) that model the observed phenomena (such as weather and water quality), which produces interpretable and physically plausible DEs that achieve up to *34% higher forecasting accuracy* than related baselines. We then tackle the task of finding communities in networks and tracking their evolution by designing a contrastive graph clustering framework, which shows up to *27% higher clustering accuracy* than existing methods. Further, we develop a method for reasoning over temporal knowledge graphs (TKGs), which infers new knowledge from the given TKG, up to *116% more accurately* than the best baseline, while being *30× faster* in model training.

Throughout the thesis, a strong emphasis is placed on developing *effective*, *accurate*, and *scalable* tools. To this end, we use mathematical techniques (e.g., approximation), exploit the characteristics of real-world networks, incorporate prior knowledge and experience, and employ powerful theoretical and practical frameworks, including graph neural networks, latent variable modeling, temporal point processes, and distributed computing. We successfully apply these tools to a large array of real-world datasets and applications, establishing new state-of-the-art results.

Acknowledgments

First of all, I would like to sincerely thank my advisor, Christos Faloutsos, for his support and guidance. Throughout my time in graduate school, I really appreciated his encouragement, insightful comments, patience, and enthusiasm in research, which made my graduate study truly enjoyable and fruitful.

I would also like to thank my other thesis committee members, Tom Mitchell, Leman Akoglu, and Xin Luna Dong, for their constructive feedback and invaluable advice in putting together and improving my thesis.

One of the most exciting parts of my time in graduate school has been my summer internships at Amazon, Microsoft Research, and Adobe Research. I am grateful to my mentors for providing wonderful environments and opportunities to work on exciting problems, and also for their advice and stimulating discussions: Xin Luna Dong, Yuxiao Dong, Andrey Kan, Eunyee Koh, and Ryan Rossi.

I was fortunate to have worked with amazing collaborators and coauthors, who deserve my gratitude: Nesreen Ahmed, Seojin Bang, Iftikhar Ahamath Burhanuddin, Duen Horng Chau, Dana Cristofor, Jun-Gi Jang, Byungsoo Jeon, Cara Jones, Jinhong Jung, U Kang, MinHyeok Kim, Dong-Kyun Kim, Sungchul Kim, Aayushi Kulshrestha, Sael Lee, Meng-Chieh Lee, Sacha Levy, Yifei Li, Fuchen Liu, Bob McKay, Purvanshi Mehta, Pratheeksha Nair, Xuan Hoai Nguyen, Sejoon Oh, Andreas Olligschlaeger, Ha-Myung Park, Reihaneh Rabbany, Catalina Vajiac, and Tong Zhao. I would especially like to thank U Kang for introducing me to data mining research, and for his mentorship, and Bob McKay for guiding me in my early pursuits as a researcher, and for his thoughtfulness.

I am thankful to the CMU Data Mining group members and visitors for the interesting discussions and conversations, including Dhivya Eswaran, Brian Hooi, Meng-Chieh Lee, Reihaneh Rabbany, Shubhranshu Shekhar, Kijung Shin, Hyun Ah Song, Catalina Vajiac, and Saranya Vijayakumar. I would also like to thank Deborah Cavlovich, Tony Mareino, and Ann Stetser for their administrative support and helping my studies at CMU run smoothly.

Finally, and most of all, I would like to thank my family, especially my parents, for their endless love, support, and belief in me throughout all stages of my life and career.

Contents

1 Introduction **1**
- 1.1 Overview and Contributions . . . 2
  - 1.1.1 Part I: Static Graphs and Tensors . . . 3
  - 1.1.2 Part II: Dynamic Graphs and Tensors . . . 9

I Static Graphs and Tensors **15**

2 Estimating Node Importance in Knowledge Graphs Using Graph Neural Networks **17**
- 2.1 Introduction . . . 17
- 2.2 Preliminaries . . . 20
  - 2.2.1 Problem Definition . . . 20
  - 2.2.2 Desiderata for Modeling Node Importance in KGs . . . 21
  - 2.2.3 Graph Neural Networks . . . 22
- 2.3 Method . . . 22
  - 2.3.1 Score Aggregation . . . 24
  - 2.3.2 Predicate-Aware Attention Mechanism . . . 25
  - 2.3.3 Centrality Adjustment . . . 26
  - 2.3.4 Model Architecture . . . 26
  - 2.3.5 Model Training . . . 28
- 2.4 Experiments . . . 28
  - 2.4.1 Datasets . . . 28
  - 2.4.2 Baselines . . . 29
  - 2.4.3 Performance Evaluation . . . 30
  - 2.4.4 Importance Estimation on Real-World Data . . . 31
  - 2.4.5 Analysis of GENI . . . 32
- 2.5 Related Work . . . 34
- 2.6 Conclusion . . . 35
- 2.7 Appendix . . . 36
  - 2.7.1 Datasets . . . 37
  - 2.7.2 Experimental Settings . . . 38
  - 2.7.3 Additional Evaluation . . . 39

3 Inferring Node Importance in a Knowledge Graph from Multiple Input Signals **41**
3.1 Introduction . . . 42
3.2 Background . . . 44
3.3 Task Description . . . 44
3.4 Methods . . . 45
3.4.1 Learning Objective . . . 46
3.4.2 Handling Rebel Input Signals . . . 49
3.4.3 Graph Neural Networks for Node Importance Estimation . . . 50
3.5 Experiments . . . 52
3.5.1 Dataset Description . . . 52
3.5.2 Performance Evaluation . . . 53
3.5.3 Baselines . . . 56
3.5.4 Q1. Accuracy . . . 56
3.5.5 Q2. Use in Downstream Tasks . . . 56
3.5.6 Q3. Handling Rebel Signals . . . 58
3.6 Related Work . . . 59
3.7 Conclusion . . . 60
3.8 Appendix . . . 60
3.8.1 Experimental Settings . . . 60

4 Principled and Scalable Recommendation Justification **63**
4.1 Introduction . . . 63
4.2 Justifying Recommendations . . . 65
4.2.1 Problem Statement . . . 65
4.2.2 Product Graph and Justifications . . . 66
4.2.3 Quantifying the Quality of Justifications . . . 68
4.2.4 Justification Discovery . . . 71
4.3 Evaluation Using Axioms . . . 74
4.3.1 Axioms . . . 74
4.3.2 Baselines . . . 76
4.3.3 Results . . . 77
4.4 Evaluation Using Real-World Data . . . 78
4.4.1 Datasets . . . 78
4.4.2 Baselines . . . 79
4.4.3 Q1. Justification Quality . . . 79
4.4.4 Q2. Scalability . . . 81
4.4.5 Q3. Relevance-Diversity Trade-Off . . . 81
4.5 Related Work . . . 82
4.6 Conclusion . . . 83
4.7 Appendix . . . 83
4.7.1 Experimental Settings . . . 83

5 Fast and Scalable Distributed Boolean Tensor Factorization **87**

5.1 Introduction . . . 87
5.2 Preliminaries . . . 90
5.2.1 Boolean Arithmetic . . . 90
5.2.2 Notation . . . 90
5.2.3 Tensor Rank and Tensor Decompositions . . . 92
5.3 Related Works . . . 97
5.3.1 Boolean Tensor Decomposition . . . 97
5.3.2 Normal Tensor Decomposition . . . 97
5.3.3 Partitioning of Sparse Tensors . . . 98
5.3.4 Distributed Computing Frameworks . . . 98
5.4 Proposed Method . . . 99
5.4.1 Updating a Factor Matrix . . . 101
5.4.2 Updating a Core Tensor . . . 102
5.4.3 Distributed Generation and Minimal Transfer of Intermediate Data 103
5.4.4 Exploiting the Characteristics of Boolean Operation and Boolean Tensor Factorization . . . 104
5.4.5 Careful Partitioning of the Workload . . . 108
5.4.6 Putting Things Together . . . 111
5.4.7 Implementation . . . 113
5.4.8 Analysis . . . 115
5.5 Experiments . . . 117
5.5.1 Experimental Settings . . . 118
5.5.2 Data Scalability . . . 122
5.5.3 Machine Scalability . . . 124
5.5.4 Reconstruction Error . . . 125
5.6 Conclusion . . . 128
5.7 Appendix . . . 129
5.7.1 Proof of Lemma 5.4 . . . 129
5.7.2 Proof of Lemma 5.5 . . . 130
5.7.3 Proof of Lemma 5.6 . . . 130
5.7.4 Proof of Lemma 5.7 . . . 131
5.7.5 Proof of Lemma 5.8 . . . 131
5.7.6 Proof of Lemma 5.9 . . . 132
5.7.7 Proof of Lemma 5.10 . . . 132
5.7.8 Proof of Lemma 5.11 . . . 132

6 Fast Automatic Model Selection for Graph Representation Learning 135
6.1 Introduction . . . 136
6.2 Problem Formulation . . . 138
6.3 Framework . . . 139
6.3.1 Offline Meta-Training . . . 139
6.3.2 Online Model Prediction . . . 142
6.3.3 Structural Meta-Graph Features . . . 142
6.3.4 Embedding Models and Graphs . . . 144

6.4 Experiments . . . 145
6.4.1 Experimental Settings . . . 145
6.4.2 Model Selection Accuracy (RQ1) . . . 147
6.4.3 Model Selection Efficiency (RQ2) . . . 149
6.4.4 Effects of Meta-Graph Features (RQ3) . . . 150
6.5 Related Work . . . 151
6.5.1 Model Selection in Machine Learning . . . 151
6.5.2 Model Selection in Graph Learning . . . 152
6.6 Conclusion . . . 153
6.7 Appendix . . . 153
6.7.1 Model Set . . . 153
6.7.2 Graph Domains . . . 153
6.7.3 Runtime . . . 153
6.7.4 AUTOGRL Algorithm . . . 155
6.7.5 Experimental settings . . . 155
6.7.6 Meta-Graph Features . . . 155

II Dynamic Graphs and Tensors 159

7 Knowledge-Guided Dynamic Systems Modeling 161
7.1 Introduction . . . 161
7.2 River Water Quality Modeling . . . 164
7.3 Methods . . . 166
7.3.1 Representing Dynamic Processes Using TAG . . . 167
7.3.2 Knowledge-Guided Genetic Model Revision . . . 169
7.3.3 Applying GMR to Real-World Problems . . . 174
7.3.4 Improving the Efficiency and Effectiveness . . . 175
7.4 Experiments . . . 176
7.4.1 Dataset and Modeling Task Description . . . 177
7.4.2 Comparators . . . 178
7.4.3 Performance Evaluation . . . 179
7.4.4 Q1. Prediction Accuracy . . . 179
7.4.5 Q2. Ecological Analysis . . . 180
7.4.6 Q3. Analysis of Speedup Techniques . . . 182
7.5 Related Work . . . 183
7.6 Conclusion . . . 184
7.7 Appendix . . . 185
7.7.1 Further Details of River Modeling . . . 185
7.7.2 Experimental Settings . . . 186

8 Jointly Modeling Event Time and Network Structure for Reasoning over Temporal Knowledge Graphs 187
8.1 Introduction . . . 187

8.2 Problem Formulation . . . 189
8.3 Modeling a Temporal Knowledge Graph . . . 191
8.3.1 Modeling Event Time . . . 191
8.3.2 Modeling Evolving Network Structure . . . 194
8.3.3 Parameter Learning . . . 196
8.4 Experiments . . . 196
8.4.1 Temporal Knowledge Graph Data . . . 196
8.4.2 Event Time Prediction (RQ1) . . . 198
8.4.3 Temporal Link Prediction (RQ2) . . . 199
8.4.4 Efficiency (RQ3) . . . 201
8.4.5 Ablation Study (RQ4) . . . 201
8.5 Related Work . . . 203
8.6 Conclusion . . . 205
8.7 Appendix . . . 205
8.7.1 Experimental Settings . . . 205

9 Contrastive Graph Clustering for Community Detection and Tracking 207
9.1 Introduction . . . 208
9.2 Problem Formulation . . . 210
9.2.1 Graph Clustering . . . 210
9.2.2 Temporal Graph Clustering . . . 211
9.3 Preliminaries . . . 211
9.4 Proposed Framework . . . 212
9.4.1 CGC: Contrastive Graph Clustering . . . 212
9.4.2 CGC for Temporal Graph Clustering . . . 216
9.5 Experiments . . . 218
9.5.1 Datasets . . . 218
9.5.2 Baselines . . . 219
9.5.3 Node Clustering Quality (RQ1) . . . 222
9.5.4 Temporal Link Prediction Accuracy (RQ2) . . . 223
9.5.5 Ablation Study (RQ3) . . . 223
9.6 Related Work . . . 225
9.7 Conclusion . . . 227
9.8 Appendix . . . 227
9.8.1 Mining Case Studies . . . 227
9.8.2 Clustering Performance over Time . . . 229
9.8.3 Experimental Settings . . . 231
9.8.4 Graph Stream Segmentation . . . 233

III Conclusions and Future Directions 235

10 Conclusions 237
10.1 Summary of Contributions . . . 237

10.1.1 Part I: Static Graphs and Tensors 237
10.1.2 Part II: Dynamic Graphs and Tensors 238

11 Future Directions **241**
11.1 Complex Anomaly Detection 241
11.2 Modeling Dynamic Networks 242
11.3 Knowledge Reasoning 243

Bibliography **245**

List of Figures

1.1 Examples of static and dynamic multi-aspect networks. 1
1.2 MULTIIMPORT and GENI accurately estimate node importance 4
1.3 J-RECS is effective and scales up near-linearly 5
1.4 DBTF-CP is fast and scalable 6
1.5 DBTF-TK is fast and scalable 7
1.6 AUTOGRL efficiently infers the best graph learning model 8
1.7 GMR is accurate and fast 11
1.8 EVOKG accurately predicts temporal links and event time 12
1.9 CGC achieves high node clustering accuracy 14

2.1 An example knowledge graph on movies and related entities 18
2.2 GENI estimates node importance accurately 19
2.3 Description of node importance estimation by GENI 24
2.4 Parameter sensitivity of GENI 34

3.1 An example knowledge graph, problem setup, and the superior performance of MULTIIMPORT in estimating node importance 43
3.2 MULTIIMPORT estimates the latent node importance 46
3.3 MULTIIMPORT infers node importance by identifying similar signals . . . 51
3.4 In- and out-of-domain evaluation 55
3.5 MULTIIMPORT effectively handles rebel signals 59

4.1 J-RECS generates effective justifications 65
4.2 A movie product graph 67
4.3 Recommendations are enriched by justifications 67
4.4 Axioms and expected relevance scores of product attributes 74
4.5 J-RECS exhibits near-linear scalability 81
4.6 Relevance-diversity trade-off 81

5.1 Rank-R Boolean CP decomposition 94
5.2 Rank-R Boolean Tucker decomposition 95
5.3 Updating a factor matrix for Boolean CP and Tucker factorization 100
5.4 DBTF-CP reduces intermediate operations by exploiting the characteristics of Boolean CP factorization 105

5.5 DBTF-TK reduces intermediate operations by exploiting the characteristics of Boolean CP factorization . . . 106
5.6 An overview of partitioning in DBTF . . . 109
5.7 Scalability of DBTF-CP and other methods on synthetic datasets . . . 121
5.8 Scalability of DBTF-TK and other methods on synthetic datasets . . . 122
5.9 Scalability of DBTF-CP and other methods on real-world datasets . . . 124
5.10 Scalability of DBTF-TK and other methods on real-world datasets . . . 125
5.11 Scalability of DBTF-CP and DBTF-TK with respect to the number of machines 125
5.12 Reconstruction error of DBTF-CP and other methods . . . 126
5.13 Reconstruction error of DBTF-TK and other methods . . . 127

6.1 Overview of AUTOGRL compared to existing naive approach . . . 136
6.2 AUTOGRL infers the best model by applying a meta-learned model to meta-graph features . . . 140
6.3 An overview of meta-graph feature construction in AUTOGRL . . . 143
6.4 AUTOGRL is fast, and incurs negligible overhead . . . 150

7.1 GMR produces an accurate forecasting model, guided by domain knowledge 164
7.2 Tree composition operations used by tree-adjoining grammar . . . 167
7.3 Representation and revision of dynamic processes in GMR . . . 168
7.4 TAG derivation tree encoding a revised differential equation . . . 168
7.5 An overview of the model revision framework . . . 170
7.6 Genetic operators in TAG3P . . . 170
7.7 Incorporating knowledge on plausible revisions in GMR . . . 173
7.8 Nakdong River basin in South Korea . . . 178
7.9 Selectivity of variables . . . 181
7.10 Mean runtime by speedup techniques . . . 182
7.11 Effect of evaluation short-circuiting . . . 183
7.12 An example river system . . . 185

8.1 An example temporal knowledge graph . . . 188
8.2 EvoKG achieves the best link and time prediction results . . . 191
8.3 EvoKG achieves the best event time prediction results . . . 198
8.4 EvoKG efficiently performs model training and inference . . . 201
8.5 Parameter sensitivity analysis of the prediction accuracy . . . 202
8.6 Modeling event time improves temporal link prediction accuracy . . . 203

9.1 CGC achieves the best node clustering performance . . . 209
9.2 Node clustering results obtained with different contrastive objectives . . . 225
9.3 Case study 1: two groups with traveling members . . . 228
9.4 Case study 2: two groups reorganizing into three . . . 228
9.5 Node clusters based on their transition patterns . . . 229
9.6 Performance of CGC over time . . . 230

List of Tables

1.1 Organization of the thesis . . . 2
1.2 MULTIIMPORT and GENI satisfy the desiderata for estimating node importance . . . 3
1.3 Knowledge-guided model revision satisfies the desiderata for modeling dynamic systems . . . 10
1.4 CGC wins on features . . . 13

2.1 Comparison of methods for estimating node importance . . . 21
2.2 Table of symbols . . . 23
2.3 Real-world knowledge graphs used in experiments . . . 28
2.4 GENI performs in-domain prediction most accurately . . . 31
2.5 GENI performs out-of-domain prediction most accurately . . . 32
2.6 Performance of GENI obtained with shared and distinct embeddings . . . 33
2.7 Performance with fixed and flexible centrality adjustment . . . 33
2.8 Top-10 movies and directors predicted by different methods . . . 36
2.9 RMSE of in-domain prediction obtained by supervised methods . . . 40

3.1 MULTIIMPORT considers all available signals to measure node importance in a knowledge graph . . . 42
3.2 Table of symbols . . . 48
3.3 Real-world knowledge graphs used in our evaluation . . . 53
3.4 Input signals in real-world knowledge graphs . . . 54
3.5 MULTIIMPORT estimates node importance more accurately than all baselines 54
3.6 MULTIIMPORT achieves the best signal prediction results . . . 57
3.7 MULTIIMPORT outperforms all baselines in forecasting signals . . . 58

4.1 Table of symbols . . . 69
4.2 J-RECS satisfies all axioms . . . 75
4.3 Statistics of real-world product graphs . . . 78
4.4 First qualitative analysis of the justifications learned by J-RECS and MP-AND 84
4.5 Second qualitative analysis of the justifications learned by J-RECS and MP-AND . . . 85

5.1 Comparison of DBTF and existing Boolean tensor factorization methods . 88
5.2 Table of symbols . . . 91

5.3 Real-world and synthetic tensors used for experiments 120

6.1 Summary of notations 141
6.2 AUTOGRL is the most accurate in the *search-within-a-model* testbed 148
6.3 AUTOGRL is the most accurate in the *search-across-all-models* testbed . . . 148
6.4 Our proposed meta-graph features enable effective automatic model selection 151
6.5 Graph representation learning models and their hyperparameter settings 154
6.6 Distribution of the graphs in the testbed 154
6.7 Runtime comparison between AUTOGRL and naive model selection . . . 155
6.8 Global statistical functions for deriving meta-graph features 156

7.1 Model revision satisfies all properties for interpretable knowledge-guided modeling of complex dynamic systems 163
7.2 Variables, connectors, and extenders used by extensions 174
7.3 Constant parameters that capture the knowledge on model parameters . . 175
7.4 Temporal variable parameters in the river process 176
7.5 GMR achieves the best forecasting accuracy 180

8.1 EvoKG deals with two major tasks for reasoning over temporal knowledge graphs (TKGs), while possessing desirable features for modeling TKGs . . 190
8.2 Table of symbols 193
8.3 Statistics of real-world temporal knowledge graphs 197
8.4 EvoKG achieves the best temporal link prediction results 200

9.1 CGC wins on features 208
9.2 Table of symbols 213
9.3 CGC achieves the best node clustering results on static graphs 220
9.4 CGC achieves the highest node clustering accuracy on the temporal graph 221
9.5 CGC achieves the best temporal link prediction performance 224
9.6 Summary of temporal real-world datasets 231
9.7 Summary of static real-world datasets 231

Chapter 1

Introduction

Graphs provide a powerful framework to model real-world entities and their relationships. Thus, data generated in diverse contexts has been modeled as graphs. Many real-world graphs are multi-aspect, where multiple types of entities interact with each other via various relations. Also, many of them are dynamic, modeling how the relations among entities and their features evolve over time. Thus, graphs with such side information (e.g., node and edge types, and edge timestamps) are naturally modeled as tensors (i.e., multi-dimensional arrays). Using graphs and tensors, we can model a large array of real-world networks, including product graphs (Figure 1.1a), knowledge graphs (Figure 1.1b), sensor networks (Figure 1.1c), social networks, and citation graphs, to name a few.

Given graphs and tensors, how can we understand them, and effectively use them for downstream tasks? Specifically, how can we analyze and model large-scale real-world networks, and obtain a better understanding of the formation and dynamics of real-world networks? Also, how can we design algorithms that effectively leverage graphs and

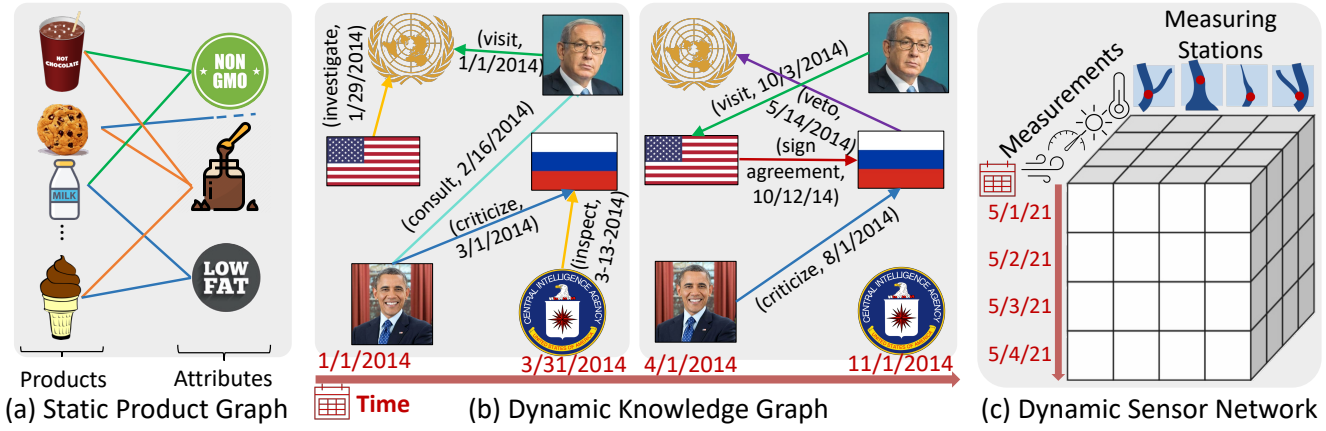


Figure 1.1: Examples of static and dynamic multi-aspect networks.

tensors for important applications such as explainable recommendation, ranking, and knowledge reasoning? This thesis focuses on these fundamental problems by *developing effective and scalable methods for mining and learning with graphs and tensors*.

Since real-world graphs and tensors are *complex* (complex connectivity patterns with multiple types of nodes and edges), *incomplete* (with many missing edges), *huge* (often containing billions of edges or more), and *continuously changing*, it is challenging or often impossible to analyze and utilize such real-world networks using conventional tools designed for simple and small networks. To develop effective, accurate, and scalable tools, we employ mathematical techniques (e.g., approximation), exploit the characteristics of real-world networks, incorporate prior knowledge, and utilize powerful theoretical and practical frameworks, including graph neural networks, latent variable modeling, temporal point processes, evolutionary computation, and distributed computing. With these advancements, we apply our methods to a wide variety of large real-world datasets and applications, and establish new state-of-the-art results.

1.1 Overview and Contributions

This thesis is organized in two parts. In the first part, we focus on addressing important mining and learning tasks for static graphs and tensors, such as estimating node importance, justifying recommendations using a product graph, and selecting graph learning models automatically. In the second part, we focus on modeling and reasoning with dynamic graphs and tensors, such as reasoning over temporal knowledge graphs, and community detection and tracking. Table 1.1 shows an organization of this thesis, along with research problems that each chapter tackles. Below, we provide an overview of the

Table 1.1: Organization of the thesis.

	Research Problem
Part I: Static Graphs and Tensors	• **Node Importance Estimation** (Chapters 2 and 3): How can we accurately estimate the importance of nodes in a multi-aspect graph? • **Recommendation Justification** (Chapter 4): How can we provide personalized explanations of recommendations using a product graph? • **Distributed Tensor Factorization** (Chapter 5): How can we factorize large-scale tensors and summarize them using a cluster of machines? • **Automatic Graph Learning Model Selection**: (Chapter 6): How can we automatically select the best graph learning algorithm for an input graph?
Part II: Dynamic Graphs and Tensors	• **Dynamic Systems Modeling** (Chapter 7): How can we find mathematical expressions that model the dynamics of real-world phenomena? • **Reasoning Over Temporal Knowledge Graphs** (Chapter 8): How to infer new knowledge by reasoning over knowledge graphs that evolve over time? • **Community Detection and Tracking** (Chapter 9): How to find communities in networks and track their evolution in an unsupervised manner?

goals and contributions of each of our proposed methods.

1.1.1 Part I: Static Graphs and Tensors

Part I addresses important mining and learning tasks for static graphs and tensors, namely, node importance estimation (Section 1.1.1.1), reasoning over graph-structured data for explainable recommendation (Section 1.1.1.2), large-scale tensor factorization (Section 1.1.1.3), and automatic graph learning model selection (Section 1.1.1.4).

1.1.1.1 Estimating Node Importance in Knowledge Graphs (Chapters 2 and 3)

"How can we estimate node importance in a knowledge graph (KG), given signals on node importance for some nodes in the KG?"

A knowledge graph (KG) is a multi-relational graph where nodes are connected via multiple types of relations. KGs such as Freebase [BEP+08], YAGO [SKW07], and DBpedia [LIJ+15] have proven highly valuable for many applications including question answering [DWZX15], recommendation [ZYL+16], and semantic search [BWY13]. Given input signals on node importance, i.e., values representing the significance or popularity of a node, such as the number of pageviews, how can we estimate node importance by making use of input signals known for some nodes along with auxiliary information in KGs such as connectivity patterns and edge types (predicates)? So far, existing techniques have approached this problem in a non-trainable framework, which is based on a fixed model structure determined by their prior assumptions on node importance and its relation to the graph structure. Also, a majority of methods are not designed for KGs, failing to utilize multi-relational information in capturing node importance.

In Chapter 2, we explore a new family of solutions for estimating node importance in KGs, namely, graph-regularized semi-supervised machine learning algorithms, and develop an effective method called GENI, based on the graph neural networks (GNNs), which

Table 1.2: [Chapters 2 and 3] **Proposed MULTIIMPORT and GENI satisfy more desiderata** for estimating node importance than existing methods. *Neighborhood*: Neighborhood awareness. *Predicate*: Making use of predicates. *Centrality*: Centrality awareness. *Input Signal*: Utilizing input signal on node importance. *Flexibility*: Flexible adaptation.

	MULTIIMPORT (Chapter 3)	**GENI (Chapter 2)**	**HAR [LNY12]**	**PPR [Hav02]**	**PR [PBMW99]**
Neighborhood	✓	✓	✓	✓	✓
Predicate	✓	✓	✓		
Centrality	✓	✓	✓	✓	✓
Flexibility	✓	✓			
Single Input Signal	✓	✓	✓	✓	
Multiple Input Signals	✓				

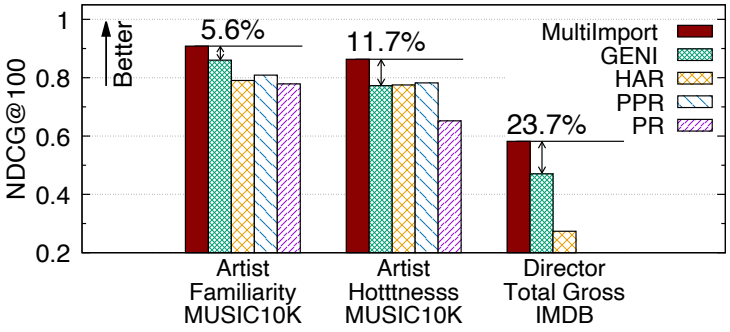


Figure 1.2: [Chapters 2 and 3] **Our proposed MULTIIMPORT and GENI win.** The two proposed semi-supervised methods outperform previous non-trainable approaches. MULTIIMPORT further improves upon GENI by learning from multiple input signals, estimating node importance even up to 23.7% more accurately.

is designed to satisfy the desiderata for an effective semi-supervised node importance estimation, e.g., centrality and predicate awareness, as summarized in Table 1.2.

In Chapter 3, we extend the problem of node importance estimation to an even more general setup (Table 1.2), where we are given multiple types of input signals, which may have been partially observed and partially overlapping with each other. We tackle this problem by designing a latent variable model and deriving an effective learning objective to infer node importance by learning to represent multiple signals in an effective graph-based estimator.

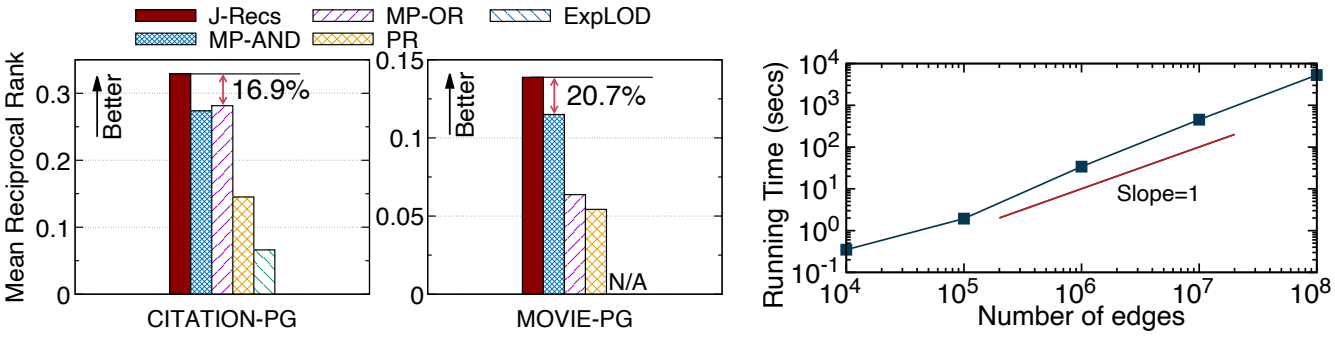
Contributions:

- **Novel Problem Formulation.** We explore graph-regularized semi-supervised machine learning algorithms for estimating node importance in KGs from heterogeneous information reflecting node importance, as opposed to previous non-trainable approaches.
- **Algorithms.** We present two algorithms, GENI and MULTIIMPORT, novel semi-supervised GNN-based methods that infer node importance by fusing heterogeneous information from multiple sources (e.g., input signals, graph structure, edge types).
- **Effectiveness.** The proposed algorithms estimate node importance up to *24% more accurately* than the best existing method.

1.1.1.2 Principled and Scalable Recommendation Justification (Chapter 4)

"How can we explain product recommendations to a user?"

Online recommendation is an essential functionality across a variety of services, including e-commerce and video streaming, where items to buy, watch, or read are suggested to users. Justifying recommendations, i.e., explaining why a user might like the recommended item [BC17], has been shown to improve user satisfaction [HKR00] and



(a) Preference Retrieval.

(b) Scalability.

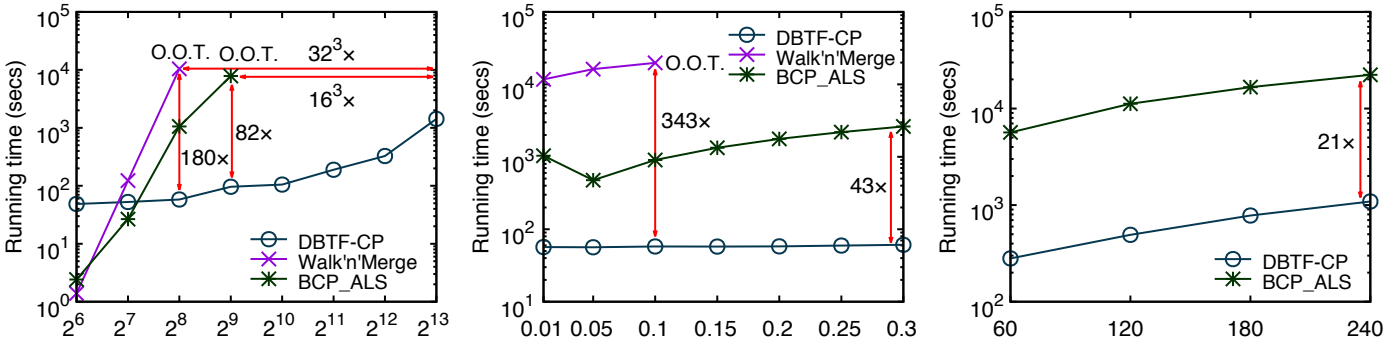
Figure 1.3: [Chapter 4] (a) **Our proposed J-RECS wins.** J-RECS generates justifications that match user preferences up to 20.7% better than the best baseline. (b) J-RECS exhibits near-linear scalability.

persuasiveness of the recommendation [TM07, TM15]. In this work, our goal is to develop a method for generating post-hoc justifications, in which recommendations and justifications are decoupled from each other. Since justifications are generated after the recommendation has been given, post-hoc justification methods can flexibly be applied to different types of recommendation algorithms (thus *model-agnostic*) [VSR09]. While several post-hoc methods have been developed, they are limited in providing diverse justifications, as they either use only one of many available types of input data, or rely on the predefined templates.

In Chapter 4, we develop a method called J-RECS, which produces concise yet diverse justifications. Given heterogeneous data on the user and products (e.g., user purchase history and product attributes), J-RECS uses a graph-based representation to leverage heterogeneous data and their relations for justifications. The graph-based representation allows J-RECS to generate justifications personalized with respect to both the user and the recommended item. In this product graph, J-RECS finds a set of relevant product attributes and use them to produce justifications, where we model the relevance score of product attributes based on the entity proximity in a graph. Also, to provide informative and engaging justifications to the user, J-RECS takes the diversity of chosen product attributes into account. Importantly, by using the submodularity of the objective function, J-RECS can efficiently infer justifications from a large product graph.

Contributions:

- **Problem Formulation.** We present a graph-based formulation of the problem of generating concise and diverse justifications given various types of user and product data.
- **Principled Approach.** We develop J-RECS, a principled post-hoc framework to infer justifications. J-RECS is guided by a set of principles characterizing desirable justifications, and does not require manually labeled data.
- **Effectiveness and Scalability.** We demonstrate that J-RECS satisfies desirable proper-



(a) Dimensionality. (b) Density. (c) Rank.

Figure 1.4: [Chapter 5] **Our proposed DBTF-CP wins.** Scalability of DBTF-CP and baselines with respect to the dimensionality and density of a tensor, and the rank of CP decomposition. Baselines often run out of time (O.O.T.) as the dimensionality and density of a tensor increase.

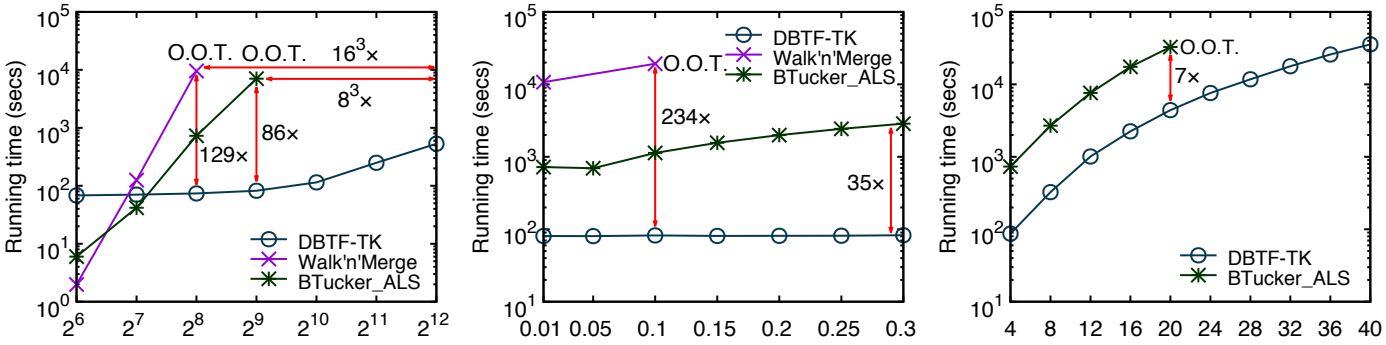
ties of justifications. Justifications generated by J-RECS match user preference up to *21% more accurately* than the best baseline (Figure 1.3a). We also show that J-RECS is scalable, *running in time linear* in the size of input data (Figure 1.3b).

1.1.1.3 Fast and Scalable Distributed Boolean Tensor Factorization (Chapter 5)

"How can we analyze tensors that are composed of 0's and 1's? How can we efficiently analyze such Boolean tensors that have millions or even billions of entries?"

Boolean tensors often represent relationship, membership, or occurrences of events such as subject-relation-object tuples in knowledge base data (e.g., 'Seoul'-'is the capital of'-'South Korea'). Boolean tensor factorization (BTF) is a useful tool for analyzing binary tensors, which can be used for applications such as latent concept discovery, clustering, recommendation, link prediction, and synonym finding. BTF requires that the input tensor, all factor matrices, and a core tensor are binary. Furthermore, BTF uses Boolean sum instead of normal addition. When the data is inherently binary, BTF is an appealing choice as it can reveal the structures and relationships underlying the binary tensor that are hard to be found by other factorizations. However, while several BTF algorithms exist, they fail to efficiently process, and scale up for large-scale Boolean tensors.

In Chapter 5, we develop a fast and scalable algorithm for large-scale BTF. Specifically, we develop a distributed method called DBTF for Boolean CP (DBTF-CP) and Tucker (DBTF-TK) factorizations running on the Apache Spark framework. First, DBTF performs distributed data generation and minimizes the transfer of intermediate data, Second, DBTF minimizes the number of operations for factorizing Boolean tensors by exploiting the characteristics of Boolean operation and Boolean tensor factorization, and caching intermediate results; this significantly decreases the number of operations required to update factor matrices. Third, DBTF carefully partitions the workload, which facilitates



(a) Dimensionality. (b) Density. (c) Core Size $R_1 = R_2 = R_3$.

Figure 1.5: [Chapter 5] **Our proposed DBTF-TK wins.** Scalability of DBTF-TK and baselines with respect to the dimensionality and density of a tensor, and the core size of Tucker decomposition. Baselines often run out of time (O.O.T.) as the dimensionality and density of a tensor, and core size increase.

the reuse of intermediate results and minimizes data shuffling.

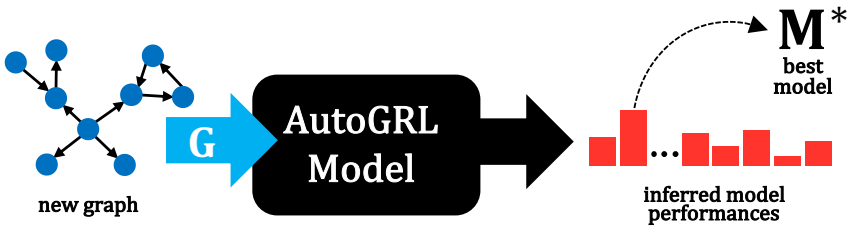
Contributions:

- **Fast and Scalable Algorithms.** We develop two distributed algorithms, DBTF-CP and DBTF-TK, for distributed Boolean CP and Tucker factorizations, which are carefully designed to achieve high speed and scalability. These algorithms are the first distributed algorithms for large-scale BTF.
- **Theoretical Analysis.** We provide an analysis of the proposed DBTF-CP and DBTF-TK in terms of time complexity, memory requirement, and the amount of shuffled data.
- **Effectiveness.** DBTF-CP decomposes up to *16^3–$32^3\times$ larger tensors* than existing methods in *82–180× less time* (Figure 1.4a). Overall, DBTF-CP achieves *21–343× speedup* and exhibits *near-linear scalability* with regard to all data aspects (Figure 1.4). DBTF-TK decomposes up to *8^3–$16^3\times$ larger tensors* than existing methods in *86–129× less time* (Figure 1.5a). Overall, DBTF-TK achieves *7–234× speedup* and exhibits *near-linear scalability* with regard to all data aspects (Figure 1.5).

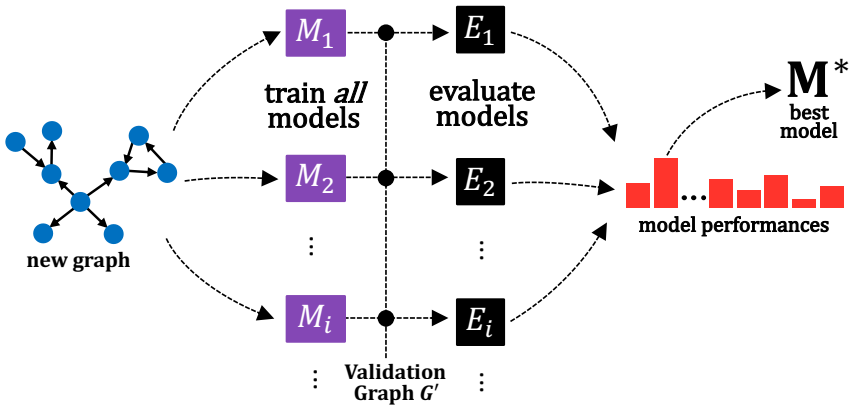
1.1.1.4 Fast and Automatic Model Selection for Graph Representation Learning (Chapter 6)

"Given a graph learning task such as link prediction on a new graph dataset, how can we automatically select the best model without performing any model training or evaluations on the new graph?"

Graph learning (i.e., machine learning on graphs) has been receiving increasing attention in recent years [XSY+21, ZCZ22], and has shown successes across a large array of applications, including traffic forecasting [JL21], ranking [PKD+19], bioinformatics [STZ+20], drug discovery [LCH17], and anomaly detection [CCL+21]. However, as more graph



(a) AUTOGRL is a meta-learned GRL model selection approach.



(b) Costly naive approach for selection of the best GRL model.

Figure 1.6: [Chapter 6] **Our proposed AUTOGRL wins.** Overview of AUTOGRL compared to existing naive approach. (a) Given an unseen graph G and a large space of models $\mathcal{M}$ to search over, AUTOGRL efficiently infers the best model $M^* \in \mathcal{M}$ without ever having to train a single model from $\mathcal{M}$ on the new graph G. (b) This is in contrast to first training each model $M \in \mathcal{M}$, evaluating each one on a hold-out dataset, and then selecting the best model.

learning methods are developed for various tasks, it gets increasingly difficult to determine which method, and also which hyperparameter settings to use for a given graph. Selecting a method and its hyperparameters (i.e., model selection) for graph learning has been largely ad hoc to date. A typical approach is to simply apply popular graph learning models to new graphs. However, it is well known that there is no universal learning algorithm that performs the best on *all* problem instances [WM97], and such consistent model selection is often suboptimal. At the other extreme lies "naive model selection" (Figure 1.6b), where all candidate models are trained on the new graph, and evaluated on a hold-out validation graph, and finally, the best performing model for this new graph is selected. This approach quickly becomes too costly, or even impractical for real-world settings where model selection needs to be done nearly instantaneously as new data continuously arrive.

In this chapter, we develop the *first meta-learning approach for automatic graph representation learning*, called AUTOGRL, which automatically infers a good model for the new graph

without requiring any model training or evaluations, as depicted in Figure 1.6a. AUTOGRL capitalizes on the prior performances of a large body of existing methods on benchmark graph datasets, and carries over this prior experience to automatically select the best model to use for the new graph. To capture the similarity across graphs from different domains, we devise specialized structural meta-graph features that quantify the structural characteristics of a graph. Then we design a meta-graph that represents the relations among models and graphs, and develop a graph meta-learner operating on the meta-graph, which estimates the relevance of each model to different graphs.

Contributions:

- **Problem Formulation.** We formulate the problem of *training and evaluation-free* model selection for graph learning, where model space encompasses a large array of graph learning algorithms and their hyperparameter configurations.
- **Framework for Automatic Graph Learning.** We propose AUTOGRL, the first approach to automatic graph learning to the best of our knowledge, which infers the best graph learning model for a new unseen graph in near real-time, without ever having to run different models as done in traditional model selection.
- **Specialized Meta-Graph Features.** We design specialized meta-graph features for meta-learning on graphs. Meta-graph features effectively capture structural characteristics of a graph, enabling an effective and efficient quantification of graph similarity.
- **Effectiveness and Efficiency.** Through extensive experiments on the benchmark environment that we have built, we show that using AUTOGRL for model selection achieves up to *15× higher mean average precision (MAP)* than consistently applying a popular method like node2vec, as well as obtaining *10% higher MAP* than the best baseline meta-learner. Furthermore, AUTOGRL is highly efficient, *incurring negligible runtime overhead (<1 second)* at inference time.

1.1.2 Part II: Dynamic Graphs and Tensors

Part II focuses on modeling and reasoning with dynamic graphs and tensors, which represent various types of time-evolving networks and dynamic real-world phenomena (such as weather and water quality), namely, knowledge-guided dynamic systems modeling (Section 1.1.2.1), reasoning over temporal knowledge graphs (Section 1.1.2.2), and contrastive graph clustering (Section 1.1.2.3).

1.1.2.1 Knowledge-Guided Dynamic Systems Modeling (Chapter 7)

"How can we model dynamic real-world systems and processes (e.g., weather, water quality, and fluid dynamics)? In particular, how can we discover mathematical expressions that underlie those dynamic processes, guided by prior knowledge?"

Modeling real-world phenomena is the goal of numerous science and engineering endeavors, such as ecological modeling [KMS+10], financial forecasting [LLC09], user modeling [WPB01], and disease prediction [PKP+18], to name a few. Building an accurate model for complex and dynamic systems improves understanding of underlying

Table 1.3: [Chapter 7] Knowledge-guided model revision satisfies all properties for interpretable and effective modeling of complex dynamic systems. Other approaches miss one or more of the properties. "?" means that it depends on the specific method used.

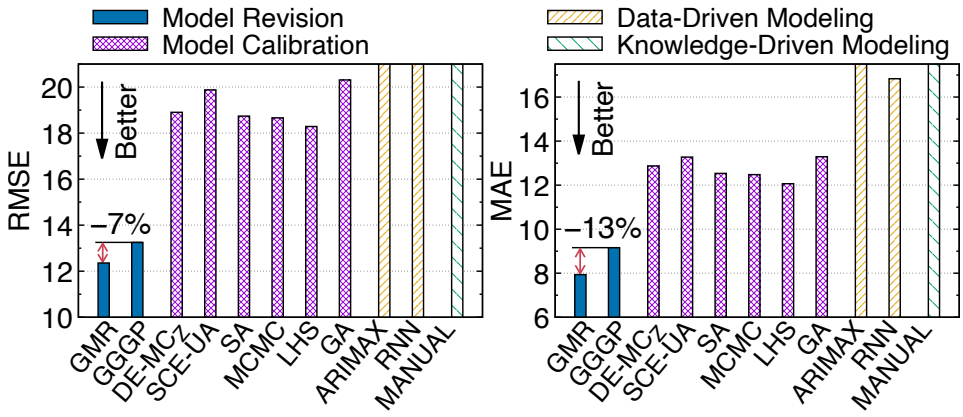
Property \ Approach	Knowledge-Driven Modeling	Data-Driven Modeling	Model Calibration	Model Revision	Knowledge-Guided Model Revision
Learning models consistent with prior knowledge	✓		?		✓
Knowledge-based model specification	✓		✓	✓	✓
Structural model update		?		✓	✓
Automatic tuning of model parameters		✓	✓	✓	✓
Capacity to model complex systems		✓		✓	✓
Interpretable	✓	?	✓	✓	✓

processes and leads to an efficient use of resources. Towards this goal, *knowledge-driven modeling* builds a model based on human expertise, yet it is often suboptimal. At the opposite extreme, *data-driven modeling* aims to learn a model directly from data, without relying on human expertise. However, to model complex system, it requires extensive data and potentially generates overfitting.

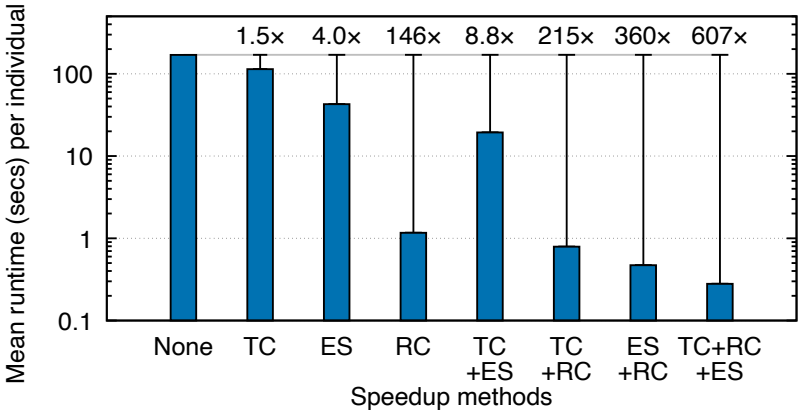
In Chapter 7, we investigate an intermediate approach, called *model revision*, in which prior knowledge and data are combined to achieve the best of both worlds. In model revision, prior knowledge specifies the initial model structure and parameter values, and both are updated iteratively to obtain a better fit to the data. In particular, Chapter 7 proposes *knowledge-guided model revision*, which further improves plain model revision by letting model revision be guided by prior knowledge and producing a revised model consistent with domain knowledge. We achieve effective and efficient knowledge-guided model revision by developing genetic model revision (GMR) framework, which can represent and incorporate prior knowledge of the dynamic systems, while employing several speedup techniques. Table 1.3 summarizes how different approaches satisfy desirable properties for knowledge-guided modeling of complex dynamic systems.

Contributions:

- **Framework.** We present GMR framework for dynamic systems modeling, which improves a knowledge-based model in a data-driven manner, guided by prior knowledge.
- **Knowledge Incorporation.** We design novel mechanisms to represent prior knowledge and perform knowledge-guided optimizations in the GMR framework.
- **River Modeling.** This is the first work to apply model revision to modeling a river system. Previous work on river modeling used model calibration alone.
- **Effectiveness and Efficiency.** GMR framework achieves the best forecasting accuracy in river modeling among a variety of methods, with *up to 34% lower error* than the best parameter fitting approach, while producing models *consistent with domain knowledge* (Figure 1.7a). Also, the proposed speedup techniques effectively cut down the



(a) Forecasting accuracy.



(b) Mean runtime by speedup methods.

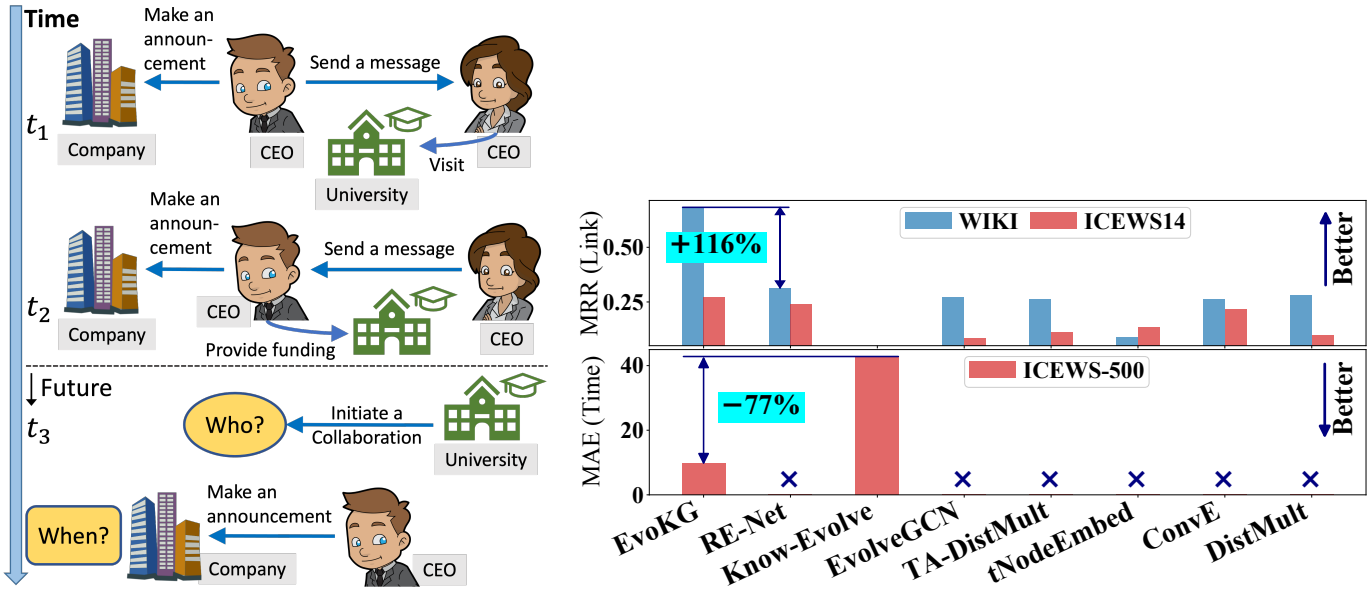
Figure 1.7: [Chapter 7] (a) **Our proposed GMR wins.** Among a variety of methods, GMR achieves the best forecasting accuracy in the river modeling task, while producing revised models consistent with domain knowledge. (b) The proposed speedup techniques lead to $607\times$ speedup.

computational cost, achieving $607\times$ *speedup* (Figure 1.7b).

1.1.2.2 Reasoning over Temporal Knowledge Graphs (Chapter 8)

"How can we perform knowledge reasoning over temporal knowledge graphs (TKGs) that continuously evolve over time?"

TKGs organize and represent facts about entities and their relations, where each fact is associated with a timestamp. Reasoning over TKGs, i.e., inferring new facts from time-evolving KGs, is crucial for many applications to provide intelligent services, including question answering, recommendation, and search. However, despite the prevalence of real-world data that can be represented as TKGs, most methods focus on reasoning over static knowledge graphs, and lack the ability to employ rich temporal dynamics in TKGs.



(a) An example TKG.

(b) Link prediction (top) and time prediction (bottom) accuracy.

Figure 1.8: [Chapter 8] (a) An example TKG, where we aim to predict temporal links and event time. (b) **Our proposed EvoKG wins.** EvoKG achieves the best link prediction (top) and time prediction (bottom) results. × indicates that the corresponding method cannot predict event time.

In Chapter 8, we first present a problem formulation that unifies the two major problems that need to be addressed for an effective reasoning over TKGs, namely, modeling the event time and the evolving network structure. Then we develop EvoKG, an effective framework that jointly models both tasks. EvoKG captures the ever-changing structural and temporal dynamics in TKGs via recurrent event modeling, and models the interactions between entities based on the temporal neighborhood aggregation framework. Further, EvoKG achieves an accurate modeling of event time, using flexible and efficient mechanisms based on neural density estimation.

Contributions:

- **Problem Formulation.** We present a problem formulation that unifies the two major tasks for TKG reasoning—modeling the event time and evolving network structure.
- **Framework.** We propose EvoKG, an effective and efficient method for reasoning over TKGs that jointly addresses the two proposed core problems.
- **Effectiveness and Efficiency.** Experiments show that EvoKG achieves up to *116% and 77% better link and event time prediction accuracy*, respectively, than existing KG reasoning methods (Figure 1.8b). Also, EvoKG performs training and inference up to *30× and 291× times faster*, respectively, than the best existing method.

Table 1.4: [Chapter 9] **Our proposed CGC wins** on features. Comparison of the proposed CGC with deep learning approaches for graph clustering. [A]: Aware of/Utilizing. CL: Clustering, RP: Representation.

Methods / *Desiderata*	AE [HS06]	GAE [KW16]	DAERNN [GCC20]	DAEGC [WPH+19]	SDCN [BWS+20]	AGCN [PLJH21]	**CGC (Ours)**
Jointly optimizing CL and RP				✓	✓	✓	✓
[A] Input node features	✓	✓		✓	✓	✓	✓
[A] Network homophily		✓	✓	✓	✓	✓	✓
[A] Hierarchical communities							✓
Temporal graph clustering			✓				✓
Learning Objective							
Contrastive learning-based							■
Reconstruction-based	■	■	■	■	■	■	

1.1.2.3 Contrastive Graph Clustering for Community Detection and Tracking (Chapter 9)

"Given events between two entities, how can we effectively find communities of entities in an unsupervised manner? Also, when the events are associated with time, how can we detect communities and track their evolution?"

In this chapter, we approach this important task from graph clustering perspective. Recently, state-of-the-art clustering performance in various domains has been achieved by deep clustering methods [XGF16, GGLY17, YFSH17, YZZ+17, YLY+19, MSFK18, LDZ19]. Especially, deep graph clustering (DGC) methods [WPH+19, BWS+20, PHF+20, PLJH21, TGC+14] have successfully extended deep clustering to graph-structured data by learning node representations and cluster assignments in a joint optimization framework. Despite some differences in modeling choices (e.g., encoder architectures), existing DGC methods are mainly based on autoencoders, minimizing reconstruction loss, and use the same clustering objective with relatively minor changes. Also, while many real-world graphs are dynamic in nature, previous DGC methods work only for static graphs.

In this work, we develop CGC, a novel end-to-end framework for graph clustering, which fundamentally differs from existing methods. CGC learns node embeddings and cluster assignments in a contrastive graph learning framework, where positive and negative samples are carefully selected in a multi-level scheme such that they reflect the hierarchical community structures and network homophily. Also, we extend CGC for time-evolving data, where temporal graph clustering is performed in an incremental learning fashion, with the ability to detect change points.

Contributions:

- **Novel Framework.** We propose CGC, a new contrastive graph clustering framework. As discussed above and summarized in Table 1.4, CGC is a significant departure from

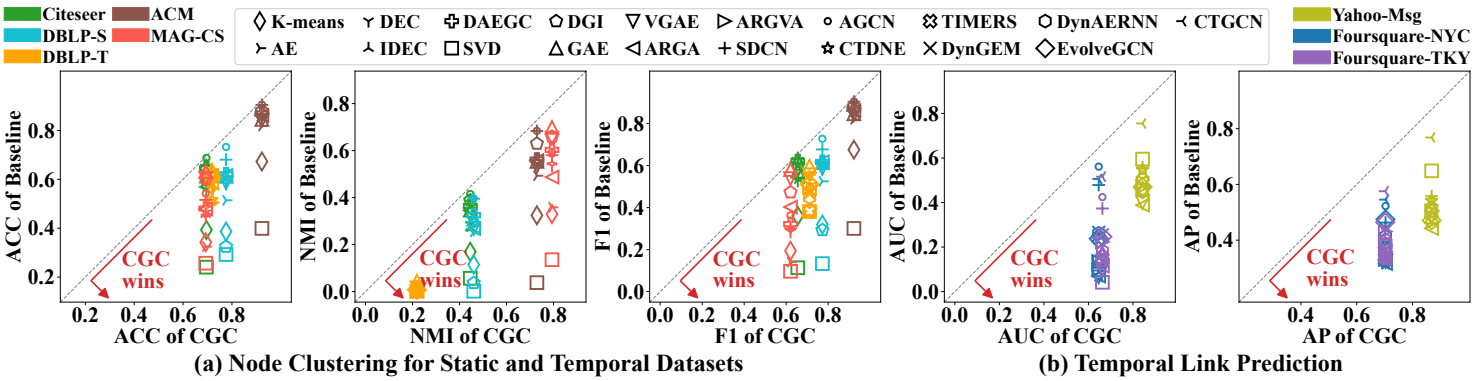


Figure 1.9: [Chapter 9] **Our proposed CGC outperforms competitors**: most points are below the diagonals for all baselines and graphs. CGC achieves more accurate (a) node clustering on static and temporal data, and (b) link prediction based on the time-evolving cluster membership.

previous deep graph clustering methods.

- **Temporal Graph Clustering.** We extend our CGC framework for temporal data. CGC is the first deep graph clustering method for clustering time-evolving networks.
- **Effectiveness.** We demonstrate the effectiveness of CGC via extensive evaluation of clustering quality on both static and temporal real-world datasets, where CGC consistently outperforms various existing methods, achieving up to *14% higher node clustering accuracy*, and *29% higher temporal link prediction accuracy* than the best baseline (Figure 1.9).

Part I

Static Graphs and Tensors

Chapter based on work published in KDD 2019 [PKD+19].

Chapter 2

Estimating Node Importance in Knowledge Graphs Using Graph Neural Networks

How can we estimate the importance of nodes in a knowledge graph (KG)? A KG is a multi-relational graph that has proven valuable for many tasks including question answering and semantic search. In this chapter, we present GENI, a method for tackling the problem of estimating node importance in KGs, which enables several downstream applications such as item recommendation and resource allocation. While a number of approaches have been developed to address this problem for general graphs, they do not fully utilize information available in KGs, or lack flexibility needed to model complex relationship between entities and their importance. To address these limitations, we explore supervised machine learning algorithms. In particular, building upon recent advancement of graph neural networks (GNNs), we develop GENI, a GNN-based method designed to deal with distinctive challenges involved with predicting node importance in KGs. Our method performs an aggregation of importance scores instead of aggregating node embeddings via predicate-aware attention mechanism and flexible centrality adjustment. In our evaluation of GENI and existing methods on predicting node importance in real-world KGs with different characteristics, GENI achieves 5–17% higher NDCG@100 than the state of the art.

2.1 Introduction

Knowledge graphs (KGs) such as Freebase [BEP+08], YAGO [SKW07], and DBpedia [LIJ+15] have proven highly valuable resources for many applications including question answering [DWZX15], recommendation [ZYL+16], semantic search [BWY13], and knowledge

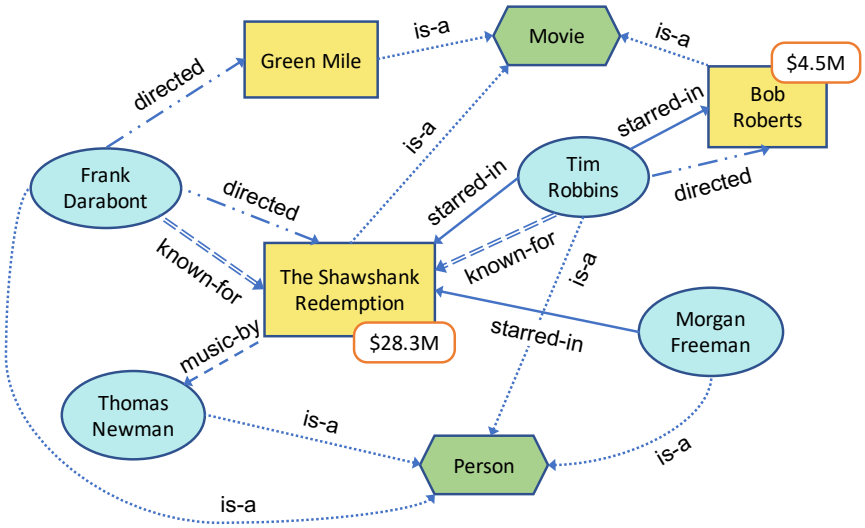


Figure 2.1: An example knowledge graph on movies and related entities. Different edge types represent different types of relations (e.g., "directed" and "starred-in"), and different shapes denote different entity types. Rounded rectangles are importance scores known in advance for some movies.

completion [WGM+14]. A KG is a multi-relational graph where nodes correspond to entities, and edges correspond to relations between the two connected entities. An edge in a KG represents a fact stored in the form of "<subject> <predicate> <object>", (e.g., "<Tim Robbins> <starred-in> <The Shawshank Redemption>"). KGs are different from traditional graphs that have only a single relation; KGs normally consist of multiple, different relations that encode heterogeneous information as illustrated by an example movie KG in Figure 2.1.

Given a KG, estimating the importance of each node is a crucial task that enables a number of applications such as recommendation, query disambiguation, and resource allocation optimization. For example, consider a situation where a customer issues a voice query "Tell me what Genie is" to a voice assistant backed by a KG. If the KG contains several entities with such a name, the assistant could use their estimated importance to figure out which one to describe. Furthermore, many KGs are large-scale, often containing millions to billions of entities for which the knowledge needs to be enriched or updated to reflect the current state. As validating information in KGs requires a lot of resources due to their size and complexity, node importance can be used to guide the system to allocate limited resources for entities of high importance.

How can we estimate the importance of nodes in a KG? In this chapter, we focus on the setting where we are given importance scores of some nodes in a KG. An importance score is a value that represents the significance or popularity of a node in the KG. For example, the number of pageviews of a Wikipedia page can be used as an importance score of the corresponding entity in a KG since important nodes tend to attract a lot of attention and search traffic. Then given a KG, how can we predict node importance by

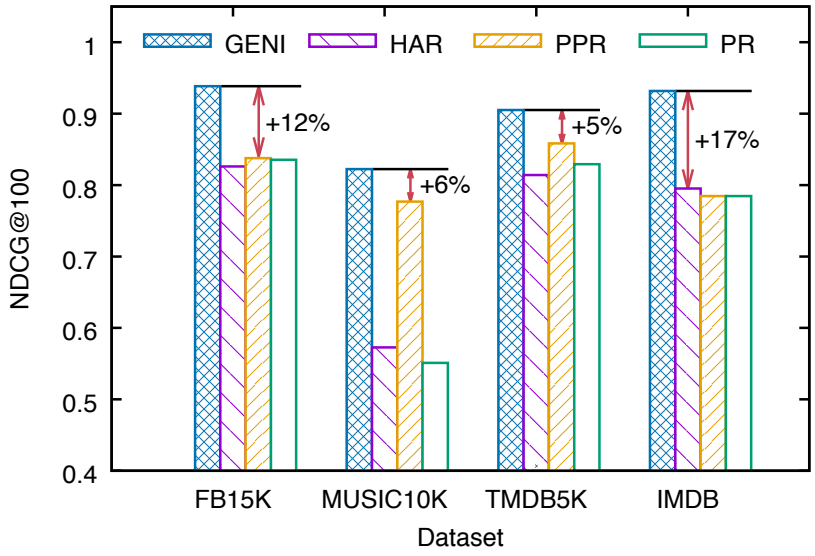


Figure 2.2: Our method GENI outperforms existing methods in predicting importance of nodes in real-world KGs. Higher values are better. See Section 2.4.4 and Table 2.4 for details.

making use of importance scores known for some nodes along with auxiliary information in KGs such as edge types (predicates)?

In the past, several approaches have been developed for node importance estimation. PageRank (PR) [PBMW99] is an early work on this problem that revolutionized the field of Web search. However, PR scores are based only on the graph structure, and unaware of importance scores available for some nodes. Personalized PageRank (PPR) [Hav02] dealt with this limitation by letting users provide their own notion of node importance in a graph. PPR, however, does not take edge types into account. HAR [LNY12] extends ideas used by PR and PPR to distinguish between different predicates in KGs while being aware of importance scores and graph topology. Still, we observe that there is much room for improvement, as evidenced by the performance of existing methods on real-world KGs in Figure 2.2. So far, existing techniques have approached this problem in a non-trainable framework that is based on a fixed model structure determined by their prior assumptions on the propagation of node importance, and involve no learnable parameters that are optimized based on the ground truth.

In this chapter, we explore a new family of solutions for the task of predicting node importance in KGs, namely, regularized supervised machine learning algorithms. Our goal is to develop a more flexible supervised approach that learns from ground truth, and makes use of additional information in KGs. Among several supervised algorithms we explore, we focus on graph neural networks (GNNs). Recently, GNNs have received increasing interests, and achieved state-of-the-art performance on node and graph classification tasks across data drawn from several domains [KW17, DBV16, HYL17, YHC+18, VCC+18]. Designed to learn from graph-structured data, and based on neighborhood aggrega-

tion framework, GNNs have the potential to make further improvements over earlier approaches. However, existing GNNs have focused on graph representation learning via embedding aggregation, and have not been designed to tackle challenges that arise with supervised estimation of node importance in KGs. Challenges include modeling the relationship between the importance of neighboring nodes, accurate estimation that generalizes across different types of entities, and incorporating prior assumptions on node importance that aid model prediction, which are not addressed at the same time by existing supervised techniques.

We present GENI, a GNN for Estimating Node Importance in KGs. GENI applies an attentive GNN for predicate-aware score aggregation to capture relations between the importance of nodes and their neighbors. GENI also allows flexible score adjustment according to node *centrality*, which captures connectivity of a node in terms of graph topology. Our main contributions are as follows.

- We explore regularized supervised machine learning algorithms for estimating node importance in KGs, as opposed to non-trainable solutions where existing approaches belong.
- We present GENI, a GNN-based method designed to address the challenges involved with supervised estimation of node importance in KGs.
- We provide empirical evidence and an analysis of GENI using real-world KGs. Figure 2.2 shows that GENI outperforms the state of the art by 5%-17% percentage points on real KGs.

The rest of this chapter is organized as follows. We present preliminaries in Section 2.2, and describe our method in Section 2.3. After providing experimental results on real KGs in Section 2.4, we review related works in Section 2.5, and conclude in Section 2.6.

2.2 Preliminaries

2.2.1 Problem Definition

A *knowledge graph* (KG) is a graph $G = (V, E = \{E_1, E_2, \ldots, E_P\})$ that represents multi-relational data where nodes V and edges E correspond to entities and their relationships, respectively; P is the number of types of edges (predicates); and E_p denotes a set of edges of type $p \in \{1, \ldots, P\}$. In KGs, there are often many types of predicates (i.e., $P \gg 1$) between nodes of possibly different types (e.g., movie, actor, and director nodes), whereas in traditional graphs, nodes are connected by just one type of edges (i.e., $P = 1$).

An *importance score* $s \in \mathbb{R}_{\geq 0}$ is a non-negative real number that represents the significance or popularity of a node. For example, the total gross of a movie can be used as an importance score for a movie KG, and the number of pageviews of an entity can be used in a more generic KG such as Freebase [BEP+08]. We assume a single set of importance scores, so the scores can compare with each other to reflect importance.

We now define the node importance estimation problem.

Table 2.1: Comparison of methods for estimating node importance. *Neighborhood*: Neighborhood awareness. *Predicate*: Making use of predicates. *Centrality*: Centrality awareness. *Input Score*: Utilizing input importance scores. *Flexibility*: Flexible adaptation.

	GENI	**HAR** [LNY12]	**PPR** [Hav02]	**PR** [PBMW99]
Neighborhood	✓	✓	✓	✓
Predicate	✓	✓		
Centrality	✓	✓	✓	✓
Input Score	✓	✓	✓	
Flexibility	✓			

Definition 1. Node Importance Estimation: Given a KG $G = (V, E = \{E_1, E_2, \ldots, E_P\})$ and importance scores $\{s\}$ for a subset $V_s \subseteq V$ of nodes, learn a function $S : V \to [0, \infty)$ that estimates the importance score of every node in KG.

Figure 2.1 shows an example KG on movies and related entities with importance scores given in advance for some movies. We approach the importance estimation problem by developing a supervised framework learning a function that maps any node in KG to its score, such that the estimation reflects its true importance as closely as possible.

Note that even when importance scores are provided for only one type of nodes (e.g., movies), we aim to do estimation for all types of nodes (e.g,. directors, actors, etc.).

Definition 2. In-Domain and Out-Of-Domain Estimation: Given importance scores for some nodes $V_s \subseteq V$ of type $\mathcal{T}$ (e.g., movies), predicting the importance of nodes of type $\mathcal{T}$ is called an *"in-domain"* estimation, and importance estimation for those nodes whose type is not $\mathcal{T}$ (e.g., actors) is called an *"out-of-domain"* estimation.

As available importance scores are often limited in terms of numbers and types, developing a method that generalizes well for both classes of estimation is an important challenge for supervised node importance estimation.

2.2.2 Desiderata for Modeling Node Importance in KGs

Based on our discussion on prior approaches (PR, PPR, and HAR), we present the desiderata that have guided the development of our method for tackling node importance estimation problem. Table 2.1 summarizes GENI and existing methods in terms of these desiderata.

Neighborhood Awareness. In a graph, a node is connected to other nodes, except for the special case of isolated nodes. As neighboring entities interact with each other, and they tend to share common characteristics (network homophily), neighborhoods should be taken into account when node importance is modeled.

Making Use of Predicates. KGs consist of multiple types of predicates. Under the assumption that different predicates could play a different role in determining node

importance, models should make predictions using information from predicates.

Centrality Awareness. Without any other information, it is reasonable to assume that highly central nodes are more important than less central ones. Therefore, scores need to be estimated in consideration of node centrality, capturing connectivity of a node.

Utilizing Input Importance Scores. In addition to graph topology, input importance scores provide valuable information to infer relationships between nodes and their importance. Thus, models should tap into both the graph structure and input scores for more accurate prediction.

Flexible Adaptation. Our assumption regarding node importance such as the one on centrality may not conform to the real distribution of input scores over KGs. Also, we do not limit models to a specific type of input scores. On the other hand, models can be provided with input scores that possess different characteristics. It is thus critical that a model can flexibly adapt to the importance that input scores reflect.

2.2.3 Graph Neural Networks

In this section, we present a generic definition of graph neural networks (GNNs). GNNs are mainly based on neighborhood aggregation architecture [KW17, HYL17, GSR+17, YHC+18, VCC+18]. In a GNN with L layers, its ℓ-th layer ($\ell = 1, \ldots, L$) receives a feature vector $\vec{h}_i^{\ell-1}$ for each node i from the $(\ell-1)$-th layer (where $\vec{h}_i^0$ is an input node feature $\vec{z}_i$), and updates it by aggregating feature vectors from the neighborhood $\mathcal{N}(i)$ of node i, possibly using a different weight $w_{i,j}^\ell$ for neighbor j. As updated feature vectors become the input to the $(\ell+1)$-th layer, repeated aggregation procedure through L layers in principle captures L-th order neighbors in learning a node's representation. This process of learning representation $\vec{h}_i^\ell$ of node i by ℓ-th layer is commonly expressed as [HYL17, YHC+18, XLT+18]:

$$\vec{h}_{\mathcal{N}(i)}^\ell \leftarrow \textsc{Transform}^\ell \left(\textsc{Aggregate} \left(\left\{ \left(\vec{h}_j^{\ell-1}, w_{i,j}^\ell \right) \mid j \in \mathcal{N}(i) \right\} \right) \right) \tag{2.1}$$

$$\vec{h}_i^\ell \leftarrow \textsc{Combine} \left(\vec{h}_i^{\ell-1}, \vec{h}_{\mathcal{N}(i)}^\ell \right) \tag{2.2}$$

where Aggregate is an aggregation function defined by the model (e.g., averaging or max-pooling operation); Transform is a model-specific function that performs a (non-linear) transformation of node embeddings via parameters in ℓ-th layer shared by all nodes (e.g., multiplication with a shared weight matrix $\mathbf{W}^\ell$ followed by some non-linearity $\sigma(\cdot)$); Combine is a function that merges the aggregated neighborhood representation with the node's representation (e.g., concatenation).

2.3 Method

Effective estimation of node importance in KGs involves addressing the requirements presented in Section 2.2.2. As a supervised learning method, the GNN framework naturally allows us to *utilize input importance scores* to train a model with *flexible adaptation*.

Table 2.2: Table of symbols.

Symbol	Definition
V_s	set of nodes with known importance scores
$\vec{z}_i$	real-valued feature vector of node i
$\mathcal{N}(i)$	neighbors of node i
L	total number of score aggregation (SA) layers
ℓ	index for an SA layer
H^ℓ	number of SA heads in ℓ-th layer
p_{ij}^m	predicate of m-th edge between nodes i and j
$\phi(e)$	learnable embedding of predicate e
σ_a, σ_s	non-linearities for attention computation and score estimation
$s_h^\ell(i)$	estimated score of node i by h-th SA head in ℓ-th layer
$s^*(i)$	centrality-adjusted score estimation of node i
$\|\|$	concatenation operator
$d(i)$	in-degree of node i
$c(i)$	centrality score of node i
$c_h^*(i)$	centrality score of node i scaled and shifted by h-th SA head
γ_h, β_h	learnable scale and shift parameters used by h-th SA head
$\vec{a}_{h,\ell}$	learnable parameter vector to compute $\alpha_{ij}^{h,\ell}$ by h-th SA head in ℓ-th layer
$\alpha_{ij}^{h,\ell}$	node i's attention on node j computed with h-th SA head in ℓ-th layer
$g(i)$	known importance score of node i

Its propagation mechanism also allows us to be *neighborhood aware*. In this section, we present GENI, which further enhances the model in three ways.

- *Neighborhood Importance Awareness*: GNN normally propagates information between neighbors through node embedding. This is to model the assumption that an entity and its neighbors affect each other, and thus the representation of an entity can be better represented in terms of the representation of its neighbors. In the context of node importance estimation, neighboring importance scores play a major role on the importance of a node, whereas other neighboring features may have little effect, if any. We thus directly aggregate importance scores from neighbors (Section 2.3.1), and show empirically that it outperforms embedding propagation (Section 2.4.4).
- *Making Use of Predicates*: We design predicate-aware attention mechanism that models how predicates affect the importance of connected entities (Section 2.3.2).
- *Centrality Awareness*: We apply centrality adjustment to incorporate node centrality into the estimation (Section 2.3.3).

An overview of GENI is provided in Figure 2.3. In Sections 2.3.1 to 2.3.3, we describe the three main enhancements using the basic building blocks of GENI shown in Figure 2.3a. Then we discuss an extension to a general architecture in Section 2.3.4. Table 2.2 provides the definition of symbols used in this chapter.

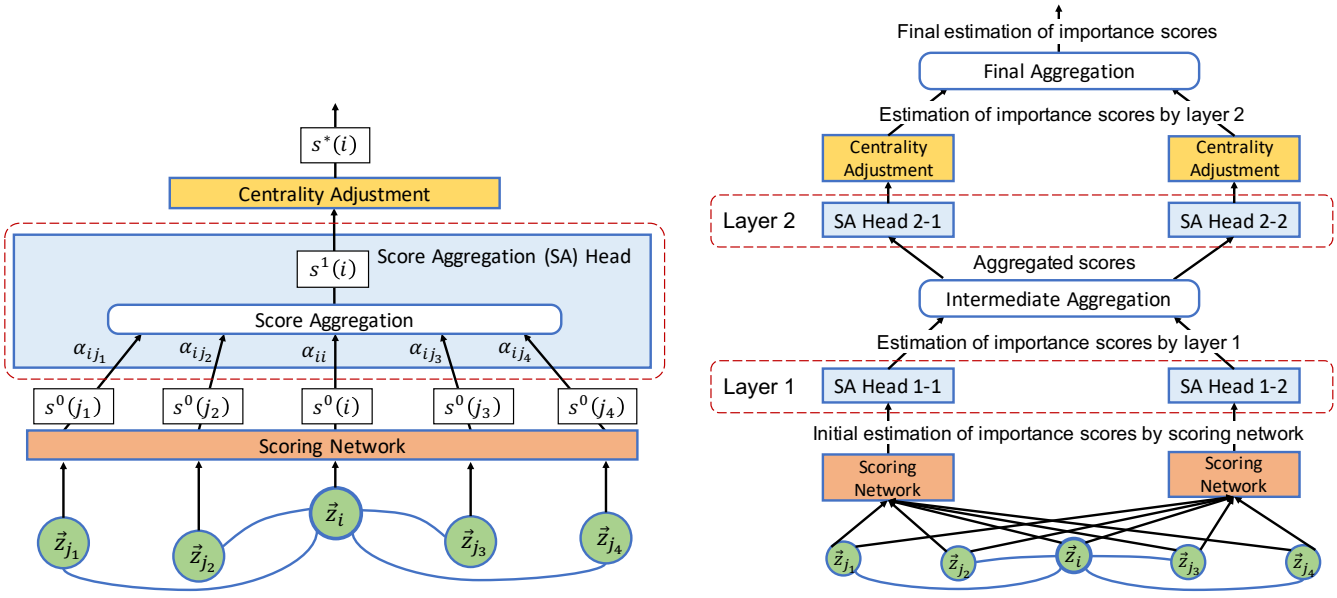


Figure 2.3: Description of node importance estimation by GENI. a: Estimation of the importance of node i based on the embeddings of node i and its neighbors $j_1, \ldots, j_4$ (connected by blue edges). The final estimation $s^*(i)$ is produced via three components of GENI shown in colored boxes, which are described in Sections 2.3.1 to 2.3.3. b: An illustration of the proposed model that consists of two layers, each of which contains two score aggregation heads. Note that the model can consist of different numbers of layers, and each layer can also have different numbers of score aggregation heads. A discussion on the extension of the basic model in a to a more comprehensive architecture in b is given in Section 2.3.4.

2.3.1 Score Aggregation

To directly model the relationship between the importance of neighboring nodes, we propose a score aggregation framework, rather than embedding aggregation. Specifically, in Equations (2.1) and (2.2), we replace the hidden embedding $\vec{h}_j^{\ell-1}$ of node j with its score estimation $s^{\ell-1}(j)$ and combine them as follows:

$$s^{\ell}(i) = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{ij}^{\ell} \, s^{\ell-1}(j) \tag{2.3}$$

where $\mathcal{N}(i)$ denotes the neighbors of node i, which will be a set of the first-order neighbors of node i in our experiments. Here, α_{ij}^{ℓ} is a learnable weight between nodes i and j for the ℓ-th layer ($\ell = 1, \ldots, L$). We train it via a shared attention mechanism which is computed by a pre-defined model with shared parameters and predicate embeddings, as we explain soon. In other words, GENI computes the aggregated score $s^{\ell}(i)$ by performing a weighted aggregation of intermediate scores from node i and its neighbors. Note that GENI does not apply TRANSFORM$^{\ell}$ function after aggregation as in Equation (2.1),

since GENI aggregates scores. Propagating scores instead of node embeddings has the additional benefit of reducing the number of model parameters.

To compute the initial estimation $s^0(i)$, GENI uses input node features. In the simplest case, they can be one-hot vectors that represent each node. More generally, they are real-valued vectors representing the nodes, which are extracted manually based on domain knowledge, or generated with methods for learning node embeddings. Let $\vec{z}_i$ be the input feature vector of node i. Then GENI computes the initial score of i as

$$s^0(i) = \textsc{ScoringNetwork}(\vec{z}_i) \tag{2.4}$$

where SCORINGNETWORK can be any neural network that takes in a node feature vector and returns an estimation of its importance. We used a simple fully-connected neural network for our experiments.

2.3.2 Predicate-Aware Attention Mechanism

Inspired by recent work that showcased successful application of attention mechanism, we employ a predicate-aware attention mechanism that attends over the neighbor's intermediate scores.

Our attention considers two factors. First, we consider the predicate between the nodes because different predicates can play different roles for score propagation. For example, even though a movie may be released in a popular (i.e., important) country, the movie itself may not be popular; on the other hand, a movie directed by a famous (i.e., important) director is more likely to be popular. Second, we consider the neighboring score itself in deciding the attention. A director who directed a few famous (i.e., important) movies is likely to be important; the fact that he also directed some not-so-famous movies in his life is less likely to make him unimportant.

GENI incorporates predicates into attention computation by using shared predicate embeddings; i.e., each predicate is represented by a feature vector of predefined length, and this representation is shared by nodes across all layers. Further, predicate embeddings are learned so as to maximize the predictive performance of the model in a flexible fashion. Note that in KGs, there could be multiple edges of different types between two nodes (e.g., see Figure 2.1). We use p_{ij}^m to denote the predicate of m-th edge between nodes i and j, and $\phi(\cdot)$ to denote a mapping from a predicate to its embedding.

In GENI, we use a simple, shared self-attention mechanism, which is a single layer feedforward neural network parameterized by the weight vector $\vec{a}$. Relation between the intermediate scores of two nodes i and j, and the role an in-between predicate plays are captured by the attentional layer that takes in the concatenation of all relevant information. Outputs from the attentional layer are first transformed by non-linearity $\sigma(\cdot)$, and then normalized via the softmax function. Formally, GENI computes the attention α_{ij}^ℓ of node i on node j for ℓ-th layer as:

$$\alpha_{ij}^\ell = \frac{\exp\left(\sigma_a\left(\sum_m \vec{a}_\ell^\top [s^{\ell-1}(i)||\phi(p_{ij}^m)||s^{\ell-1}(j)]\right)\right)}{\sum_{k\in\mathcal{N}(i)\cup\{i\}} \exp\left(\sigma_a\left(\sum_m \vec{a}_\ell^\top [s^{\ell-1}(i)||\phi(p_{ik}^m)||s^{\ell-1}(k)]\right)\right)} \tag{2.5}$$

where σ_a is a non-linearity, $\vec{a}_\ell$ is a weight vector for ℓ-th layer, and $||$ is a concatenation operator.

2.3.3 Centrality Adjustment

Existing methods such as PR, PPR, and HAR make a common assumption that the importance of a node positively correlates with its centrality in the graph. In the context of KGs, it is also natural to assume that more central nodes would be more important than less central ones, unless the given importance scores present contradictory evidence. Making use of this prior knowledge becomes especially beneficial in cases where we are given a small number of importance scores compared to the total number of entities, and in cases where the importance scores are given for entities of a specific type out of the many types in KG.

Given that the in-degree $d(i)$ of node i is a common proxy for its centrality and popularity, we define the initial centrality $c(i)$ of node i to be

$$c(i) = \log(d(i) + \epsilon) \tag{2.6}$$

where ϵ is a small positive constant.

While node centrality provides useful information on the importance of a node, strictly adhering to the node centrality could have a detrimental effect on model prediction. We need flexibility to account for the possible discrepancy between the node's centrality in a given KG and the provided input importance score of the node. To this end, we use a scaled and shifted centrality $c^*(i)$ as our notion of node centrality:

$$c^*(i) = \gamma \cdot c(i) + \beta \tag{2.7}$$

where γ and β are learnable parameters for scaling and shifting. As we show in Section 2.4.5, this flexibility allows better performance when in-degree is not the best proxy of centrality.

To compute the final score, we apply centrality adjustment to the score estimation $s^L(i)$ from the last layer, and apply a non-linearity σ_s as follows:

$$s^*(i) = \sigma_s \left(c^*(i) \cdot s^L(i) \right) \tag{2.8}$$

2.3.4 Model Architecture

The simple architecture depicted in Figure 2.3a consists of a scoring network and a single score aggregation (SA) layer (i.e., $L = 1$), followed by a centrality adjustment component. Figure 2.3b extends it to a more general architecture in two ways. First, we extend the framework to contain multiple SA layers; that is, $L > 1$. As a single SA layer aggregates the scores of direct neighbors, stacking multiple SA layers enables aggregating scores from a larger neighborhood. Second, we design each SA layer to contain a variable number of SA heads, which perform score aggregation and attention

computation independently of each other. Empirically, we find using multiple SA heads to be helpful for the model performance and the stability of optimization procedure (Section 2.4.5).

Let h be an index of an SA head, and H^ℓ be the number of SA heads in ℓ-th layer. We define $s'^{\ell-1}_h(i)$ to be node i's score that is estimated by $(\ell-1)$-th layer, and fed into h-th SA head in ℓ-th (i.e., the next) layer, which in turn produces an aggregation $s^\ell_h(i)$ of these scores:

$$s^\ell_h(i) = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha^{h,\ell}_{ij} \; s'^{\ell-1}_h(j) \tag{2.9}$$

where $\alpha^{h,\ell}_{ij}$ is the attention coefficient between nodes i and j computed by SA head h in layer ℓ.

In the first SA layer, each SA head h receives input scores from a separate scoring network SCORINGNETWORK_h, which provides the initial estimation $s^0_h(i)$ of node importance. For the following layers, output from the previous SA layer becomes the input estimation. Since in ℓ-th $(\ell \geq 1)$ SA layer, H^ℓ SA heads independently produce H^ℓ score estimations in total, we perform an aggregation of these scores by averaging, which is provided to the next layer. That is,

$$s'^{\ell}_h(i) = \begin{cases} \text{SCORINGNETWORK}_h(\vec{z}_i) & \text{if } \ell = 0 \\ \text{AVERAGE}\left(\left\{ s^\ell_h(i) \;\middle|\; h = 1, \ldots, H^\ell \right\}\right) & \text{if } \ell \geq 1 \end{cases} \tag{2.10}$$

where $\vec{z}_i$ denotes the input feature vector of node i.

Multiple SA heads in ℓ-th layer compute attention between neighboring nodes in the same way as in Equation (2.5), yet independently of each other using its own parameters $\vec{a}_{h,\ell}$:

$$\alpha^{h,\ell}_{ij} = \frac{\exp\left(\sigma_a\left(\sum_m \vec{a}^\top_{h,\ell}[s'^{\ell-1}_h(i) || \phi(p^m_{ij}) || s'^{\ell-1}_h(j)]\right)\right)}{\sum_{k \in \mathcal{N}(i)\cup\{i\}} \exp\left(\sigma_a\left(\sum_m \vec{a}^\top_{h,\ell}[s'^{\ell-1}_h(i) || \phi(p^m_{ik}) || s'^{\ell-1}_h(k)]\right)\right)} \tag{2.11}$$

Centrality adjustment is applied to the output from the final SA layer. In order to enable independent scaling and shifting by each SA head, separate parameters γ_h and β_h are used for each head h. Then centrality adjustment by h-th SA head in the final layer is:

$$c^*_h(i) = \gamma_h \cdot c(i) + \beta_h \tag{2.12}$$

With H^L SA heads in the final L-th layer, we perform additional aggregation of centrality-adjusted scores by averaging, and apply a non-linearity σ_s, obtaining the final estimation $s^*(i)$:

$$s^*(i) = \sigma_s\left(\text{AVERAGE}\left(\left\{ c^*_h(i) \cdot s^L_h(i) \;\middle|\; h = 1, \ldots, H^L \right\}\right)\right) \tag{2.13}$$

Table 2.3: Real-world KGs. See Section 2.4.1 and Section 2.7.1 for details. SCC: Strongly connected component. OOD: Out-of-domain.

Name	# Nodes	# Edges	# Predicates	# SCCs.	Input Score Type	# Nodes w/ Scores	Data for OOD Evaluation
FB15K	14,951	592,213	1,345	9	# Pageviews	14,108 (94%)	N/A
MUSIC10K	24,830	71,846	10	130	Song hotttnesss	4,214 (17%)	Artist hotttnesss
TMDB5K	123,906	532,058	22	15	Movie popularity	4,803 (4%)	Director ranking
IMDB	1,567,045	14,067,776	28	1	# Votes for movies	215,769 (14%)	Director ranking

2.3.5 Model Training

In order to predict node importance with input importance scores known for a subset of nodes $V_s \subseteq V$, we train GENI using mean squared error between the given importance score $g(i)$ and the model estimation $s^*(i)$ for node $i \in V_s$; thus, the loss function is

$$\frac{1}{|V_s|} \sum_{i \in V_s} (s^*(i) - g(i))^2 \tag{2.14}$$

Note that SCORINGNETWORK is trained jointly with the rest of GENI. To avoid overfitting, we apply weight decay with an early stopping criterion based on the model performance on validation entities.

2.4 Experiments

In this section, we aim to answer the following questions.

- How do GENI and baselines perform on real-world KGs with different characteristics? In particular, how well do methods perform in- and out-of-domain estimation (Definition 2)?
- How do the components of GENI, such as centrality adjustment, and different parameter values affect its estimation?

We describe datasets, baselines, and evaluation plans in Sections 2.4.1 to 2.4.3, and answer the above questions in Sections 2.4.4 and 2.4.5.

2.4.1 Datasets

In our experiments, we use four real-world KGs with different characteristics. Here we introduce these KGs along with the importance scores used for in- and out-of-domain (OOD) evaluations (see Definition 2). Summaries of the datasets (such as the number of nodes, edges, and predicates) are given in Table 2.3. More details such as data sources and how they are constructed can be found in Section 2.7.1.

FB15K is a subset of Freebase, which is a large collaborative knowledge base containing general facts, and has been widely used for research and practical applications [BUG+13, BEP+08]. FB15K has a much larger number of predicates and a higher density than other KGs we evaluated. For each entity, we use the number of pageviews for the

corresponding Wikipedia page as its score. Note that we do not perform OOD evaluation for FB15K since importance scores for FB15K apply to all types of entities.

MUSIC10K is a music KG sampled from the Million Song Dataset[1], which includes information about songs such as the primary artist and the album the song belongs to. The dataset provides two types of popularity scores called "song hotttnesss" and "artist hotttnesss" computed by the Echo Nest platform by considering data from many sources such as mentions on the web, play counts, etc[2]. We use "song hotttnesss" as input importance scores, and "artist hotttnesss" for OOD performance evaluation.

TMDB5K is a movie KG derived from the TMDb 5000 movie dataset[3]. It contains movies and related entities such as movie genres, companies, countries, crews, and casts. We use the "popularity" information for movies as importance scores, which is provided by the original dataset. For OOD evaluation, we use a ranking of top-200 highest grossing directors[4]. Worldwide box office grosses given in the ranking are used as importance scores for directors.

IMDB is a movie KG created from the public IMDb dataset, which includes information such as movies, genres, directors, casts, and crews. IMDB is the largest KG among those we evaluate, with $12.6\times$ as many nodes as TMDB5K. IMDb dataset provides the number of votes a movie received, which we use as importance scores. For OOD evaluation, we use the same director ranking used for TMDB5K.

2.4.2 Baselines

Methods for node importance estimation in KGs can be classified into two families of algorithms.

Non-Trainable Approaches. Previously developed methods mostly belong to this category. We evaluate the following methods:

- PageRank (PR) [PBMW99]
- Personalized PageRank (PPR) [Hav02]
- HAR [LNY12]

Supervised Approaches. We explore the performance of representative supervised algorithms on node importance estimation:

- Linear regression (LR): an ordinary least squares algorithm.
- Random forests (RF): a random forest regression model.
- Neural networks (NN): a fully-connected neural network.
- Graph attention networks (GAT) [VCC+18]: This is a GNN model reviewed in Section 2.2.3. We add a final layer that takes the node embedding and outputs the

[1] https://labrosa.ee.columbia.edu/millionsong/
[2] https://musicmachinery.com/tag/hotttnesss/
[3] https://www.kaggle.com/tmdb/tmdb-movie-metadata
[4] https://www.the-numbers.com/box-office-star-records/worldwide/lifetime-specific-technical-role/director

importance score of a node.

All these methods and GENI use the same data (node features and input importance scores). In our experiments, node features are generated using node2vec [GL16]. Depending on the type of KGs, other types of node features, such as bag-of-words representation, can also be used. Note that the graph structure is explicitly used only by GAT, although other supervised baselines make an implicit use of it when node features encode graph structural information.

We will denote each method by the name in parentheses. Experimental settings for baselines and GENI are provided in Section 2.7.2.

2.4.3 Performance Evaluation

We evaluate methods based on their in- and out-of-domain (OOD) performance. We performed 5-fold cross validation, and report the average and standard deviation of the following metrics on ranking quality and correlation: normalized discounted cumulative gain and Spearman correlation coefficient. Higher values are better for all metrics. We now provide their formal definitions.

Normalized discounted cumulative gain (NDCG) is a measure of ranking quality. Given a list of nodes ranked by predicted scores, and their graded relevance values (which are non-negative, real-valued ground truth scores in our setting), discounted cumulative gain at position k ($DCG@k$) is defined as:

$$DCG@k = \sum_{i=1}^{k} \frac{r_i}{\log_2(i+1)} \tag{2.15}$$

where r_i denotes the graded relevance of the node at position i. Note that due to the logarithmic reduction factor, the gain r_i of each node is penalized at lower ranks. Consider an ideal DCG at rank position k ($IDCG@k$) which is obtained by an ideal ordering of nodes based on their relevance scores. Normalized DCG at position k ($NDCG@k$) is then computed as:

$$NDCG@k = \frac{DCG@k}{IDCG@k} \tag{2.16}$$

Our motivation for using $NDCG@k$ is to test the quality of ranking for the top k entities.

Spearman correlation coefficient (SPEARMAN) measures the rank correlation between the ground truth scores $\vec{g}$ and predicted scores $\vec{s}$; that is, the strength and direction of the monotonic relationship between the rank values of $\vec{g}$ and $\vec{s}$. Converting $\vec{g}$ and $\vec{s}$ into ranks $\vec{g}_r$ and $\vec{s}_r$, respectively, Spearman correlation coefficient is computed as:

$$Spearman = \frac{\sum_i (g_{ri} - \overline{g}_r)(s_{ri} - \overline{s}_r)}{\sqrt{\sum_i (g_{ri} - \overline{g}_r)^2} \sqrt{\sum_i (s_{ri} - \overline{s}_r)^2}} \tag{2.17}$$

Table 2.4: In-domain prediction results on real-world datasets. GENI consistently outperforms all baselines. Numbers after ± symbol are standard deviation from 5-fold cross validation. Best results are in bold, and second best results are underlined.

Method	FB15K		MUSIC10K		TMDB5K		IMDB	
	NDCG@100	SPEARMAN	NDCG@100	SPEARMAN	NDCG@100	SPEARMAN	NDCG@100	SPEARMAN
PR	0.8354 ± 0.016	0.3515 ± 0.015	0.5510 ± 0.021	-0.0926 ± 0.034	0.8293 ± 0.026	0.5901 ± 0.011	0.7847 ± 0.048	0.0881 ± 0.004
PPR	0.8377 ± 0.015	0.3667 ± 0.015	0.7768 ± 0.009	0.3524 ± 0.046	0.8584 ± 0.013	$\underline{0.7385 \pm 0.010}$	0.7847 ± 0.048	0.0881 ± 0.004
HAR	0.8261 ± 0.005	0.2020 ± 0.012	0.5727 ± 0.017	0.0324 ± 0.044	0.8141 ± 0.021	0.4976 ± 0.014	0.7952 ± 0.036	0.1318 ± 0.005
LR	0.8750 ± 0.005	0.4626 ± 0.019	0.7301 ± 0.023	0.3069 ± 0.032	0.8743 ± 0.015	0.6881 ± 0.013	0.7365 ± 0.009	0.5013 ± 0.002
RF	0.8734 ± 0.005	0.5122 ± 0.019	$\underline{0.8129 \pm 0.012}$	$\underline{0.4577 \pm 0.012}$	0.8503 ± 0.016	0.5959 ± 0.022	0.7651 ± 0.010	0.4753 ± 0.005
NN	0.9003 ± 0.005	0.6031 ± 0.012	0.8015 ± 0.017	0.4491 ± 0.027	0.8715 ± 0.006	0.7009 ± 0.009	0.8850 ± 0.016	0.5120 ± 0.008
GAT	$\underline{0.9205 \pm 0.009}$	$\underline{0.7054 \pm 0.013}$	0.7666 ± 0.016	0.4276 ± 0.023	$\underline{0.8865 \pm 0.011}$	0.7180 ± 0.010	$\underline{0.9110 \pm 0.011}$	$\underline{0.7060 \pm 0.007}$
GENI	$\mathbf{0.9385 \pm 0.004}$	$\mathbf{0.7772 \pm 0.006}$	$\mathbf{0.8224 \pm 0.018}$	$\mathbf{0.4783 \pm 0.009}$	$\mathbf{0.9051 \pm 0.005}$	$\mathbf{0.7796 \pm 0.009}$	$\mathbf{0.9318 \pm 0.005}$	$\mathbf{0.7387 \pm 0.002}$

where $\overline{g}_r$ and $\overline{s}_r$ are the mean of $\vec{g}_r$ and $\vec{s}_r$.

For in-domain evaluation, we use NDCG@100 and SPEARMAN as they complement each other: NDCG@100 looks at the top-100 predictions, and SPEARMAN considers the ranking of all entities with known scores. For NDCG, we also tried different cut-off thresholds and observed similar results. Note that we often have a small volume of data for OOD evaluation. For example, for TMDB5K and IMDB, we used a ranking of 200 directors with known scores, while TMDB5K and IMDB have 2,578 and 287,739 directors, respectively. Thus SPEARMAN is not suitable for OOD evaluation as it considers only those small number of entities in the ranking, and ignores all others, even if they are predicted to be highly important; thus, for OOD evaluation, we report NDCG@100 and NDCG@2000.

Additionally, we report regression performance in Section 2.7.3.2.

2.4.4 Importance Estimation on Real-World Data

We evaluate GENI and baselines in terms of in- and out-of-domain (OOD) predictive performance.

2.4.4.1 In-Domain Prediction

Table 2.4 summarizes in-domain prediction performance. GENI outperforms all baselines on four datasets in terms of both NDCG@100 and SPEARMAN. It is noteworthy that supervised approaches generally perform better in-domain prediction than non-trainable ones, especially on FB15K and IMDB, which are more complex and larger than the other two. It demonstrates the applicability of supervised models to our problem. On all KGs except MUSIC10K, GAT outperforms other supervised baselines, which use the same node features but do not explicitly take the graph network structure into account. This shows the benefit of directly utilizing network connectivity. By modeling the relation between scores of neighboring entities, GENI achieves further performance improvement

Table 2.5: Out-of-domain prediction results on real-world datasets. GENI consistently outperforms all baselines. Numbers after ± symbol are standard deviation from 5-fold cross validation. Best results are in bold, and second best results are underlined.

Method	MUSIC10K		TMDB5K		IMDB	
	NDCG@100	NDCG@2000	NDCG@100	NDCG@2000	NDCG@100	NDCG@2000
PR	0.6520 ± 0.000	0.8779 ± 0.000	0.8337 ± 0.000	0.8079 ± 0.000	0.0000 ± 0.000	0.1599 ± 0.000
PPR	$\underline{0.7324 \pm 0.006}$	$\underline{0.9118 \pm 0.002}$	0.8060 ± 0.041	0.7819 ± 0.022	0.0000 ± 0.000	0.1599 ± 0.000
HAR	0.7113 ± 0.004	0.8982 ± 0.001	$\underline{0.8913 \pm 0.010}$	$\underline{0.8563 \pm 0.007}$	0.2551 ± 0.019	0.3272 ± 0.005
LR	0.6644 ± 0.006	0.8667 ± 0.001	0.4990 ± 0.013	0.5984 ± 0.002	0.3064 ± 0.007	0.2755 ± 0.003
RF	0.6898 ± 0.022	0.8796 ± 0.003	0.5993 ± 0.040	0.6236 ± 0.005	$\underline{0.4066 \pm 0.145}$	0.3719 ± 0.040
NN	0.6981 ± 0.017	0.8836 ± 0.005	0.5675 ± 0.023	0.6172 ± 0.009	0.2158 ± 0.035	0.3105 ± 0.019
GAT	0.6909 ± 0.009	0.8834 ± 0.003	0.5349 ± 0.016	0.5999 ± 0.007	0.3858 ± 0.065	$\underline{0.4209 \pm 0.016}$
GENI	$\mathbf{0.7964 \pm 0.007}$	$\mathbf{0.9121 \pm 0.002}$	$\mathbf{0.9078 \pm 0.004}$	$\mathbf{0.8776 \pm 0.002}$	$\mathbf{0.4519 \pm 0.051}$	$\mathbf{0.4962 \pm 0.025}$

over GAT. Among non-trainable baselines, HAR often performs worse than PR and PPR, which suggests that considering predicates could hurt performance if predicate weight adjustment is not done properly.

2.4.4.2 Out-Of-Domain Prediction

Table 2.5 summarizes OOD prediction results. GENI achieves the best results for all KGs in terms of both NDCG@100 and NDCG@2000. In contrast to in-domain prediction where supervised baselines generally outperform non-trainable ones, we observe that non-trainable methods achieve higher OOD results than supervised baselines on MUSIC10K and TMDB5K. In these KGs, only about 4,000 entities have known scores. Given scarce ground truth, non-trainable baselines could perform better by relying on a prior assumption on the propagation of node importance. Further, note that the difference between non-trainable and supervised baselines is more drastic on TMDB5K where the proportion of nodes with scores is the smallest (4%). On the other hand, on IMDB, which is our largest KG with the greatest number of ground truth, supervised baselines mostly outperform non-trainable methods. In particular, none of the top-100 directors in IMDB predicted by PR and PPR belong to the ground truth director ranking. With 14% of nodes in IMDB associated with known scores, supervised methods learn to generalize better for OOD prediction. Although neighborhood aware, GAT is not better than other supervised baselines. By applying centrality adjustment, GENI achieves superior performance to both classes of baselines regardless of the number of available known scores.

2.4.5 Analysis of GENI

2.4.5.1 Effect of Considering Predicates

To see how the consideration of predicates affects model performance, we run GENI on FB15K, which has the largest number of predicates, and report NDCG@100 and SPEARMAN when a single embedding is used for all predicates (denoted by "shared embedding") vs. when each predicate uses its own embedding (denoted by "distinct

Table 2.6: Performance of GENI on FB15K when a single embedding is used for all predicates (shared embedding) vs. when each predicate uses its own embedding (distinct embedding).

Metric	Shared Embedding	Distinct Embedding
NDCG@100	0.9062 ± 0.008	$\mathbf{0.9385 \pm 0.004}$
SPEARMAN	0.6894 ± 0.007	$\mathbf{0.7772 \pm 0.006}$

Table 2.7: Performance of PR, log in-degree baseline, and GENI with fixed and flexible centrality adjustment (CA) on FB15K and TMDB5K.

Method	FB15K		TMDB5K	
	NDCG@100	SPEARMAN	NDCG@100	SPEARMAN
PR	0.835 ± 0.02	0.352 ± 0.02	0.829 ± 0.03	0.590 ± 0.01
Log In-Degree	0.810 ± 0.02	0.300 ± 0.03	0.852 ± 0.02	0.685 ± 0.02
GENI-Fixed CA	0.868 ± 0.01	0.613 ± 0.01	0.899 ± 0.01	0.771 ± 0.01
GENI-Flexible CA	$\mathbf{0.938 \pm 0.00}$	$\mathbf{0.777 \pm 0.01}$	$\mathbf{0.905 \pm 0.01}$	$\mathbf{0.780 \pm 0.01}$

embedding"). Note that using "shared embedding", GENI loses the ability to distinguish between different predicates. In the results given in Table 2.6, we observe that NDCG@100 and SPEARMAN are increased by 3.6% and 12.7%, respectively, when a dedicated embedding is used for each predicate. This shows that GENI successfully makes use of predicates for modeling the relation between node importance; this is especially crucial in KGs such as FB15K that consist of a large number of predicates.

2.4.5.2 Flexibility for Centrality Adjustment.

In Equation (2.7), we perform scaling and shifting of $c(i)$ for flexible centrality adjustment (CA). Here we evaluate the model with fixed CA without scaling and shifting where the final estimation $s^*(i) = \sigma_s(c(i) \cdot s^L(i))$. In Table 2.7, we report the performance of GENI on FB15K and TMDB5K obtained with fixed and flexible CA while all other parameters were identical. When node centrality strongly correlates with input scores, fixed CA obtains similar results to flexible CA. This is reflected on the result of TMDB5K dataset, where PR and log in-degree baseline (LID), which estimates node importance as the log of its in-degree, both estimate node importance close to the input scores. On the other hand, when node centrality is not in good agreement with input scores, as demonstrated by the poor performance of PR and LID as on FB15K, flexible CA performs much better than fixed CA (8% higher NDCG@100, and 27% higher SPEARMAN on FB15K).

2.4.5.3 Parameter Sensitivity

We evaluate the parameter sensitivity of GENI by measuring performance on FB15K varying one of the following parameters while fixing others to their default values (shown in parentheses): number of score aggregation (SA) layers (1), number of SA

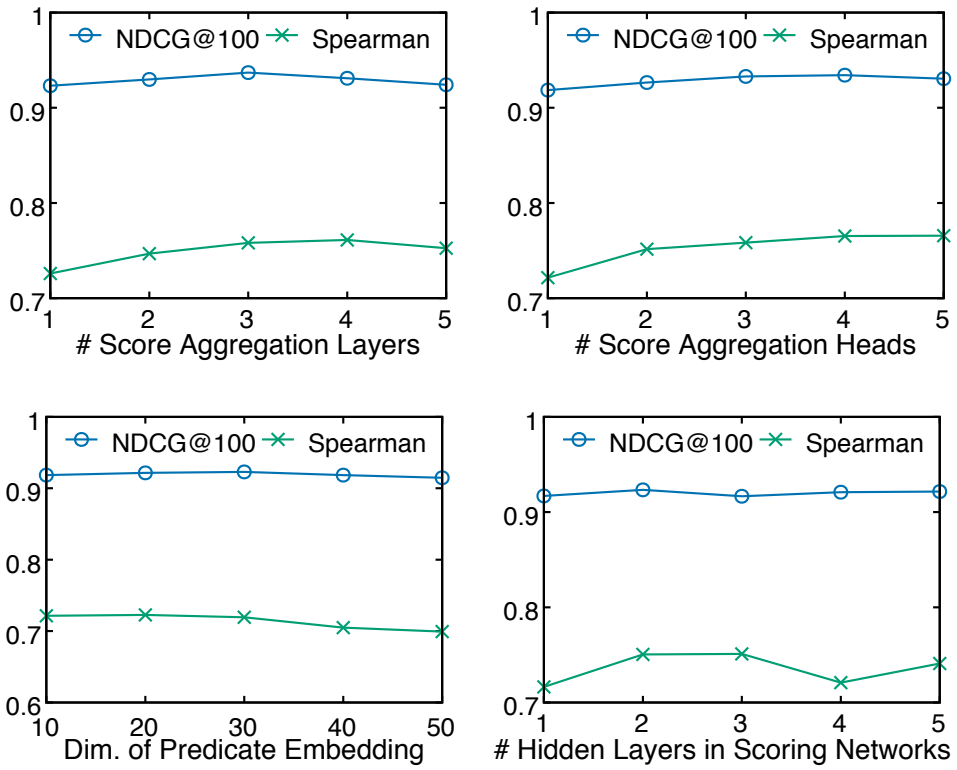


Figure 2.4: Parameter sensitivity of GENI on FB15K. We report results varying one parameter on x-axis, while fixing all others.

heads in each SA layer (1), dimension of predicate embedding (10), and number of hidden layers in scoring networks (1 layer with 48 units). Results presented in Figure 2.4 shows that the model performance tends to improve as we use a greater number of SA layers and SA heads. For example, SPEARMAN increases from 0.72 to 0.77 as the number of SA heads is increased from 1 to 5. Using more hidden layers for scoring networks also tends to boost performance, although exceptions are observed. Increasing the dimension of predicate embedding beyond an appropriate value negatively affects the model performance, although GENI still achieves high SPEARMAN compared to baselines.

2.5 Related Work

In this section, we review previous works on node importance estimation and graph neural networks.

Node Importance Estimation. Many approaches have been developed for node importance estimation [PBMW99, Hav02, TFP08, JPSK17, LNY12, Kle99]. PageRank (PR) [PBMW99] is based on the random surfer model where an imaginary surfer randomly moves to a neighboring node with probability d, or teleports to any other node randomly with probability $1 - d$. PR predicts the node importance to be the limiting probability of

the random surfer being at each node. Accordingly, PR scores are determined only by the graph structure, and unaware of input importance scores. Personalized PageRank (PPR) [Hav02] deals with this limitation by biasing the random walk to teleport to a set of nodes relevant to some specific topic, or alternatively, nodes with known importance scores. Random walk with restart (RWR) [TFP08, JPSK17] is a closely related method that addresses a special case of PPR where teleporting is restricted to a single node. PPR and RWR, however, are not well suited for KGs since they do not consider edge types. To make a better use of rich information in KGs, HAR [LNY12] extends the idea of random walk used by PR and PPR to solve limiting probabilities arising from multi-relational data, and distinguishes between different predicates in KGs while being aware of importance scores. Previous methods can be categorized as non-trainable approaches with a fixed model structure that do not involve model parameter optimization. In this chapter, we explore supervised machine learning algorithms with a focus on graph neural networks.

Graph Neural Networks (GNNs). GNNs are a class of neural networks that learn from arbitrarily structured graph data. Many GNN formulations have been based on the notion of graph convolutions. The pioneering work of Bruna et al. [BZSL14] defined the convolution operator in the Fourier domain, which involved performing the eigendecomposition of the graph Laplacian; as a result, its filters were not spatially localized, and computationally costly. A number of works followed to address these limitations. Henaff et al. [HBL15] introduced a localization of spectral filters via the spline parameterization. Defferrard et al. [DBV16] designed more efficient, strictly localized convolutional filters. Kipf and Welling [KW17] further simplified localized spectral convolutions via a first-order approximation. To reduce the computational footprint and improve performance, recent works explored different ways of neighborhood aggregation. One direction has been to restrict neighborhoods via sampling techniques such as uniform neighbor sampling [HYL17], vertex importance sampling [CMX18], and random walk-based neighbor importance sampling [YHC+18]. Graph attention networks (GAT) [VCC+18], which is most closely related to our method, explores an orthogonal direction of assigning different importance to different neighbors by employing self-attention over neighbors [VSP+17]. While GAT exhibited state-of-the-art results, it was applied only to node classifications, and is unaware of predicates. Building upon recent developments in GNNs, GENI tackles the challenges for node importance estimation in KGs, which have not been addressed by existing GNNs.

2.6 Conclusion

Estimating node importance in KGs is an important problem with many applications such as item recommendation and resource allocation. In this chapter, we present a method GENI that addresses this problem by utilizing rich information available in KGs in a flexible manner which is required to model complex relation between entities and their importance. Our main ideas can be summarized as score aggregation via predicate-aware attention mechanism and flexible centrality adjustment. Experimental results on predicting node importance in real-world KGs show that GENI outperforms

existing approaches, achieving 5–17% higher NDCG@100 than the state of the art.

2.7 Appendix

In the appendix, we provide details on datasets, experimental settings, and additional experimental results, such as a case study on TMDB5K and regression performance evaluation for in-domain predictions.

Table 2.8: Top-10 movies and directors with highest predicted importance scores by GENI, HAR, and GAT on TMDB5K. "*ground truth rank*"−"*estimated rank*" is shown for each prediction.

(a) Top-10 movies (in-domain estimation). A *ground truth rank* is computed from known importance scores of movies used for testing.

	GENI		**HAR**		**GAT**	
1	The Dark Knight Rises	11	Jason Bourne	63	The Dark Knight Rises	11
2	The Lego Movie	70	The Wolf of Wall Street	21	Clash of the Titans	103
3	Spectre	10	Rock of Ages	278	Ant-Man	4
4	Les Misérables	94	Les Misérables	94	The Lego Movie	68
5	The Amazing Spider-Man	22	The Dark Knight Rises	7	Jack the Giant Slayer	126
6	Toy Story 2	39	V for Vendetta	27	Spectre	7
7	V for Vendetta	26	Now You See Me 2	81	The Wolf of Wall Street	16
8	Clash of the Titans	97	Spectre	5	The 5th Wave	67
9	Ant-Man	-2	Austin Powers in Goldmember	140	The Hunger Games: Mockingjay - Part 2	-4
10	Iron Man 2	29	Alexander	141	X-Men: First Class	767

(b) Top-10 directors (out-of-domain estimation). A *ground truth rank* corresponds to the rank in a director ranking (N/A indicates that the director is not in the director ranking).

	GENI		**HAR**		**GAT**	
1	Steven Spielberg	0	Steven Spielberg	0	Noam Murro	N/A
2	Tim Burton	9	Martin Scorsese	44	J Blakeson	N/A
3	Ridley Scott	6	Ridley Scott	6	Pitof	N/A
4	Martin Scorsese	42	Clint Eastwood	19	Paul Tibbitt	N/A
5	Francis Ford Coppola	158	Woody Allen	112	Rupert Sanders	N/A
6	Peter Jackson	-4	Robert Zemeckis	1	Alan Taylor	145
7	Robert Rodriguez	127	Tim Burton	4	Peter Landesman	N/A
8	Gore Verbinski	8	David Fincher	40	Hideo Nakata	N/A
9	Joel Schumacher	63	Oliver Stone	105	Drew Goddard	N/A
10	Robert Zemeckis	-3	Ron Howard	-2	Tim Miller	N/A

2.7.1 Datasets

We perform evaluation using four real-world KGs that have different characteristics. All KGs were constructed from public data sources, which we specify in the footnote. Summaries of these datasets (such as the number of nodes, edges, and predicates) are given in Table 2.3. Below, we provide details on the construction of each KG.

FB15K. We used a sample of Freebase[5] used by [BUG+13]. The original dataset is divided into training, validation, and test sets. We combined them into a single dataset, and later divided them randomly into three sets based on our proportion for training, validation, and test data. In order to find the number of pageviews of a Wikipedia page, which is the importance score used for FB15K, we used Freebase/Wikidata mapping[6]. Most entities in FB15K can be mapped to the corresponding Wikidata page, from which we found the link to the item's English Wikipedia page, which provides several information including the number of pageviews in the past 30 days.

MUSIC10K. We built MUSIC10K from the sample[7] of the Million Song Dataset[8]. This dataset is a collection of audio features and metadata for one million popular songs. Among others, this dataset includes information about songs such as the primary artist and the album the song belongs to. We constructed MUSIC10K by adding nodes for these three entities (i.e., songs, artists, and albums), and edges of corresponding types between them as appropriate. Note that MUSIC10K is much more fragmented than other datasets.

TMDB5K. We constructed TMDB5K from the TMDb 5000 movie dataset[9]. This dataset contains movies and relevant information such as movie genres, companies, countries, crews, and casts in a tabular form. We added nodes for each of these entities, and added edges between two related entities with appropriate types. For instance, given that "Steven Spielberg" directed "Schindler's List", we added two corresponding director and movie nodes, and added an edge of type "directed" between them.

IMDB. We created IMDB from public IMDb datasets[10]. IMDb datasets consist of several tables, which contain information such as titles, genres, directors, writers, principal casts and crews. As for TMDB5K, we added nodes for these entities, and connected them with edges of corresponding types. In creating IMDB, we focused on entities related to movies, and excluded other entities that have no relation with movies. In addition, IMDb datasets include titles each person is known for; we added edges between a person and these titles to represent this special relationship.

Scores. For FB15K, TMDB5K, IMDB, we added 1 to the importance scores as an offset, and log-transformed them as the scores were highly skewed. For MUSIC10K, two types of

[5] https://everest.hds.utc.fr/doku.php?id=en:smemlj12
[6] https://developers.google.com/freebase/
[7] https://think.cs.vt.edu/corgis/csv/music/music.html
[8] https://labrosa.ee.columbia.edu/millionsong/
[9] https://www.kaggle.com/tmdb/tmdb-movie-metadata
[10] https://www.imdb.com/interfaces/

provided scores were all between 0 and 1, and we used them without log transformation.

2.7.2 Experimental Settings

2.7.2.1 Cross Validation and Early Stopping

We performed 5-fold cross validation; i.e., for each fold, 80% of the ground truth scores were used for training, and the other 20% were used for testing. For methods based on neural networks, we applied early stopping by using 15% of the original training data for validation and the remaining 85% for training, with a patience of 50. That is, the training was stopped if the validation loss did not decrease for 50 consecutive epochs, and the model with the best validation performance was used for testing.

2.7.2.2 Software

We used several open source libraries, and used Python 3.6 for our implementation.

Graph Library. We used NetworkX 2.1 for graphs and graph algorithms: *MultiDiGraph* class was used for all KGs as there can be multiple edges of different types between two entities; NetworkX's *pagerank_scipy* function was used for PR and PPR.

Machine Learning Library. We chose TensorFlow 1.12 as our deep learning framework. We used scikit-learn 0.20.0 for other machine learning algorithms such as random forest and linear regression.

Other Libraries and Algorithms. For GAT, we used the reference TensorFlow implementation provided by the authors[11]. We implemented HAR in Python 3.6 based on the algorithm description presented in [LNY12]. For node2vec, we used the implementation available from the project page[12]. NumPy 1.15 and SciPy 1.1.0 were used for data manipulation.

2.7.2.3 Hyperparameters and Configurations

PageRank (PR) and Personalized PageRank (PPR) We used the default values for NetworkX's *pagerank_scipy* function with 0.85 as a damping factor.

HAR [LNY12]. As in PPR, normalized input scores were used as probabilities for entities; equal probability was assigned to all relations. We set $\alpha = 0.15$, $\beta = 0.15$, $\gamma = 0$. The maximum number of iterations was set to 30. Note that HAR is designed to compute two types of importance scores, hub and authority. For MUSIC10K, TMDB5K, and IMDB KGs, these scores are identical since each edge in these graphs has a matching edge with an inverse predicate going in the opposite direction. Thus for these KGs, we only report authority scores. For FB15K, we compute both types of scores, and report authority scores as hub scores are slightly worse overall.

[11] https://github.com/PetarV-/GAT
[12] https://snap.stanford.edu/node2vec/

Linear Regression (LR) and Random Forests (RF). For both methods, we used default parameter values defined by scikit-learn.

Neural Networks (NN). Let $[n_1, n_2, n_3, n_4]$ denote a 3-layer neural network where n_1, n_2, n_3 and n_4 are the number of neurons in the input, first hidden, second hidden, and output layers, respectively. For NN, we used an architecture of $[N_F, 0.5 \times N_F, 0.25 \times N_F, 1]$ where N_F is the dimension of node features. We applied a rectified linear unit (ReLU) non-linearity at each layer, and used Adam optimizer with a learning rate $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and a weight decay of 0.0005.

Graph Attention Networks (GAT) [VCC+18]. We used a GAT model with two attentional layers, each of which consists of four attention heads, which is followed by a fully connected NN (FCNN). Following the settings in [VCC+18], we used a Leaky ReLU with a negative slope of 0.2 for attention coefficient computation, and applied an exponential linear unit (ELU) non-linearity to the output of each attention head. The output dimension of an attention head in all layers except the last was set to $\max(0.25 \times N_F, 20)$. For FCNN after the attentional layers, we used an architecture of $[0.75 \times N_F, 1]$ with ReLU as non-linearity. Adam optimizer was applied with a learning rate $\alpha = 0.005$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and a weight decay of 0.0005.

GENI. We used an architecture where each score aggregation (SA) layer contains four SA heads. For FB15K, we used a model with three SA layers, and for other KGs, we used a model with one SA layer. For SCORINGNETWORK, a two-layer FCNN with an architecture of $[N_F, 0.75 \times N_F, 1]$ was used. GENI was trained with Adam optimizer using a learning rate $\alpha = 0.005$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and a weight decay of 0.0005. The dimension of predicate embedding was set to 10 for all KGs. We used a Leaky ReLU with a negative slope of 0.2 for attention coefficient computation (σ_a), and a RELU for the final score estimation (σ_s). We defined $\mathcal{N}(i)$ as outgoing neighbors of node i. Similar results were observed when we defined $\mathcal{N}(i)$ to include both outgoing and incoming neighbors of node i. Since the initial values for γ and β (parameters for centrality adjustment) affect model performance, we determined these initial values for each dataset based on the validation performance.

node2vec [GL16]. We set the number of output dimensions to 64 for FB15K, MUSIC10K, and TMDB5K, and 128 for IMDB. Other parameters were left to their default values. Note that node2vec was used in our experiments to generate node features for supervised methods.

2.7.3 Additional Evaluation

2.7.3.1 Case Study

We take a look at the predictions made by GENI, HAR, and GAT on TMDB5K. Given popularity scores for some movies, methods estimate the importance score of all other entities in TMDB5K. Among them, Table 2.8 reports the top-10 movies and directors that are estimated to have the highest importance scores by each method with "ground truth rank"−"estimated rank" shown for each entity.

Table 2.9: RMSE (root-mean-squared error) of in-domain prediction obtained by supervised methods. Lower RMSE is better. GENI consistently outperforms all baselines. Numbers after $\pm$ symbol are standard deviation from 5-fold cross validation. Best results are in bold, and second best results are underlined.

Method	FB15K	MUSIC10K	TMDB5K	IMDB
LR	1.3536 ± 0.017	0.1599 ± 0.002	0.8431 ± 0.028	1.7534 ± 0.005
RF	1.2999 ± 0.024	$\underline{0.1494 \pm 0.002}$	0.9223 ± 0.015	1.8181 ± 0.011
NN	1.2463 ± 0.015	0.1622 ± 0.009	0.8496 ± 0.012	2.0279 ± 0.033
GAT	$\underline{1.0798 \pm 0.031}$	0.1635 ± 0.007	$\underline{0.8020 \pm 0.010}$	$\underline{1.2972 \pm 0.018}$
GENI	$\mathbf{0.9471 \pm 0.017}$	$\mathbf{0.1491 \pm 0.002}$	$\mathbf{0.7150 \pm 0.003}$	$\mathbf{1.2079 \pm 0.011}$

In-domain estimation is presented in Table 2.8a. A ground truth rank is computed from the known importance scores of movies reserved for testing. The top-10 movies predicted by GENI is qualitatively better than the two others. For example, among the ten predictions of GAT and HAR, the difference between ground truth rank and predicted rank is greater than 100 for three movies. On the other hand, the rank difference for GENI is less than 100 for all predictions.

Out-of-domain estimation is presented in Table 2.8b. As importance scores for directors are unknown, we use the director ranking introduced in Section 2.4.1. A ground truth rank denotes the rank in the director ranking, and "N/A" indicates that the director is not included in the director ranking. The quality of the top-10 directors estimated by GENI and HAR is similar to each other with five directors appearing in both rankings (e.g., Steven Spielberg). Although GAT is not considerably worse than GENI for in-domain estimation, its out-of-domain estimation is significantly worse than others: nine out of ten predictions are not even included in the list of top-200 highest earning directors. By respecting node centrality, GENI yields a much better ranking consistent with ground truth.

2.7.3.2 Regression Performance Evaluation for In-Domain Predictions

In order to see how accurately supervised approaches recover the importance of nodes, we measure the regression performance of their in-domain predictions. In particular, we report RMSE (root-mean-squared error) of supervised methods in Table 2.9. Non-trainable methods are excluded since their output is not in the same scale as the input scores. GENI performs better than other supervised methods on all four real-world datasets. Overall, the regression performance of supervised approaches follows a similar trend to their performance in terms of ranking measures reported in Table 2.4.

Chapter based on work published in KDD 2020 [PKD+20].

Chapter 3

Inferring Node Importance in a Knowledge Graph from Multiple Input Signals

Given multiple input signals, how can we infer node importance in a knowledge graph (KG)? Node importance estimation is a crucial and challenging task that can benefit a lot of applications including recommendation, search, and query disambiguation. A key challenge towards this goal is how to effectively use input from different sources. On the one hand, a KG is a rich source of information, with multiple types of nodes and edges. On the other hand, there are external input signals, such as the number of votes or pageviews, which can directly tell us about the importance of entities in a KG. While several methods have been developed to tackle this problem, their use of these external signals has been limited as they are not designed to consider multiple signals simultaneously. In this chapter, we develop an end-to-end model MULTIIMPORT, which infers latent node importance from multiple, potentially overlapping, input signals. MULTIIMPORT is a latent variable model that captures the relation between node importance and input signals, and effectively learns from multiple signals with potential conflicts. Also, MULTIIMPORT provides an effective estimator based on attentive graph neural networks. We ran experiments on real-world KGs to show that MULTIIMPORT handles several challenges involved with inferring node importance from multiple input signals, and consistently outperforms existing methods, achieving up to 23.7% higher NDCG@100 than the state-of-the-art method.

3.1 Introduction

Real-world networks consist of several types of entities, interacting with each other via multiple types of relations. These complex and rich interactions between entities from diverse domains are abstracted by a *knowledge graph* (KG), which is a multi-relational graph where nodes are real-world entities or concepts, and edges denote the corresponding relation (also called *predicate*) between nodes. Given a KG, estimating node importance is a crucial task that has been studied extensively [PBMW99, Hav02, Kle99, TFP08, LNY12, JPSK17, PKD+19], as it enables a large number of applications such as recommendation, search, and ranking, to name a few.

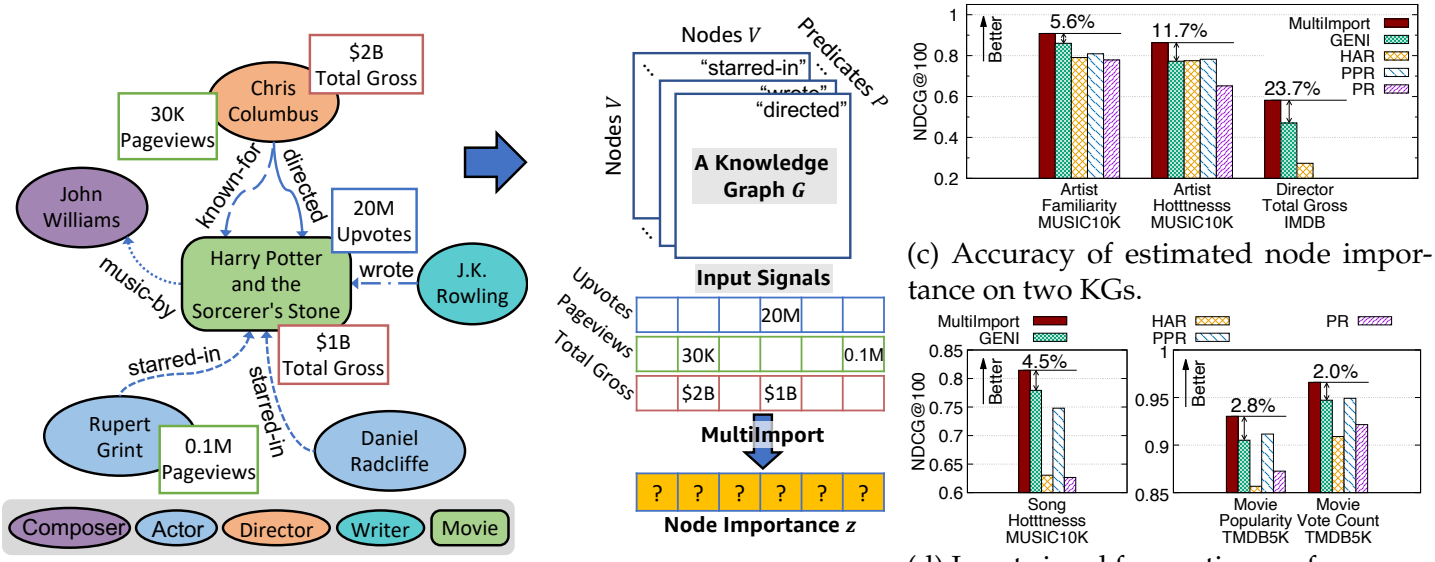
A key challenge to achieve this goal lies in effectively using input from different sources. On the one hand, KGs represent how entities are related to each other. In particular, compared to the conventional graphs that make no distinction between edges, KGs provide abundant information as they comprise heterogeneous entities and predicates. For instance, consider the cross-domain KG on the movie "Harry Potter and the Sorcerer's Stone" and related entities (Figure 3.1a), which consists of entities from various domains (e.g., "actor", "composer", and "movie") and multiple relations (e.g., "directed" and "wrote").

On the other hand, we can often obtain relevant data on the entities in a KG from external sources such as the World Wide Web. Among them, some are direct indicators of the importance or popularity of an entity as they capture how much time, money, or attention people have spent on it. The number of votes and pageviews are examples of such signals. We call this data that captures node importance *input signal*. A number of input signals are usually available for a KG, such as the total gross of movies in a movie KG, although they are often sparse and some are applicable to only specific types of entities, as illustrated in Figure 3.1a.

Existing approaches for measuring node importance include random walk-based methods, such as PageRank (PR) [PBMW99], Personalized PageRank (PPR) [Hav02], and HAR [LNY12], and more recent supervised techniques, such as GENI [PKD+19], which learn to estimate node importance. In the context of measuring entity importance in a multi-relational graph, these methods can be compared in terms of what type of input

Table 3.1: Comparison of MULTIIMPORT and baselines in terms of what type of input each method considers to measure node importance in a knowledge graph. Only MULTIIMPORT can consider multiple input signals.

Input	**MULTI IMPORT**	**GENI** [PKD+19]	**HAR** [LNY12]	**PPR** [Hav02]	**PR** [PBMW99]
Graph Structure	✓	✓	✓	✓	✓
Multiple Predicates	✓	✓	✓		
Single Input Signal	✓	✓	✓	✓	
Multiple Input Signals	✓				



(a) A movie knowledge graph.

(b) Problem setup.

(c) Accuracy of estimated node importance on two KGs.

(d) Input signal forecasting performance on two KGs.

Figure 3.1: (a) A knowledge graph (KG) on a movie and related entities. Node color denotes an entity type and an edge type denotes the type of relation between entities. Rectangles represent input signals (e.g., total gross). Note that a single entity can have a variable number of input signals. (b) Given a KG and input signals, MULTIIMPORT infers the importance of all nodes. (c) MULTIIMPORT infers up to 23.7% more accurate node importance than the state-of-the-art method (GENI). (d) MULTIIMPORT achieves up to 4.5% higher forecasting results than baselines, with an NDCG@100 of 0.98 on TMDB5K. See Section 3.5 for details.

they can use, as Table 3.1 summarizes. While PR can consider only the graph structure, the development of more advanced random walk-based techniques enabled considering additional input. State-of-the-art results on this task have been achieved by GENI, which is built upon a supervised framework optimized to use both the KG and an external input signal.

However, all existing approaches can only consider up to one input signal, even though several signals are typically available from diverse sources. Also, while it is left to the users to decide which signal to use, no guideline has been provided. Importantly, by ignoring all other signals except for one, they lose information that can complement each other and provide more reliable and accurate evidence for node importance when used together.

In this chapter, we present MULTIIMPORT, a supervised approach that makes an effective use of multiple input signals towards learning accurate and trustworthy node importance in a KG. Note that among different types of input listed in Table 3.1, input signals are the most direct and strongest indicator of node popularity. However, utilizing multiple signals raises several challenges that require careful design choices. First, given sparse, potentially overlapping, multiple input signals, it is not clear how unknown node importance can be inferred. Also, using all available signals may lead to worse results,

when there exist conflicts among signals. Developing an effective graph-based estimator is another challenge to model the relation between node importance, input signals, and the KG.

To address these challenges, we model the task using a latent variable model and derive an effective learning objective. By adopting an iterative clustering-based training scheme, we handle those signals that may deteriorate the estimation quality. Also, we use predicate-aware, attentive graph neural networks (GNNs) to model the interactions among input signals and the KG. Our contributions are summarized as follows:

- **Problem Formulation.** We formulate the problem of inferring node importance in a KG from multiple input signals.
- **Algorithm.** We present MULTIIMPORT, a novel supervised method that effectively learns from multiple input signals by handling the aforementioned challenges.
- **Effectiveness.** We show the superiority of MULTIIMPORT using experiments on real-world KGs . Figures 3.1c and 3.1d show that MULTIIMPORT outperforms existing methods across multiple signals and KGs, achieving up to 23.7% higher NDCG@100 than the state of the art.

3.2 Background

Graph Neural Networks (GNNs) [GSR+17, YHC+18, VCC+18] are deep learning architectures for graph-structured data. GNNs consist of multiple layers, where each one updates the embeddings of each node by aggregating the embeddings from the neighborhood, and combining it with the current embeddings. How the ℓ-th layer in GNNs computes the embeddings $\mathbf{h}_i^\ell$ of node i can be summarized as follows:

$$\mathbf{h}_i^\ell \leftarrow \textsc{Combine}^\ell\left(\mathbf{h}_i^{\ell-1},\ \textsc{Aggregate}^\ell\left(\left\{\mathbf{h}_j^{\ell-1} \mid j \in \mathcal{N}(i)\right\}\right)\right)$$

where $\mathcal{N}(i)$ denotes the neighbors of node i, $\textsc{Aggregate}^\ell$ is an operator that aggregates (e.g., averaging) the embeddings of neighbors, potentially after applying some form of transformation to them, and $\textsc{Combine}^\ell$ is an operator that merges the aggregated embeddings with the embeddings $\mathbf{h}_i^{\ell-1}$ of node i. Different GNNs may use different definitions of $\mathcal{N}(i)$, $\textsc{Aggregate}^\ell(\cdot)$, and $\textsc{Combine}^\ell(\cdot)$.

3.3 Task Description

In this section, we present key concepts and the task description.

Knowledge Graph. A *knowledge graph* (KG) is a heterogeneous network with multiple types of entities and relations. As shown in Figure 3.1b, a KG can be represented by a third-order tensor $\boldsymbol{\mathcal{G}} \in \mathbb{R}^{|V|\times|P|\times|V|}$, in which a non-zero at (s, ρ, o) indicates that a subject $s \in V$ is related to an object $o \in V$ via a predicate $\rho \in P$ where V and P are the sets of indices for entities and predicates, respectively. Real-world KGs, such as Freebase [BEP+08] and DBpedia [LIJ+15], usually contain a large number of predicates. Also, two entities can be related via multiple predicates as in Figure 3.1a.

Node Feature. Node-specific information is often available, and can be encoded in a vector of fixed length F. Examples include document embedding for the entity description, and more domain-specific features like motif gene sets from the Molecular Signatures Database [STM+05]. We use $\boldsymbol{X} \in \mathbb{R}^{|V| \times F}$ to denote all node features.

Node Importance. A *node importance* $z \in \mathbb{R}_{\geq 0}$ is a non-negative real number that represents the importance of an entity in a KG, with a higher value denoting a higher node importance. Node importance is a latent quantity, and thus not directly observable.

Input Signal. An *input signal* $S : V' \to \mathbb{R}_{\geq 0}$ $(V' \subseteq V)$ is a partial map between a node and a non-negative real number that represents the significance or popularity of the node. For entities in a KG, there are often several external data that could serve as an input signal. Examples of those signals include the number of copies sold (e.g., of books), the total gross of movies and directors, and the number of votes, reviews, and pageviews given for products. Note that they may highlight node popularity from different perspectives: e.g., number of clicks in the last one month (higher for trending movies) vs. total number of clicks so far (higher for classic movies). Also, some signals may be available only for some type of nodes (e.g., the number of tickets sold for movies).

In this work, we consider a set of M input signals $\{S_i : V_i' \to \mathbb{R}_{\geq 0} \mid V_i' \subseteq V, i = 1, \ldots, M\}$ where the signal domain V_i' might or might not overlap with each other. We note the following facts.

- Input signals can be in different scales. For example, while signal A ranges from 1 to 5, signal B could range from 0 to 100.
- Input signals often have high correlation as important nodes tend to have high values across different signals. However, signals may have a varying degree of correlation when signals capture different aspects of node importance or some involve more noise.

Task Description. Based on these concepts, our task of estimating node importance in a KG is summarized as follows (Figure 3.1b presents a pictorial overview):

Definition 3. Node Importance Estimation: Given a KG $\boldsymbol{G} \in \mathbb{R}^{|V| \times |P| \times |V|}$, node features $\boldsymbol{X} \in \mathbb{R}^{|V| \times F}$, and a set of M input signals $\{S_i : V_i' \to \mathbb{R}_{\geq 0} \mid V_i' \subseteq V, i = 1, \ldots, M\}$, where V and P denote the sets of indices of entities and predicates in $\boldsymbol{G}$, respectively, estimate the latent importance $z \in \mathbb{R}_{\geq 0}$ of every node in V.

3.4 Methods

Inferring node importance in a KG from multiple input signals requires addressing three major challenges.

1. **Formulating learning objective.** Given a KG and potentially overlapping signals, how can we infer latent node importance?
2. **Handling rebel input signals.** Given input signals that may possess different characteristics or involve more noise, how can we deal with potential conflicts and infer the node importance?

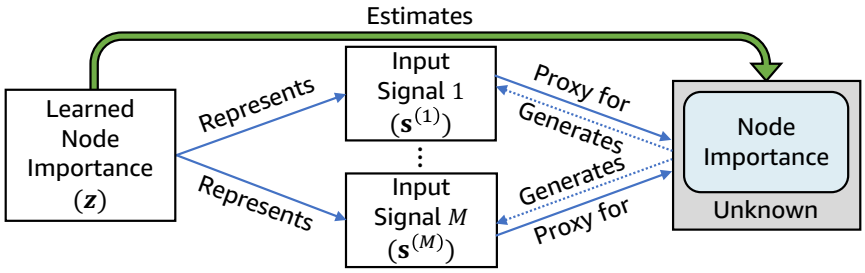


Figure 3.2: MULTIIMPORT estimates the latent node importance by learning to represent input signals, which are proxies for the unknown node importance.

3. **Effective graph-based estimation.** How can we effectively model the relations between node importance, input signals, and the KG?

In this section, we present MULTIIMPORT that addresses the above challenges with the following ideas.

1. **Modeling the task using a latent variable model** enables capturing relations between node importance and input signals, and provides an optimization framework (Section 3.4.1).
2. **Iterative clustering-based training** handles rebel input signals, effectively inferring the node importance (Section 3.4.2).
3. **Predicate-aware, attentive GNNs** provide a powerful node importance estimator to infer graph-regularized node importance (Section 3.4.3).

The definition of symbols used in this chapter is provided in Table 3.2.

3.4.1 Learning Objective

Given our task to infer node importance, one may consider using input signals directly as node features in supervised methods, as they provide useful cues on the significance of a node. However, since signals are partially observed, we will first need to fill in the missing values to use them as a node feature. Further, even if input signals are available for all nodes, with all signals treated as node features, it is not obvious how to infer node importance from them, as node importance is a latent value.

Given that node importance is unknown and cannot be directly observed, we assume that there is an underlying variable governing node importance, and observed input signals are generated by this variable with noise and possibly via non-linear transformations. Accordingly, we consider an input signal to be a partial indicator of latent node importance, and at the same time, to be a reasonably good proxy for the unknown node importance. Based on these assumptions, we approach our goal of estimating unknown node importance by learning to represent input signals. Figure 3.2 illustrates our assumptions on the relationship among the learned node importance, input signals, and node importance.

Notations. To formally define the learning objective, we introduce a few symbols. Let $\mathcal{G}$

collectively denote the third-order tensor $\boldsymbol{G}$ representing the KG and the node features $\boldsymbol{X}$. We denote the number of dimensions of vector $\mathbf{v}$ by $\dim(\mathbf{v})$. Given M observed input signals, let $\boldsymbol{s}^{(i)}$ denote a vector corresponding to the i-th signal, and $\boldsymbol{S}$ denote a set of M signal vectors, i.e., $\boldsymbol{S} = \{\boldsymbol{s}^{(1)}, \boldsymbol{s}^{(2)}, \ldots, \boldsymbol{s}^{(M)}\}$. Let $\boldsymbol{z} \in \mathbb{R}^{|V|}$ denote a vector of estimated importance for all nodes. We use $\boldsymbol{z}^{(i)}$ to refer to the vector of estimated importance of those nodes for which signal i is available. Thus, $\dim(\boldsymbol{z}^{(i)})$ equals $\dim(\boldsymbol{s}^{(i)})$. Since signals are partially observable, we have that $\dim(\boldsymbol{s}^{(i)}) = \dim(\boldsymbol{z}^{(i)}) \leq |V| = \dim(\boldsymbol{z})$. To denote the j-th value of a signal or estimated importance, we use a subscript, e.g., $s_j^{(i)}$ and $z_j^{(i)}$.

Maximum a Posteriori Learning. Our goal can be summarized as learning an estimator $f(\cdot)$ that produces estimated importance $\boldsymbol{z}$ for all nodes in the KG. Specifically, as we consider a graph-based estimator with learnable parameters, the estimation by $f(\cdot)$ is determined by the given KG $\boldsymbol{\mathcal{G}}$ and its learnable parameters $\boldsymbol{\theta}$. In other words, we have:

$$\boldsymbol{z} = f(\boldsymbol{\mathcal{G}}, \boldsymbol{\theta}). \tag{3.1}$$

In order to optimize $f(\cdot)$, we aim to maximize the posterior probability of model parameters $\boldsymbol{\theta}$ given that we have observed the KG $\boldsymbol{\mathcal{G}}$ and input signals $\boldsymbol{S}$:

$$\max p(\boldsymbol{\theta}|\boldsymbol{\mathcal{G}}, \boldsymbol{S}). \tag{3.2}$$

By the Bayes' theorem, this is equivalent to maximizing:

$$p(\boldsymbol{\theta}|\boldsymbol{\mathcal{G}}, \boldsymbol{S}) = \frac{p(\boldsymbol{\mathcal{G}}, \boldsymbol{S}|\boldsymbol{\theta})\, p(\boldsymbol{\theta})}{p(\boldsymbol{\mathcal{G}}, \boldsymbol{S})} \propto p(\boldsymbol{\theta})\, p(\boldsymbol{\mathcal{G}}|\boldsymbol{\theta})\, p(\boldsymbol{S}|\boldsymbol{\mathcal{G}}, \boldsymbol{\theta}). \tag{3.3}$$

The first term $p(\boldsymbol{\theta})$ in Equation (3.3) represents the prior probability of the model parameters $\boldsymbol{\theta}$, which we assume to be a Gaussian distribution with zero mean and an isotropic covariance:

$$p(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\mathbf{0}, \lambda^{-1}\mathbf{I}). \tag{3.4}$$

The second term $p(\boldsymbol{\mathcal{G}}|\boldsymbol{\theta})$ in Equation (3.3) is the likelihood of the observed KG $\boldsymbol{\mathcal{G}}$ given $\boldsymbol{\theta}$. As we later discuss in Section 3.4.3, given features $\mathbf{x}_i$ of node i, MULTIIMPORT embeds node i in an intermediate low-dimensional space by projecting $\mathbf{x}_i$ using a learnable function $g(\cdot)$. Let (s, ρ, o) denote a subject-predicate-object triple in the KG. Using factorization-based KG embeddings [WMWG17], we model the observed triple (s, ρ, o) as a diagonal bilinear interaction between node embeddings $g(\mathbf{x}_s)$, $g(\mathbf{x}_o)$ and the learnable predicate embedding $\mathbf{w}_\rho$ with normally distributed errors. Specifically,

$$p(\boldsymbol{\mathcal{G}}|\boldsymbol{\theta}) = \prod_{(s,\rho,o)\in\boldsymbol{\mathcal{G}}} p((s, \rho, o)|\boldsymbol{\theta}) \tag{3.5}$$

$$= \prod_{(s,\rho,o)\in\boldsymbol{\mathcal{G}}} \mathcal{N}\left(g(\mathbf{x}_s)^\intercal \operatorname{diag}(\mathbf{w}_\rho)\, g(\mathbf{x}_o) \middle| 1, \nu^{-1}\right) \tag{3.6}$$

Table 3.2: Table of symbols.

Symbol	Definition
$\boldsymbol{\mathcal{G}}$	knowledge graph with node features
V, P	set of indices of nodes and predicates in a KG
$\mathbf{x}_i, \boldsymbol{X}$	feature vector of node i, and a matrix of all node features
$\boldsymbol{\theta}$	learnable parameters of the node importance estimator $f(\cdot)$
M	number of input signals
$\boldsymbol{s}^{(i)}$	a vector of i-th observed input signal
$\boldsymbol{S}$	a set of M input signals, i.e., $\boldsymbol{S} = \{\boldsymbol{s}^{(1)}, \boldsymbol{s}^{(2)}, \ldots, \boldsymbol{s}^{(M)}\}$
$\boldsymbol{z}$	a vector of estimated importance of all nodes
$\boldsymbol{z}^{(i)}$	a vector of estimated importance of those nodes with signal i
$\dim(\mathbf{z})$	number of dimensions of vector $\mathbf{z}$
$\mathrm{diag}(\mathbf{w})$	a diagonal matrix whose diagonal entries are given by vector $\mathbf{w}$
$\mathcal{N}(i)$	neighboring edges of node i
h_i^ℓ	importance of node i estimated by the ℓ-th layer in the GNN
h_i^*	final estimated importance of node i by the GNN (i.e., $z_i = h_i^*$)
ω_{ij}^ℓ	node i's attention on the m-th edge from node j in layer ℓ
$\pi(\rho_{ij}^m)$	learnable predicate embedding of m-th edge between nodes i and j

where $\mathrm{diag}(\mathbf{w}_\rho)$ denotes a diagonal matrix where the diagonal entries are given by $\mathbf{w}_\rho$. This term can also be seen as an assumption on the homophily between neighboring nodes in the space represented by node embedding $g(\cdot)$ and predicate ρ.

The third term $p(\boldsymbol{S}|\boldsymbol{\mathcal{G}}, \boldsymbol{\theta})$ of Equation (3.3) is the likelihood of observed input signals $\boldsymbol{S}$ given the KG $\boldsymbol{\mathcal{G}}$ and model parameters $\boldsymbol{\theta}$. Given M signals, we assume that they are conditionally independent. Accordingly, we have that:

$$p(\boldsymbol{S}|\boldsymbol{\mathcal{G}}, \boldsymbol{\theta}) = p(\boldsymbol{s}^{(1)}|\boldsymbol{\mathcal{G}}, \boldsymbol{\theta}) \cdot \ldots \cdot p(\boldsymbol{s}^{(M)}|\boldsymbol{\mathcal{G}}, \boldsymbol{\theta}). \tag{3.7}$$

Recall that $\boldsymbol{z}$ is a function of $\boldsymbol{\mathcal{G}}$ and $\boldsymbol{\theta}$, or in other words, $\boldsymbol{\mathcal{G}}$ and $\boldsymbol{\theta}$ fully determine $\boldsymbol{z}$, and input signals can be partially observed (i.e., $\dim(\boldsymbol{s}^{(i)})$ may not equal $\dim(\boldsymbol{z})$). Equation (3.7) can be expressed in terms of input signals and the corresponding estimated importance:

$$p(\boldsymbol{S}|\boldsymbol{\mathcal{G}}, \boldsymbol{\theta}) \propto p(\boldsymbol{s}^{(1)}|\boldsymbol{z}^{(1)}) \cdot \ldots \cdot p(\boldsymbol{s}^{(M)}|\boldsymbol{z}^{(M)}) \tag{3.8}$$

in which the log-likelihood $\log p(\boldsymbol{s}^{(i)}|\boldsymbol{z}^{(i)})$ of observing signal $\boldsymbol{s}^{(i)}$ given $\boldsymbol{z}^{(i)}$ is proportional to:

$$\log p(\boldsymbol{s}^{(i)}|\boldsymbol{z}^{(i)}) \propto \log \prod_{j=1}^{\dim(\boldsymbol{s}^{(i)})} p(z_j^{(i)})^{p(s_j^{(i)})} = \sum_{j=1}^{\dim(\boldsymbol{s}^{(i)})} p(s_j^{(i)}) \log p(z_j^{(i)}). \tag{3.9}$$

Note that taking a negative of Equation (3.9) leads to the cross entropy.

With respect to the probability $p(\boldsymbol{s}^{(i)})$ of observing signal $\boldsymbol{s}^{(i)}$, we consider two things. First, input signals can be in different scales. Since signals are obtained from diverse sources, their values could be in different scales and units, and thus may not be directly comparable (e.g., # clicks vs. dwell time vs. the total revenue in dollars). Second, for most downstream applications of node importance, the rank of each entity's importance matters much more than the raw value itself. In light of these observations, we consider the probability of observing a signal vector in terms of ranking.

To do so, once we obtain a list of entity rankings from the signal vector, we need a probability model that measures the likelihood of the ranked list. Permutation probability [CQL+07] is one such model, in which the likelihood of a ranked list is defined with respect to a given permutation of the list, such that the permutation which corresponds to sorting entities according to their ranking is most likely to be observed. In our setting, however, this model is not feasible since there are $O(|V|!)$ permutations to be considered. Instead, we use a tractable approximation of it called top one probability. Given signal $\boldsymbol{s}^{(i)}$, the top one probability $p(s_j^{(i)})$ of j-th entity in $\boldsymbol{s}^{(i)}$ represents the probability of that entity to be ranked at the top of the list given the signal values of other entities, and is defined as:

$$p(s_j^{(i)}) = \frac{\phi(s_j^{(i)})}{\sum_{k=1}^{\dim(\boldsymbol{s}^{(i)})} \phi(s_k^{(i)})} = \frac{\exp(s_j^{(i)})}{\sum_{k=1}^{\dim(\boldsymbol{s}^{(i)})} \exp(s_k^{(i)})}. \tag{3.10}$$

Here, $\phi(\cdot)$ is a strictly increasing positive function, which we define to be an exponential function. Similarly, given model estimation $\boldsymbol{z}^{(i)}$, the top one probability $p(z_j^{(i)})$ of j-th entity in $\boldsymbol{z}^{(i)}$ is computed as:

$$p(z_j^{(i)}) = \frac{\exp(z_j^{(i)})}{\sum_{k=1}^{\dim(\boldsymbol{z}^{(i)})} \exp(z_k^{(i)})}. \tag{3.11}$$

Taking a negative logarithm of our posterior in Equation (3.3) and plugging in Equations (3.4), (3.6), (3.8) and (3.9), we get the following loss:

$$\begin{aligned} \mathcal{L} &= -\log\left(p(\boldsymbol{\theta})\, p(\boldsymbol{\mathcal{G}}|\boldsymbol{\theta})\, p(\boldsymbol{S}|\boldsymbol{\mathcal{G}}, \boldsymbol{\theta})\right) \\ &= \left(-\sum_{i=1}^{M} \sum_{j=1}^{\dim(\boldsymbol{s}^{(i)})} p(s_j^{(i)}) \log p(z_j^{(i)})\right) \\ &\quad + \frac{\nu}{2}\left(\sum_{(s,\rho,o)\in\boldsymbol{\mathcal{G}}} \left(g(\mathbf{x}_s)^\intercal \operatorname{diag}(\mathbf{w}_\rho)\, g(\mathbf{x}_o) - 1\right)^2\right) + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2 \end{aligned} \tag{3.12}$$

where $p(s_j^{(i)})$ and $p(z_j^{(i)})$ are given by Equations (3.10) and (3.11).

3.4.2 Handling Rebel Input Signals

In Section 3.4.1, MULTIIMPORT infers node importance from all M input signals. This is based on the assumption that the given signals have been generated by a common hidden variable as depicted in Figure 3.2, that is, input signals are homogeneous and a

high correlation exists among them. However, some signals (which we call *rebel* signals) may exhibit a low correlation with the others when they possess different characteristics from others, or involve more noise. As a result, when given multiple signals where some are weakly correlated with others, learning from all of them leads to a worse estimator due to the violation of our modeling assumption.

To effectively infer node importance from multiple signals while handling rebel signals, we adopt an iterative clustering-based training, where closely related input signals are put into the same cluster and our estimator is trained using not all the given signals, but only those in the same cluster. To do so, we need to be able to measure the relatedness of input signals. Again, considering that signals can be in different scales and ranks are important for downstream applications, we compare a pair of signals in terms of the Spearman correlation coefficient, a well-known rank correlation measure.

However, comparing input signals is not always possible, since they can be disjoint as signals are partially observable. To handle this, MULTIIMPORT (a) initially assigns M signals to their own cluster, (b) separately infers node importance from each one, (c) do a pairwise comparison between observed and inferred values, and (d) merges clusters by applying existing clustering algorithms, such as DBSCAN [EKSX96], on the pairwise signal similarity. With the resulting clusters, we repeat the same process again until there is no change in the clustering. This is because learning from an enlarged cluster can lead to a higher modeling accuracy, as we show in Section 3.5.4. If there is enough overlap between signals' observed values, we may omit step (b), and compute their similarity directly from observed values in step (c). These steps are illustrated in Figure 3.3, and Algorithm 3.1 gives our learning algorithm.

Given multiple signals, our focus is to infer a single number for each node, which represents the major aspect of node importance, as supported by the signals and relevant to downstream applications. This requires sorting the final clusters based on their priority. Examples of such priority policy π include: (1) cluster size (prefer a cluster with a larger number of signals); (2) cluster quality (prefer a cluster with a higher reconstruction accuracy); (3) signal preference (prefer a cluster with signals that are important for the given application). In experiments, we use cluster size as our priority.

Incremental Learning. Our approach naturally lends itself to incremental learning settings where new signals are added after the model training. As in the initial training phase, new signals are first put into their own cluster, and MULTIIMPORT merges them with existing clusters based on the similarity of inferred importance.

3.4.3 Graph Neural Networks for Node Importance Estimation

Given this optimization framework, we now present a supervised estimator $f(\cdot)$ that can model complex relations among input signals and the information of the KG. In MULTIIMPORT, we utilize GNNs, which have been shown to be a powerful model for learning on a graph. We adopt and improve upon the recent development of GNNs, such as the dynamic neighborhood aggregation via an attention mechanism, by making

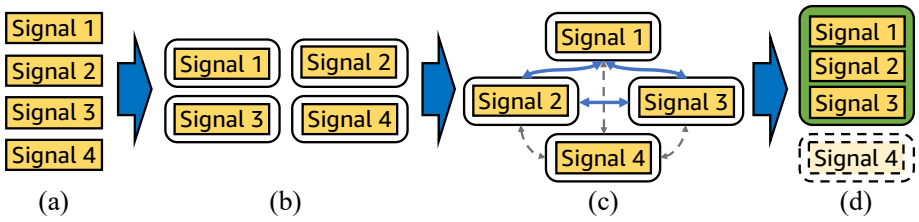


Figure 3.3: MULTIIMPORT identifies similar signals (those in the green cluster), and infers node importance from them. See text for details of steps (a) to (d).

Algorithm 3.1: Learning algorithm

Input: knowledge graph $\mathcal{G}$, input signals $\boldsymbol{S}$, merge threshold t, priority policy π.
Output: estimated node importance.

repeat
  Assign input signals without cluster membership (e.g., newly added signals) to their own cluster, if any
  foreach *cluster* c **do**
    Infer node importance by training an estimator $f_c(\cdot)$ with the loss function in Equation (3.12)
  Merge those clusters whose similarity is greater than threshold t
until *there is no change in the clustering*;
return node importance inferred from the cluster that has the highest priority according to the policy π

extensions and simplifications.

MULTIIMPORT first projects features $\mathbf{x}_i$ of node i to a low-dimensional space using a learnable function $g(\cdot)$. Let $\mathbf{x}'_i = g(\mathbf{x}_i)$. As discussed in Section 3.4.1, MULTIIMPORT allows assuming homophily among neighboring nodes in this embedding space. Given these intermediate node embeddings, our estimator further transforms them into one dimensional embedding to directly represent nodes by their importance. To do so, MULTIIMPORT uses another learnable function $g'(\cdot)$. In other words, MULTIIMPORT represents node i as $h_i^0 \in \mathbb{R}$ in the space of node importance such that $h_i^0 = g'(\mathbf{x}'_i)$. Note that both $g(\cdot)$ and $g'(\cdot)$ are learnable functions, and can be a simple linear transformation or multi-layer neural networks.

Then, given h_i^0 for all i, we apply attentive GNNs to it on the given KG to obtain graph-regularized node importance such that the estimated importance smoothly changes with respect to the KG in a predicate-aware manner. MULTIIMPORT is a multi-layer GNN with L layers. The ℓ-th layer performs a weighted aggregation of the node importance estimated by the $(\ell-1)$-th layer from the neighborhood $\mathcal{N}(i)$ to produce a new estimation h_i^ℓ for node i:

$$h_i^\ell = \sum_{(j,m)\in\mathcal{N}(i)} \omega_{(i,j,m)}^\ell \, h_j^{\ell-1} \tag{3.13}$$

Although attentive GNNs usually compute the attention weight for neighboring nodes assuming simple graphs, KGs are directed graphs with parallel edges. Therefore, instead of node-level attention, we compute edge-level attention weights, which enables making a distinction among edges between two nodes. We define $\mathcal{N}(i)$ to be a set of neighboring edges of node i such that $(j, m) \in \mathcal{N}(i)$ if there exists an m-th edge between nodes i and j (under some edge ordering). The weight $\omega^{\ell}_{(i,j,m)}$ of the edge $(j, m) \in \mathcal{N}(i)$ is then computed using a predicate-aware attention parameterized by a weight vector $\mathbf{a}_\ell$:

$$\omega^{\ell}_{(i,j,m)} = \frac{\exp\left(\text{LeakyReLU}\left(\mathbf{a}_\ell^\intercal \left[h_i^{\ell-1} || \pi(\rho_{ij}^m) || h_j^{\ell-1}\right]\right)\right)}{\sum_{(k,n)\in\mathcal{N}(i)} \exp\left(\text{LeakyReLU}\left(\mathbf{a}_\ell^\intercal \left[h_i^{\ell-1} || \pi(\rho_{ik}^n) || h_k^{\ell-1}\right]\right)\right)} \tag{3.14}$$

where ρ_{ij}^m denotes the predicate type of m-th edge between nodes i and j, $\pi(\cdot)$ is a learnable function that maps a predicate type to its embedding (i.e., $\pi(\rho_{ij}^m)$ is the embedding of the predicate of the m-th edge between nodes i and j), and $||$ is a concatenation operator. Motivated by GENI, we generate the final estimation h_i^* for node i by making a centrality-based adjustment to the estimation h_i^L made by the final layer L, where in-degree d_i of node i is used to define its centrality c_i:

$$\begin{aligned} c_i &= \alpha \cdot \log(d_i + \epsilon) + \beta \\ h_i^* &= \text{ReLU}\left(c_i \cdot h_i^L\right) \end{aligned} \tag{3.15}$$

where α and β are learnable parameters and ϵ is a small positive value. In summary, the final estimated node importance $\mathbf{z}$ is produced as follows:

$$\boldsymbol{z} = f(\boldsymbol{\mathcal{G}}, \boldsymbol{\theta}) = [h_1^*, \ldots, h_{|V|}^*]^\intercal. \tag{3.16}$$

3.5 Experiments

In this section, we address the following questions.

Q1. Accuracy. How consistent is estimated node importance with input signals? In particular, how does the estimation performance change when multiple input signals are considered?

Q2. Use in downstream tasks. How useful is the estimated importance for downstream tasks?

Q3. Handling rebel signals. How does a rebel signal affect the performance, and how well does our method handle it?

After describing datasets, evaluation plans, and baselines in Sections 3.5.1 to 3.5.3, we address the above questions in Sections 3.5.4 to 3.5.6. Experimental settings are presented in Section 3.8.

3.5.1 Dataset Description

We used four publicly available real-world KGs that have different characteristics, and were used in a previous study on node importance estimation [PKD+19]. We constructed

Table 3.3: Real-world KGs used in our evaluation. These KGs vary in different aspects, such as the size and number of predicates. SCC: Strongly connected component.

Name	# Nodes	# Edges	# Predicates	# SCCs
FB15K	14,951	592,213	1,345	9
MUSIC10K	24,830	71,846	10	130
TMDB5K	123,906	532,058	22	15
IMDB	1,567,045	14,067,776	28	1

these datasets following the description in [PKD+19]. Below we give a brief description of these KGs. Statistics of these data are given in Table 3.3, and the list of available input signals in each KG is provided in Table 3.4.

FB15K [BUG+13] is a KG sampled from the Freebase knowledge base [BEP+08], which consists of general knowledge harvested from many sources, and compiled by collaborative efforts. FB15K is much denser and contains a much larger number of predicates than other KGs.

MUSIC10K is a music KG representing the relation between songs, artists, and albums. MUSIC10K is constructed from a subset of the Million Song Dataset[1], and it provides three input signals called "song hotttnesss", "artist hotttnesss", and "artist familiarity", which are popularity scores computed by considering several relevant data such as playback count.

TMDB5K is a movie KG representing relations among movie-related entities such as movies, actors, directors, crews, casts, and companies. TMDB5K is constructed from the TMDb 5000 datasets[2], and contains several signals including the "popularity" score computed by considering relevant statistics like the number of votes[3].

IMDB is a movie KG constructed from the daily snapshot of the IMDb dataset[4] on movies and related entities, e.g., genres, directors, casts, and crews. IMDB is the largest KG among the four KGs. As IMDb dataset provides only one input signal (# votes), we collected popularity signal from TMDb for 5% of the movies in IMDB.

3.5.2 Performance Evaluation

For evaluation, we use normalized discounted cumulative gain (NDCG), which is a widely used metric for relevance ranking problems. Given a list of nodes for which we have the estimated importance and the ground truth scores, we sort the list by the estimated importance, and consider the ground truth signal value at position i (denoted

[1]http://millionsongdataset.com/
[2]https://www.kaggle.com/tmdb/tmdb-movie-metadata
[3]https://developers.themoviedb.org/3/getting-started/popularity
[4]https://www.imdb.com/interfaces/

Table 3.4: Input signals in real-world KGs. The percentage of nodes covered by each signal is given in the parentheses.

Name	Type	Input Signals
FB15K	Generic	# Pageviews, # total edits, and # page watchers on Wikipedia (all 94%)
MUSIC10K	Artist Song	Artist hotttnesss (14%) and artist familiarity (16%) Song hotttnesss (17%)
TMDB5K	Movie Director	Popularity, revenue, budget, and vote count (all 4%) Box office grosses for top 200 directors
IMDB	Movie Director	# Votes (14%) and popularity (from TMDb, 5%) Box office grosses for top 200 directors

Table 3.5: MULTIIMPORT estimates node importance more accurately than baselines, and using additional signals improves the accuracy. MULTIIMPORT-1 is the same as MULTIIMPORT except that it used only one signal denoted with an asterisk (*). Methods that can use only one input signal also used the one marked with an asterisk (*). The best result is in bold and in dark gray. The second best result is underlined and in light gray. TR: Training. ID: In-Domain. OOD: Out-Of-Domain.

Method	FB15K			MUSIC10K		
	# Page Watchers* (Generic, TR, ID)	# Total Edits (Generic, TR, ID)	# Pageviews (Generic, ID)	Familiarity* (Artist, TR, ID)	Hotttnesss (Artist, TR, ID)	Hotttnesss (Song, OOD)
PR	0.7747 ± 0.02	0.8579 ± 0.00	0.8441 ± 0.00	0.7788 ± 0.01	0.6520 ± 0.00	0.4846 ± 0.00
PPR	0.7810 ± 0.02	0.8604 ± 0.00	0.8450 ± 0.00	0.8090 ± 0.01	0.7823 ± 0.01	0.6422 ± 0.02
HAR	0.7625 ± 0.01	0.9080 ± 0.00	0.8732 ± 0.00	0.7905 ± 0.01	0.7751 ± 0.01	0.6377 ± 0.01
GENI	0.8548 ± 0.02	0.8787 ± 0.04	0.8464 ± 0.03	0.8603 ± 0.01	0.7727 ± 0.02	0.6804 ± 0.01
MULTIIMPORT-1	$\underline{0.8879 \pm 0.02}$	$\underline{0.9250 \pm 0.03}$	$\underline{0.8863 \pm 0.03}$	$\underline{0.8839 \pm 0.01}$	$\underline{0.8046 \pm 0.02}$	$\underline{0.7109 \pm 0.02}$
MULTIIMPORT	$\mathbf{0.9150 \pm 0.01}$	$\mathbf{0.9498 \pm 0.01}$	$\mathbf{0.9066 \pm 0.01}$	$\mathbf{0.9083 \pm 0.00}$	$\mathbf{0.8633 \pm 0.02}$	$\mathbf{0.7173 \pm 0.01}$

Method	TMDB5K				IMDB		
	Popularity* (Movie, TR, ID)	Vote Count (Movie, TR, ID)	Revenue (Movie, ID)	Total Gross (Director, OOD)	# Votes* (Movie, TR, ID)	Popularity (Movie, TR, ID)	Total Gross (Director, OOD)
PR	0.8294 ± 0.02	0.8482 ± 0.00	0.9009 ± 0.00	0.8829 ± 0.00	0.7927 ± 0.02	0.7019 ± 0.00	0.0000 ± 0.00
PPR	0.8585 ± 0.01	0.9133 ± 0.01	0.9535 ± 0.00	0.8691 ± 0.02	0.7927 ± 0.02	0.7169 ± 0.01	0.0000 ± 0.00
HAR	0.8131 ± 0.02	0.9350 ± 0.00	0.9537 ± 0.00	0.9298 ± 0.01	0.7976 ± 0.02	$\underline{0.7671 \pm 0.00}$	0.2735 ± 0.03
GENI	0.9055 ± 0.02	0.9367 ± 0.00	0.9646 ± 0.00	0.9398 ± 0.00	0.9367 ± 0.00	0.7079 ± 0.01	0.4703 ± 0.02
MULTIIMPORT-1	$\underline{0.9075 \pm 0.02}$	$\mathbf{0.9555 \pm 0.00}$	$\underline{0.9650 \pm 0.00}$	$\underline{0.9497 \pm 0.00}$	$\underline{0.9493 \pm 0.00}$	0.7581 ± 0.01	$\underline{0.5607 \pm 0.01}$
MULTIIMPORT	$\mathbf{0.9302 \pm 0.01}$	$\underline{0.9536 \pm 0.00}$	$\mathbf{0.9716 \pm 0.00}$	$\mathbf{0.9558 \pm 0.00}$	$\mathbf{0.9542 \pm 0.01}$	$\mathbf{0.8444 \pm 0.02}$	$\mathbf{0.5819 \pm 0.03}$

by r_i) to compute the discounted cumulative gain at position k ($DCG@k$) as follows:

$$DCG@k = \sum_{i=1}^{k} \frac{r_i}{\log_2(i+1)}.$$

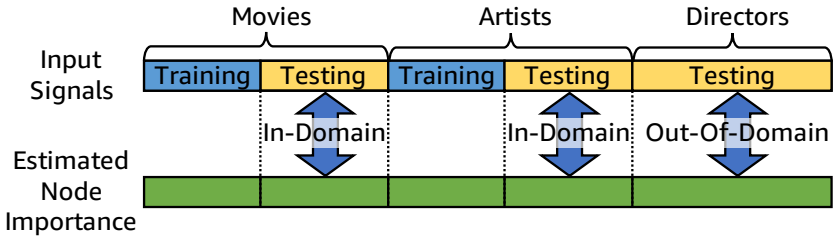


Figure 3.4: In- and out-of-domain evaluation where three input signals are given and two are used for training.

The gain is accumulated from the top to the bottom of the list, and gets reduced at lower positions due to the logarithmic reduction factor. An ideal DCG at rank position k ($IDCG@k$) can be obtained by sorting nodes by their ground truth values, and computing $DCG@k$ for this ordered list. Normalized DCG at position k ($NDCG@k$) ranges from 0 to 1, and is defined as:

$$NDCG@k = \frac{DCG@k}{IDCG@k}$$

with higher values indicating a better ranking quality.

In computing NDCG, we consider all entities if the input signal is generic and applies to all entity types as in FB15K. Otherwise, we consider only those entities of the type for which the ground truth signal applies. For instance, to compute NDCG with respect to movie popularity, we only consider movies. In general, we compute NDCG over those entities that have ground truth values. An exception is the director signal, which is limited to the top 200 directors based on their worldwide box office grosses. When computing NDCG for the director signal, we consider all director entities, and assume the ground truth of the directors to be 0 if they are outside the top 200 list. In our experiments, we report NDCG@100; using different values for the threshold k yielded similar results.

Generalization. In order to evaluate the generalization ability of each method, we perform 5-fold cross validation where 80% of the input signals are used for training, while the remaining 20% are used for testing. Similar results were observed with different number of folds. Also, since input signals often apply to a specific type of nodes (e.g., directors), we also consider how well a method generalizes to the nodes of unseen types. Consider Figure 3.4 for an example. Given input signals on movies, artists, and directors, movie and artist signals are used for both training and testing, while the director signal is used for testing alone. Here we call an evaluation on movies and artists *in-domain* as training involved input signals on these types of nodes, and an evaluation on directors *out-of-domain* since training used no input signal on this type. It is desirable to achieve a higher accuracy in both criteria.

3.5.3 Baselines

We use the following baselines: PageRank (PR) [PBMW99], Personalized PageRank (PPR) [Hav02], HAR [LNY12], and GENI [PKD+19]. PR, PPR, and HAR are representative random walk-based algorithms for measuring node importance. GENI is a supervised method that achieved the state-of-the-art result. We omitted results from other supervised algorithms, including linear regression, random forests, dense neural networks, and other GNNs, such as GAT [VCC+18] and GCN [KW17], as GENI has been shown to outperform them in our experiments.

3.5.4 Q1. Accuracy

The quality of estimated node importance $\boldsymbol{z}$ can be measured by how well it correlates with the observed signals. That is, accurately estimated node importance $\boldsymbol{z}$ should strongly correlate with input signals. We measure the degree of correlation between the estimation $\boldsymbol{z}$ and input signals using NDCG. In Table 3.5, we report NDCG@100 of estimated node importance with respect to each signal in our four KGs.

In the table, only those signals marked with TR were used for training. For baselines that can accept at most one input signal (PPR, HAR, and GENI), we used the signal marked with an asterisk (*) as the training signal. MULTIIMPORT-1 is identical to MULTIIMPORT except that only one signal (marked with *) is used for training.

Overall, MULTIIMPORT consistently outperformed baselines across all signals on four datasets, in terms of both in-domain (ID) and out-of-domain (OOD) evaluation. Note that MULTIIMPORT inferred node importance by learning from the given multiple signals. A comparison between MULTIIMPORT and MULTIIMPORT-1 shows that learning from multiple input signals led to a performance improvement of up to 11%. Even if the training signals were given for the same type of entities (e.g., artists or movies), considering multiple signals also improved the performance on OOD entities (e.g., songs or directors). While GENI performed better than random walk-based methods in most cases, it was outperformed by MULTIIMPORT due to its inability to consider multiple signals.

3.5.5 Q2. Use in Downstream Tasks

Estimated importance $\boldsymbol{z}$ can be viewed as a summary of KG nodes in terms of input signals. This summary can be used as a feature in downstream applications. In this section, we evaluate how useful MULTIIMPORT's estimation $\boldsymbol{z}$ is in downstream signal prediction and forecasting tasks, as opposed to the estimation learned by baselines.

Signal Prediction. The signal prediction task is to predict some input signal $\boldsymbol{s}^{(i)}$ using a machine learning model (M) that uses other input signals $\boldsymbol{s}^{(1)}, \ldots, \boldsymbol{s}^{(i-1)}$ and the estimated node importance $\boldsymbol{z}$ as input features. Here $\boldsymbol{z}$ can be generated by different methods. Each method uses only $\boldsymbol{s}^{(1)}, \ldots, \boldsymbol{s}^{(i-1)}$, and no other input signals (i.e., $\boldsymbol{s}^{(i)}$ is not available during generation of $\boldsymbol{z}$). Once $\boldsymbol{z}$ is obtained, we compare the performance of M when the input features consist of only $\boldsymbol{s}^{(1)}, \ldots, \boldsymbol{s}^{(i-1)}$ vs. when they are composed

Table 3.6: MULTIIMPORT achieves the best signal prediction results (marked in bold and in dark gray), where we use input signals and estimated node importance as input features to predict another input signal. In only one exception, MULTIIMPORT achieves the second best result (underlined and in light gray). z_m denotes node importance estimated by method m.

(a) Predicting "budget" on TMDB5K by using popularity (p), vote count (v) and estimated node importance (z) as features.

Input Features	NDCG@10	NDCG@100
p, v	0.8296 ± 0.09	0.8176 ± 0.00
p, v, z_{PR}	0.8700 ± 0.06	0.8260 ± 0.00
p, v, z_{PPR}	0.8602 ± 0.07	0.8312 ± 0.01
p, v, z_{HAR}	0.8341 ± 0.08	0.8062 ± 0.02
p, v, z_{GENI}	$\underline{0.8791 \pm 0.06}$	$\underline{0.8740 \pm 0.01}$
$p, v, z_{\text{MULTIIMPORT}}$	$\mathbf{0.8930 \pm 0.07}$	$\mathbf{0.8757 \pm 0.00}$

(b) Predicting "artist hotttnesss" on MUSIC10K by using artist familiarity (f) and estimated node importance (z) as features.

Input Features	NDCG@10	NDCG@100
f	0.3727 ± 0.12	0.5751 ± 0.05
f, z_{PR}	$\underline{0.4092 \pm 0.03}$	$\underline{0.5839 \pm 0.01}$
f, z_{PPR}	0.3685 ± 0.12	0.5727 ± 0.05
f, z_{HAR}	0.3727 ± 0.12	0.5751 ± 0.05
f, z_{GENI}	0.3484 ± 0.15	0.5650 ± 0.06
$f, z_{\text{MULTIIMPORT}}$	$\mathbf{0.4186 \pm 0.06}$	$\mathbf{0.5886 \pm 0.00}$

(c) Predicting "# page watchers" on FB15K by using # num pageviews (p), # num total edits (e) and estimated node importance (z) as features.

Input Features	NDCG@10	NDCG@100
p, e	0.8681 ± 0.01	0.8859 ± 0.02
p, e, z_{PR}	0.9010 ± 0.01	0.8975 ± 0.02
p, e, z_{PPR}	0.9010 ± 0.01	0.8975 ± 0.02
p, e, z_{HAR}	$\underline{0.9052 \pm 0.00}$	0.8945 ± 0.02
p, e, z_{GENI}	0.8884 ± 0.00	$\mathbf{0.9076 \pm 0.02}$
$p, e, z_{\text{MULTIIMPORT}}$	$\mathbf{0.9084 \pm 0.00}$	$\underline{0.9062 \pm 0.02}$

of both $s^{(1)}, \ldots, s^{(i-1)}$ and z. Also, we compare the prediction performance of M as we use z obtained with different methods. The motivation for this signal prediction test is that estimated importance can be considered as de-noising compression of input signals. Thus a high-quality estimate of importance should be a useful input feature for downstream signal prediction model. We used a linear regression model as our M and optimized it using the loss shown in Equation (3.12) (without the second term), using input signals from TMDB5K, MUSIC10K, and FB15K as input features. Table 3.6 shows the performance in terms of NDCG. MULTIIMPORT achieved the best results across all datasets, except for one case where it achieved the second best result, still obtaining up to 12% better result compared to when z was not used as input features. This shows that the estimation obtained by MULTIIMPORT captures useful information contained in the input signals.

Forecasting. The forecasting task is concerned with predicting scores for newly added

Table 3.7: MULTIIMPORT outperforms all baselines in forecasting signals. TR denotes the signals used for training MULTIIMPORT; baselines were trained with the one marked with an asterisk (*). The bottom table shows how data was split.

Method	TMDB5K			MUSIC10K
	Popularity* (Movie, TR) NDCG@100	Vote Count (Movie, TR) NDCG@100	Budget (Movie) NDCG@100	Hotttnesss* (Song, TR) NDCG@100
PR	0.8726	0.9215	0.9294	0.6266
PPR	0.9116	0.9492	0.9640	0.7480
HAR	0.8567	0.9090	0.9265	0.6306
GENI	0.9051	0.9472	0.9647	0.7792
MULTIIMPORT	**0.9303**	**0.9660**	**0.9829**	**0.8145**

Dataset	Training (# Entities)	Testing (# Entities)
TMDB5K	Movies released until 2013 (4243)	Movies released from 2014 (559)
MUSIC10K	Songs released until 2005 (1961)	Songs released from 2006 (751)

KG entities. Consider movie KGs such as TMDB5K or IMDB as an example. Given input signals for movies released until some time point t, the task is to accurately estimate the input signals of those movies released after time t. In this setting, high forecasting accuracy would be an indication of the usefulness of MULTIIMPORT. Table 3.7 shows the forecasting performance of MULTIIMPORT and baselines on TMDB5K and MUSIC10K, and how movies and songs were split into training and testing sets. We set the split point such that it is close to the end of the range of release dates, while the testing set contains enough number of entities. For this test, we excluded those movies and songs without the release date. On TMDB5K, baselines were trained using the signal marked with an asterisk (*), while MULTIIMPORT used both signals denoted with TR. Across all signals, MULTIIMPORT consistently outperformed baselines, achieving up to 4.5% higher NDCG@100.

3.5.6 Q3. Handling Rebel Signals

To see the effect of handling rebel signals, we report in Figure 3.5 how the modeling accuracy changes on MUSIC10K (left) and TMDB5K (right) as rebel signals are handled or not. For each KG, we trained MULTIIMPORT using the two signals shown on the x- and y-axis, first time handling rebel signals and second time ignoring to handle rebel signals, and report the NDCG@100 for both signals. For MUSIC10K, we used PR scores, which turn out to be a weak estimator of node importance in this KG, and randomly generated values as rebel signals; for TMDB5K, we included vote average as a rebel signal. In both KGs, by identifying and dropping rebel signals, MULTIIMPORT can achieve up to 14.7% higher NDCG@100 in modeling the input signals in comparison to when failing to handling rebel signals.

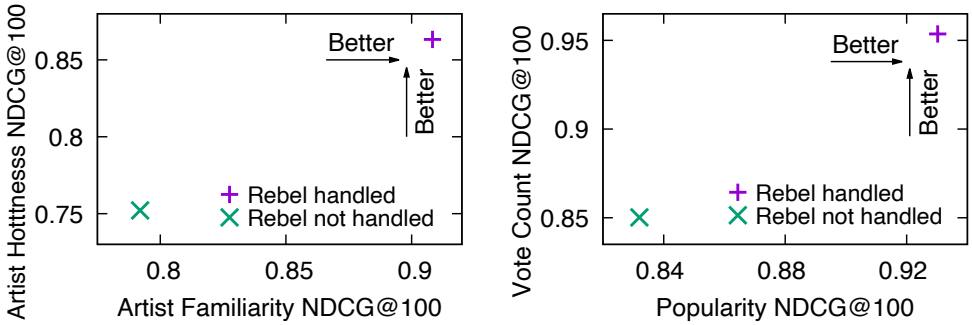


Figure 3.5: Rebel signals can hurt. MULTIIMPORT effectively handles them, achieving a higher accuracy.

3.6 Related Work

In this section, we review existing works on node importance estimation (NIE), data fusion and reconstruction, and graph neural networks (GNNs). Some of the discussion of NIE and GNNs are repeated from Section 2.5 for the reader's convenience.

Estimating Node Importance is a major graph mining problem, and importance scores have been used in many real world applications, including search and recommendation. PageRank (PR) [PBMW99] is a random walk model that propagates the importance of each node by either traversing the graph structure or teleporting to a random node with a fixed probability. PR outputs universal node importance for all nodes but these scores do not directly capture proximity between nodes. Personalized PageRank (PPR) [Hav02] and Random Walk with Restart (RWR) [TFP08] were then proposed to address this limitation by introducing a random jump biased to a particular set of target nodes. However, all these methods were designed for homogeneous graphs and thus do not take into account different types of edges in a KG. To leverage edge type information for node importance estimation, HAR [LNY12] employs a signal propagation schema that is sensitive to edge types. Earlier in this thesis, we develop GENI (Chapter 2), a graph neural network-based model for estimating node importance. By using graph attention mechanism, GENI adaptively fuses information from different edge types, and achieved the state-of-the-art results for node importance estimation. Despite its success, GENI can use only one signal, while multiple signals may come from different sources. In this chapter, we propose a new method MULTIIMPORT that can harmonize signals from multiple sources.

Data Fusion and Reconstruction are related to estimating node importance from input signals. Data fusion approaches [YHY08, DBS09, DBHS10] integrate potentially conflicting information about entities from multiple data sources (e.g., websites) by considering the trustworthiness of data sources and the dependence between them. Among several values of an object, these methods aim to identify the true value of the object. Our problem setup is related to, but different from data fusion in that our goal is to learn latent node importance which is broadly consistent with input signals, while focusing on a subset of signals which correlate well with each other, instead of identifying which

signal value is more accurate than others.

Data reconstruction methods aim to complete missing values in the partially observed data. Matrix and tensor decomposition [KB09, PJLK16, POK17, POK19, OPJ+19] are representative approaches to this task, which reconstruct observed data using a relatively small number of latent factors. While a rank-1 decomposition of input signals is analogous to our problem, MULTIIMPORT simultaneously considers various data sources, such as the KG and input signals, while handling rebel signals and employing GNNs for effective inference over a KG.

Graph Neural Networks apply deep learning ideas to arbitrary graph structured data. These methods have attracted extensive research interest in recent years [KW17, GSR+17, YHC+18, XLT+18, VCC+18]. Kipf and Welling [KW17] proposed a spectral approach, called GCN, which employs a localized first-order approximation of graph convolutions. More recently, Veličković et al. [VCC+18] proposed graph attention networks (GATs) to aggregate localized neighbor information based on attention mechanism. GAT provides an efficient framework to integrate deep learning into graph mining, and has been adopted to recommender systems [WZG+19], knowledge graph reasoning [ZPW+19], and graph classification [LRK18]. While our work is also based on a graph attention architecture, we additionally consider predicates in attention computation, and extend this architecture in a multi-task setting for learning node importance in a KG using multiple signals.

3.7 Conclusion

Estimating node importance in a KG is a crucial task that has received a lot of interest. A major challenge in successfully achieving this goal is in utilizing multiple types of input effectively. In particular, input signals provide strong evidence for the popularity of entities in a KG. In this chapter, we develop an end-to-end framework MULTIIMPORT that draws on information from both the KG and external signals, while dealing with challenges arising from the simultaneous use of multiple input signals, such as inferring node importance from sparse signals, and potential conflicts among them. We ran experiments on real-world KGs to show that MULTIIMPORT successfully handles these challenges, and consistently outperforms existing approaches. For future work, we plan to develop a method for modeling the temporal evolution of node importance in a KG.

3.8 Appendix

In the appendix, we present experimental settings.

3.8.1 Experimental Settings

PageRank (PR) and Personalized PageRank (PPR). We used NetworkX 2.3's *pagerank_scipy* function to run PR and PPR. We used the default parameter values set by NetworkX, including the damping factor of 0.85 for both algorithms.

HAR. We implemented HAR in Python 3.7. In experiments, we set α and β to 0.15 and γ to 0. We ran the algorithm for 30 iterations. Normalized input signal values were used as the probability of entities (as in PPR). All relations were assigned an equal probability. Among hub and authority scores HAR computes for each entity, we used the maximum of the two values as HAR's estimation, and we observed similar results when reporting only one type of scores.

node2vec. We used the reference node2vec implementation[5] to generate node features. For MUSIC10K, FB15K, and TMDB5K, we set the number of dimensions to 64, and for IMDB, we set it to 128. For other parameters, we used the default values used by the reference implementation.

GENI. We implemented GENI using the Deep Graph Library 0.3.1. We used GENI with two layers, each consisting of four attention heads. For the scoring network, we used a two-layer fully-connected neural network, where the number of hidden neurons in the first hidden layer was 75% of the input feature dimension. We set the dimensions of predicate embedding to 10. GENI was trained using Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$, a learning rate of 0.005, and a weight decay of 0.0005. We applied ReLU to estimated node importance, ELU to node centrality, and Leaky ReLU to unnormalized attention coefficients.

MULTIIMPORT. We implemented MULTIIMPORT using the Deep Graph Library 0.3.1. For a fair comparison with GENI, we used two layers for MULTIIMPORT, and each layer contained four attention heads. The output from each attention head was averaged, and fed into the next layer. For $g(\cdot)$, we used a linear transformation with bias, which projects input features to 75% of the input feature dimension. For $g'(\cdot)$, we used another linear transformation with bias. We set the dimension of predicate embedding to 10. For training, we used the Adam optimizer with $\beta_1 = 0.9, \beta_2 = 0.999$, and a learning rate of 0.005. We set λ to 0.001. We set ν to 0.0002 on FB15K and IMDB, applying the corresponding loss term in Equation (3.12) to 20% and 5% of randomly sampled edges, respectively; ν was set 0.0 on MUSIC10K and TMDB5K. ELU was applied for node centrality.

Early Stopping. For MULTIIMPORT and GENI, we applied early stopping based on the performance on the validation data set (15% of training data) with a patience of 30, and set the maximum number of iterations to 3000. For testing, we used the model that achieved the best validation performance.

Input Signal Preprocessing. We applied log transformation for all signals as their distribution is highly skewed, except for those signals available on MUSIC10K as they are normalized scores.

[5] https://snap.stanford.edu/node2vec/

Chapter based on work published in ICDM 2020 [PKFD20].

Chapter 4

Principled and Scalable Recommendation Justification

Online recommendation is an essential functionality across a variety of services, including e-commerce and video streaming, where items to buy, watch, or read are suggested to users. Justifying recommendations, i.e., explaining why a user might like the recommended item, has been shown to improve user satisfaction and persuasiveness of the recommendation. In this chapter, we develop a method for generating post-hoc justifications that can be applied to the output of any recommendation algorithm. Existing post-hoc methods are often limited in providing diverse justifications, as they either use only one of many available types of input data, or rely on the predefined templates. We address these limitations of earlier approaches by developing J-RECS, a method for producing concise and diverse justifications. J-RECS is a recommendation model-agnostic method that generates diverse justifications based on various types of product and user data (e.g., purchase history and product attributes). The challenge of jointly processing multiple types of data is addressed by designing a principled graph-based approach for justification generation. In addition to theoretical analysis, we present an extensive evaluation on synthetic and real-world data. Our results show that J-RECS satisfies desirable properties of justifications, and efficiently produces effective justifications, matching user preferences up to 20% more accurately than baselines.

4.1 Introduction

Recommender systems have a profound and ever increasing impact on how online users make purchase decisions, consume various types of content, and engage with the service. While recommender systems have seen significant progress in terms of recommendation accuracy, algorithms widely used in practice are mostly black boxes. This includes recommenders based on the latent factor models such as matrix factorization [KBV09,

ZWFM06], as well as some deep learning-based recommenders [WWY15, FML+19]. Such systems can be limited in their ability to justify recommendations.

Justification refers to explaining why a user might like the recommended item [BC17]. In other words, while recommendations suggest users *what* they might like, justifications reveal *why* the recommended item might match their preferences. For instance, a list of recommended products can be supplemented with a justification that "these items are similar to what you recently purchased." Several studies have shown that justifications can improve user satisfaction [HKR00], increase the persuasiveness and reliability of recommendations [TM07, TM15], and help users make more accurate and efficient decisions [BM05].

In this chapter, we focus on post-hoc justification of recommendations. In post-hoc approaches, recommendations and justifications are decoupled from each other; that is, justifications are generated after the recommendation has been given. The main advantage of generating justifications post-hoc is that post-hoc methods can be easily applied to different types of recommendation algorithms (thus *recommendation model-agnostic*), which allows a greater freedom in the design of explanations [VSR09].

Existing post-hoc methods typically select justifications from predefined templates [ZC20], such as "your neighbors' rating for this item is ..." [HKR00], or they provide justifications based on only one type of data, such as keywords [BM05], although many types of data are often available. While these methods have been shown to produce concise justifications, they are limited in their ability to provide diverse justifications. Moreover, some of these methods generate justifications in a non-personalized manner [DHW+17], while other post-hoc methods require labeled ground truth data to train a justification model [NLM19], thus posing an additional hurdle.

In summary, major challenges of generating post-hoc justifications are in handling heterogeneous data (e.g., user purchase history, product attributes and reviews) to generate flexible and diversified justifications without the need for manually labeled data, while enabling that the justification diversity can be increased without changing the underlying algorithm. We address these challenges by proposing a novel principled graph-based method called J-RECS. We use the graph to represent heterogeneous data that can be leveraged for justifications. Moreover, the graph-based representation allows us to generate justifications personalized with respect to both the user and the recommended item. Finally, we derive an objective function that agrees with intuition and leads to concise and diverse justifications. This chapter makes the following contributions.

- **Problem Formulation.** We present a graph-based formulation of the problem of generating concise and diverse justifications given various types of user and product data.
- **Principled Approach.** We develop J-RECS, a principled post-hoc framework to infer justifications. J-RECS is guided by a set of principles characterizing desired justifications, and does not require manually labeled data.
- **Effectiveness.** We demonstrate that J-RECS satisfies desirable properties of justifi-

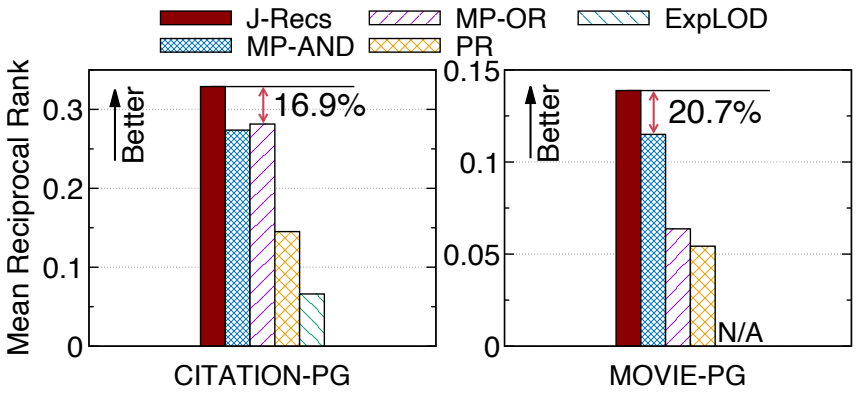


Figure 4.1: J-RECS generates justifications that match user preferences better than existing methods. Higher values are better. See Section 4.4.3 for details.

cations, and show the effectiveness of J-RECS in experiments on real-world data (Figure 4.1).

- **Scalability.** Our proposed J-RECS is scalable, and runs in time linear in the size of input data (Figure 4.5).

The rest of the chapter is organized as follows. We formulate the problem of graph-based recommendation justification and present our framework in Section 4.2. Then we provide evaluation results using axioms and real-world data in Sections 4.3 and 4.4, respectively. After discussing related work in Section 4.5, we conclude in Section 4.6.

4.2 Justifying Recommendations

In this section, we first provide the problem statement and define the product graph and justifications. We then describe how J-RECS identifies good justifications efficiently. The symbols used in this chapter is given in Table 4.1.

4.2.1 Problem Statement

Products can be any item, such as movies, audio tracks, and papers, which can be suggested by recommender systems. Let $\mathcal{P} = \{p_1, p_2, \ldots, p_L\}$ denote the set of all products. We are given a product $r_u \in \mathcal{P}$, which is recommended to user u by an external recommendation algorithm. We also have a set $\mathcal{Q}_u \subseteq \mathcal{P}$ of products, to which user u gave positive feedback. For instance, $\mathcal{Q}_u$ can be the products user u purchased or rated highly. Let q_i^u denote the i-th product in $\mathcal{Q}_u$. For simplicity, we may omit subscript u, and use Q, r, and q_i. Now consider various types of product information, which we collectively denote by $\mathcal{D}$. Examples of product data include the followings.

- Product details: e.g., flavor, category, color of a product; actors, directors, and genres of a movie
- Product keywords: e.g., movie keywords submitted by users
- Product reviews; sentences in the product reviews
- Product co-purchase and co-view records

Given these input data, our problem is stated as follows:

> Given products $\mathcal{P}$, product data $\mathcal{D}$, recommended product $r_u \in \mathcal{P}$, and products $\mathcal{Q}_u \subseteq P$ that received positive feedback by user u, efficiently infer justifications relevant to the recommendation r_u in a way that best reflects the user's preference expressed by $\mathcal{Q}_u$.

In the following sections, we further formalize this problem by defining the notion of justification, justification score, and the optimization objective.

4.2.2 Product Graph and Justifications

A good justification needs to capture the user's preference, while being relevant to the recommended item. Product data provide useful information that can help with identifying good justifications. A major challenge in effectively employing product data lies in how to jointly take into account various sources of information in the product data.

To address this challenge, we need to be able to measure the relevance and similarity between products and product data. We note that each type of product data provides a signal that lets us identify a set of products that are similar in some specific respect. For instance, products with the "chocolate flavor" are likely to have a similar taste, and movies that share several keywords tend to have a lot of similarity. Also, knowledge of similar products can enable us to see the relatedness of seemingly different product attributes. To make the most of this mutually influential relationship, we combine all available information into a graph, which we call a *product graph*, and find good justifications in terms of it.

Product Graph. We refer to an instance of specific product data as *"attribute entity"* (or *"attribute"* in short) and use the term *"attribute type"* to denote a specific type of product data. For instance, each one of the examples given for the product data (e.g., movie genre, product color, review) represents one attribute type, and science fiction is an attribute of the movie genre type. We denote the set of all product attributes by $\mathcal{A}$. Products $\mathcal{P}$ and their attributes $\mathcal{A}$ are nodes in a product graph $\mathcal{G}$. A product graph can also have *non-attribute entities* as nodes, which are entities not directly connected to products. Instead, they connect similar product attributes, enabling us to identify similar products that do not share the same attributes. Examples include facts common to actors, and common review keywords. Figure 4.2 shows an example product graph.

Justification. We aim to find a set of relevant product attributes, such as "red color" and "high efficiency", and produce justifications in a format determined by the type of selected attributes. More precisely, given a product graph $\mathcal{G}$ and the recommended item r_u, justifications for r_u are a subset $\mathcal{A}' \subseteq \mathcal{A}$ of attributes selected among those that are connected to r_u in $\mathcal{G}$. In Figure 4.2, boxes with a bold red outline denote the attributes that are chosen to form justifications.

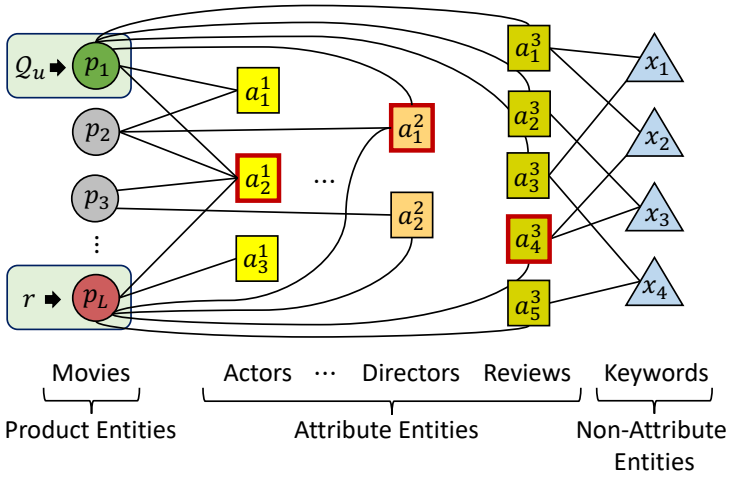


Figure 4.2: A movie product graph. $\mathcal{Q}_u$ denotes the set of products that received positive feedback from user u, and r denotes the recommended product. Boxes with a bold red outline denote the attribute entities that are selected to form justifications.

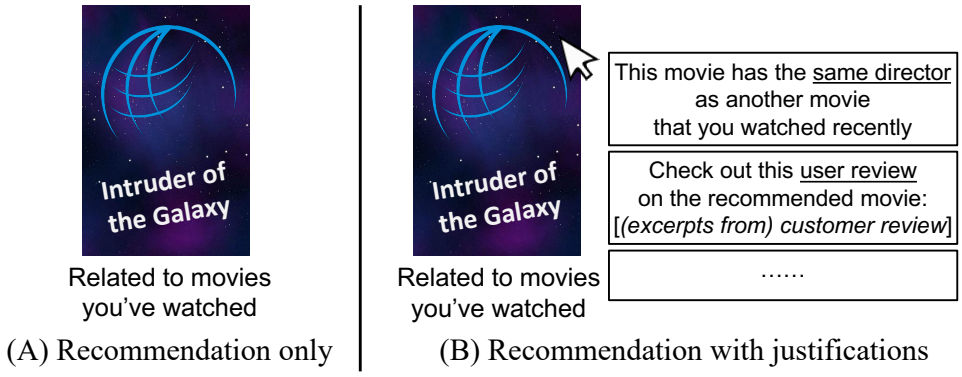


Figure 4.3: Product graph-based justifications enrich the recommendation with relevant and diverse information on why the user might like the recommended item.

A product attribute, appropriately chosen in light of user preferences, naturally lends itself to an intuitive and concise justification. For instance, when the director James Cameron is chosen for a movie recommendation, this translates to the justification that *"We recommend this movie directed by James Cameron, who made movies that are similar to other movies you watched."* When the user watched some of James Cameron's movies, we can further enrich this justification with that information. Similarly, when a review gets selected, this translates to the justification like *"Check out this user review, which closely reflects your preference, and explains why you should consider buying this item."* Figure 4.3 illustrates how such justifications enrich a movie recommendation, making the recommendation more persuasive. As in these examples, the final justifications can be generated based on custom rules that consider the attribute type and previous user actions. Note that, since our justifications are based on the product graph, the diversity of justifications

increases as we add more product data to it, without needing to make changes to the framework.

4.2.3 Quantifying the Quality of Justifications

To select good justifications tailored to the user and the recommendation, we need to be able to measure the goodness of product attributes. Towards this goal, we design justification scores, such that higher scores indicate better justifications in consideration of the product graph $\mathcal{G}$, products $\mathcal{Q}_u$, and the recommended item r. We quantify the justification score by considering two aspects, namely, relevance and diversity.

4.2.3.1 Relevance Score

Intuitively, a good justification should be highly relevant to the recommended product r. To measure the relevance of a product attribute, we consider how probable attribute a is given the recommended product r, that is,

$$\Pr(A = a|R = r) \tag{4.1}$$

where R and A are random variables denoting a recommended product, and an attribute of the recommended product, respectively. Consider a random variable U, which denotes the product that matches the preference of user u. Given L products $p_1, p_2, \ldots, p_L$ in $\mathcal{P}$, applying the sum rule of probability, the likelihood (4.1) can be expressed as:

$$\Pr(A = a|R = r) = \sum_{i=1}^{L} \Pr(A = a, U = p_i|R = r). \tag{4.2}$$

Among the L terms in (4.2), we know the user's preferences on the products in $\mathcal{Q}_u$, and have no information for other products. So, we assume that the products to which user u gave no feedback are equally likely given the recommended item. Specifically, we assume that $P(A=a, U=p_i|R=r) = P(A=a, U=p_j|R=r)$ for all $p_i, p_j \notin \mathcal{Q}_u$ and any attribute $a \in \mathcal{A}$. In other words, in measuring the relevance score, our focus is on the sum of likelihood terms of products in $\mathcal{Q}_u$:

$$\sum_{i=1}^{|\mathcal{Q}_u|} \Pr(A = a, U = q_i|R = r). \tag{4.3}$$

We observe that, by the product rule of probability, the following holds for each of the $|\mathcal{Q}_u|$ terms in (4.3):

$$\Pr(a, q_i|r) = \underbrace{\Pr(a|q_i, r)}_{\substack{\text{Attribute} \\ \text{relevane}}} \cdot \underbrace{\Pr(q_i|r)}_{\substack{\text{Feedback} \\ \text{relevance}}} \tag{4.4}$$

where we have omitted random variables to simplify notations. In (4.4), the first term $\Pr(a|q_i, r)$ denotes the likelihood of attribute a given product q_i and recommended

Table 4.1: Table of symbols.

Symbol	Definition
$\mathcal{P}$	Set of L products ($\mathcal{P} = \{p_1, p_2, \ldots, p_L\}$)
$r_u(r)$	Product recommended to user u
$\mathcal{Q}_u(\mathcal{Q})$	Set of products to which user u gave positive feedback
$q_i^u(q_i)$	i-th product in $\mathcal{Q}_u$ (i.e., $q_i \in \mathcal{Q}_u$)
a	Product attribute entity
$\mathcal{A}$	Set of all attribute entities
$\mathcal{A}_{(r)}$	Attribute entities of product r
$\mathcal{A}'$	Set of selected attribute entities
ρ	Relative weight of r_u in comparison to q_i
λ_1, λ_2	Non-negative weights for the justification diversity
B	Budget (maximum number of justifications to select)

item r, which closely matches our goal of finding a justification relevant to both the recommendation and the user's preference. The second term $\Pr(q_i|r)$ represents how probable product q_i is, given recommendation r. This term measures the relevance of feedback $q_i \in \mathcal{Q}_u$ with respect to r, which enables considering the fact that some of the products in $\mathcal{Q}_u$ may not be relevant to the current recommendation. In other words, $\Pr(q_i|r)$ acts as a weight for the attribute relevance term.

Let $R_r(a)$ denote the relevance score of attribute a as a justification of the recommendation r, which we define to be

$$R_r(a) = \sum_{i=1}^{|\mathcal{Q}_u|} \Pr(a|q_i, r) \cdot \Pr(q_i|r). \tag{4.5}$$

Modeling Attribute and Feedback Relevance. We model the two relevance terms in (4.4) in terms of the product graph $\mathcal{G}$, as it provides rich information on how products and attributes are related to each other. Specifically, we consider a random walk using personalized PageRank (PPR) [Hav02] over $\mathcal{G}$, and model the attribute relevance, $\Pr(a|q_i, r)$, by the proximity of a with respect to q_i and r in terms of PPR on $\mathcal{G}$. Intuitively, given a random walker who travels over $\mathcal{G}$ and returns to q_i and r with a fixed probability, an attribute which gets visited more times than others is deemed more likely than other attributes, in terms of $\mathcal{G}$ given q_i and r. We use the following notation

$$\text{PPR}(a|q_i = 1 - \rho, r = \rho) \tag{4.6}$$

to denote the PPR score of attribute a with respect to q_i and r, in which q_i and r are given the probability mass of $1 - \rho$ and ρ in the personalization vector, respectively, and ρ controls the relative importance of r in comparison to q_i.

Note that PPR scores are computed for all nodes in $\mathcal{G}$, while we are concerned about the attributes of the recommended item. So we normalize PPR scores such that the scores

of recommended item's attributes sum to one, denoting the normalized score by nPPR. Then, $\Pr(a|q_i, r)$ is computed as

$$\Pr(a|q_i, r) \triangleq \text{nPPR}(a|q_i = 1 - \rho, r = \rho). \tag{4.7}$$

Similarly, we model the feedback relevance, $\Pr(q_i|r)$, using PPR with respect to r over $\mathcal{G}$, this time normalizing PPR scores over products in $\mathcal{Q}$. Thus, $\Pr(q_i|r)$ is compute as

$$\Pr(q_i|r) \triangleq \text{nPPR}(q_i|r = 1). \tag{4.8}$$

By (4.5), and our choice given by (4.7) and (4.8) to use nPPR on $\mathcal{G}$ to model the relevance terms, we define $R_r(a)$ as follows:

$$R_r(a) \triangleq \sum_{i=1}^{|\mathcal{Q}_u|} \text{nPPR}(q_i|r=1) \cdot \text{nPPR}(a|q_i=1-\rho, r=\rho). \tag{4.9}$$

Finally, based on the above definition of attribute relevance (4.9), we define the relevance score $R_r(\mathcal{A}')$ of a set $\mathcal{A}' \subseteq \mathcal{A}$ of attributes with respect to recommended product r to be:

$$R_r(\mathcal{A}') \triangleq \sum_{a \in \mathcal{A}'} R_r(a). \tag{4.10}$$

We show in Sections 4.3 and 4.4 that our proposed approach yields more accurate relevance scores, which agree with our intuition, than other choices to model the attribute relevance.

4.2.3.2 Diversity Score

To provide informative and engaging justifications to the user, we want the justifications to consist of diverse product attributes. For example, users would find it more interesting to see a combination of relevant reviews, product features, and purchasing history than seeing only reviews. Diversity has multiple aspects to it, and some aspects may be application dependent. Below we introduce two diversity aspects. Note that our method can easily be extended to incorporate different aspects of justification diversity.

We first consider the diversity in terms of attribute types. Let a_t denote the type of attribute a. We capture this diversity using the number of attribute types covered by the selected attributes $\mathcal{A}'$:

$$C_{\text{Type}}(\mathcal{A}') = |\{a_t : a \in \mathcal{A}'\}|. \tag{4.11}$$

Secondly, for textual product data such as customer reviews, we may consider the diversity of their topics and sentiments, which can be extracted by existing methods, e.g.,

by applying the latent Dirichlet allocation to product reviews with a decision threshold. Let $\mathcal{T}$ denote the set of such topics of the textual data, and $\mathcal{T}(a) \subseteq \mathcal{T}$ be the set of topics attribute a represents. We capture this second diversity aspect using the number of topics attributes $\mathcal{A}'$ represent, defined by:

$$C_{\text{Topic}}(\mathcal{A}') = \left| \bigcup_{a \in \mathcal{A}'} \mathcal{T}(a) \right|. \tag{4.12}$$

4.2.3.3 Justification Score

Our goal is multi-objective as we aim to produce justifications with high relevance and diversity. We cast this into a single-objective optimization problem using a weighted sum scalarization. Since relevance and diversity terms can have different magnitude, we adopt the normalization scheme given in [GR06] such that each term is to be bounded between 0 and 1. For example, we define the first diversity term D_{Type} to be:

$$D_{\text{Type}}(\mathcal{A}') = \frac{C_{\text{Type}}(\mathcal{A}') - C^{\text{Min}}_{\text{Type}}}{C^{\text{Max}}_{\text{Type}} - C^{\text{Min}}_{\text{Type}}}, \tag{4.13}$$

where $C^{\text{Max}}_{\text{Type}} = \max_{\mathcal{A}_B}\{C_{\text{Type}}(\mathcal{A}_B) | \mathcal{A}_B \subseteq \mathcal{A}, 0 < |\mathcal{A}_B| \leq B\}$ and $C^{\text{Min}}_{\text{Type}} = \min_{\mathcal{A}_B}\{C_{\text{Type}}(\mathcal{A}_B) | \mathcal{A}_B \subseteq \mathcal{A}, 0 < |\mathcal{A}_B| \leq B\}$, with B denoting the maximum number of attributes to be selected. Note that $0 \leq D_{\text{Type}}(\mathcal{A}') \leq 1$. In the event that $C^{\text{Max}}_{\text{Type}}$ and $C^{\text{Min}}_{\text{Type}}$ are equivalent, we define $D_{\text{Type}}(\mathcal{A}') = 1$.

Applying the same normalization, we define the normalized relevance score and the topical diversity term as follows:

$$nR_r(\mathcal{A}') = \left(R_r(\mathcal{A}') - R^{\text{Min}}_r\right) / \left(R^{\text{Max}}_r - R^{\text{Min}}_r\right) \tag{4.14}$$

$$D_{\text{Topic}}(\mathcal{A}') = \left(C_{\text{Topic}}(\mathcal{A}') - C^{\text{Min}}_{\text{Topic}}\right) / \left(C^{\text{Max}}_{\text{Topic}} - C^{\text{Min}}_{\text{Topic}}\right) \tag{4.15}$$

where $R^{\text{Max}}_r, R^{\text{Min}}_r, C^{\text{Max}}_{\text{Topic}}, C^{\text{Min}}_{\text{Topic}}$ are defined in the same way as in the first term, using $R_r(\mathcal{A}')$ and $C_{\text{Topic}}(\mathcal{A}')$ instead of $C_{\text{Type}}(\mathcal{A}')$. In sum, given a recommended item r, we define the justification score $J_r(\mathcal{A}')$ of a set $\mathcal{A}' \subseteq \mathcal{A}$ of attributes as:

$$J_r(\mathcal{A}') = nR_r(\mathcal{A}') + \lambda_1 \cdot D_{\text{Type}}(\mathcal{A}') + \lambda_2 \cdot D_{\text{Topic}}(\mathcal{A}') \tag{4.16}$$

where λ_1 and λ_2 are non-negative weights for diversity terms.

4.2.4 Justification Discovery

Based on the above definition of relevance, diversity, and justification scores, we formally define the justification discovery problem as follows.

Problem 4.1. Justification Discovery:
Given a product graph $\mathcal{G}$, a recommended item $r \in \mathcal{P}$, products $\mathcal{Q} \subseteq \mathcal{P}$ that received positive feedback, and a budget B, find a set $\mathcal{A}^* \subseteq \mathcal{A}$ of product attributes that maximizes the justification score, i.e.,

$$\mathcal{A}^* = \arg\max_{\mathcal{A}' \subseteq \mathcal{A}} J_r(\mathcal{A}') \text{ such that } |\mathcal{A}'| \leq B. \tag{4.17}$$

Due to the combinatorial nature of this optimization problem, solving it exactly is computationally intractable. Instead, we show that the objective (4.17) is submodular, which allows us to efficiently obtain near-optimal justifications.

Theorem 4.1.:
The justification score $J_r(\mathcal{A}')$ given by (4.16) is a non-negative, monotone, submodular function.

Proof. (a) $J_r(\mathcal{A}')$ is non-negative since it is a weighted sum of three non-negative scores with non-negative weights.

(b) A set function f is monotone if for every $\mathcal{A}_1 \subseteq \mathcal{A}_2$, we have that $f(\mathcal{A}_1) \leq f(\mathcal{A}_2)$. Given that

$$\begin{aligned} R_r(\mathcal{A}_1) &= \sum\nolimits_{a \in \mathcal{A}_1} R_r(a) \leq \sum\nolimits_{a \in \mathcal{A}_2} R_r(a) = R_r(\mathcal{A}_2), \\ C_{\text{Type}}(\mathcal{A}_1) &= |\{a_t : a \in \mathcal{A}_1\}| \leq |\{a_t : a \in \mathcal{A}_2\}| = C_{\text{Type}}(\mathcal{A}_2), \\ C_{\text{Topic}}(\mathcal{A}_1) &= \left|\bigcup\nolimits_{a \in \mathcal{A}_1} \mathcal{T}(a)\right| \leq \left|\bigcup\nolimits_{a \in \mathcal{A}_2} \mathcal{T}(a)\right| = C_{\text{Topic}}(\mathcal{A}_2), \end{aligned}$$

$J_r(\mathcal{A}')$ is monotone as it is a non-negative weighted sum of these monotone scores.

(c) Given a set function f of attributes $\mathcal{A}'$, and attribute a, let $\Delta f(a|\mathcal{A}') = f(\mathcal{A}' \cup \{a\}) - f(\mathcal{A}')$ be the marginal gain of adding a to $\mathcal{A}'$. Then f is submodular if for every $\mathcal{A}_1, \mathcal{A}_2 \subseteq \mathcal{A}$ with $\mathcal{A}_1 \subseteq \mathcal{A}_2$ and every $a \in \mathcal{A} \setminus \mathcal{A}_2$, it holds that $\Delta f(a|\mathcal{A}_1) \geq \Delta f(a|\mathcal{A}_2)$.

As every $a \in \mathcal{A} \setminus \mathcal{A}_2$ results in the same marginal gain $R_r(a)$ to both $R_r(\mathcal{A}_1)$ and $R_r(\mathcal{A}_2)$, $\Delta R_r(a|\mathcal{A}_1) = \Delta R_r(a|\mathcal{A}_2)$.

Let $\mathcal{T}(\mathcal{A}') = \bigcup_{a \in \mathcal{A}'} \mathcal{T}(a)$. The topics $\mathcal{T}(a)$ covered by a can be classified into three cases. The first case is the set $\mathcal{T}_1 \subseteq \mathcal{T}(a)$ of topics that belong to $\mathcal{T}(\mathcal{A}_1)$. Since $\mathcal{A}_1 \subseteq \mathcal{A}_2$, $\mathcal{T}_1 \subseteq \mathcal{T}(\mathcal{A}_2)$; thus, these topics make no additional contributions to both $D_{\text{Topic}}(\mathcal{A}_1)$ and $D_{\text{Topic}}(\mathcal{A}_2)$. The second case is the set $\mathcal{T}_2 \subseteq \mathcal{T}(a)$ of topics that do not belong to $\mathcal{T}(\mathcal{A}_1)$, but belong to $\mathcal{T}(\mathcal{A}_2)$. Since these topics already belong to $\mathcal{T}(\mathcal{A}_2)$, they make positive contributions to $D_{\text{Topic}}(\mathcal{A}_1)$, while making no contributions to $D_{\text{Topic}}(\mathcal{A}_2)$. The third case is the set $\mathcal{T}_3 \subseteq \mathcal{T}(a)$ of topics that do not belong to both $\mathcal{T}(\mathcal{A}_1)$ and $\mathcal{T}(\mathcal{A}_2)$. In this case, the topics in $\mathcal{T}_3$ make equal contributions to $D_{\text{Topic}}(\mathcal{A}_1)$ and $D_{\text{Topic}}(\mathcal{A}_2)$. Thus, in all cases, $\Delta D_{\text{Topic}}(a|\mathcal{A}_1) \geq \Delta D_{\text{Topic}}(a|\mathcal{A}_2)$. The same argument applies to show that $\Delta D_{\text{Type}}(a|\mathcal{A}_1) \geq \Delta D_{\text{Type}}(a|\mathcal{A}_2)$.

Algorithm 4.1: J-RECS algorithm.

Input: Product graph $\mathcal{G}$, recommended product r, products $\mathcal{Q}$ with positive feedback, budget B.
Output: Justifications $\mathcal{A}^*$ s.t. $|\mathcal{A}^*| \leq B$.

```
Compute the relevance score R_r(a) given by (4.9).
A* ← ∅
while |A*| ≤ B do
    a* ← arg max_{a ∈ A_(r) \ A*} (J_r(A* ∪ {a}) − J_r(A*))
    A* ← A* ∪ {a*}
return A*
```

Therefore, since three functions $R_r(\mathcal{A}')$, $D_{\text{Type}}(\mathcal{A}')$, and $D_{\text{Topic}}(\mathcal{A}')$ are all submodular, the justification score $J_r(\mathcal{A}')$, which is a weighted sum of these submodular functions with non-negative weights, is also submodular. ■

Theorem 4.2.:
Problem 4.1 admits a $(1-\frac{1}{e})$-approximation.

Proof. Maximizing a non-negative, monotone, submodular function subject to a cardinality constraint admits a $(1-\frac{1}{e})$ approximation under a greedy approach in which the item with the largest marginal gain is selected at each step [NW78]. This theorem follows since our maximization objective $J_r(\mathcal{A}')$ is non-negative, monotone, and submodular by Theorem 4.1. ■

J-RECS algorithm. Theorem 4.2 leads to the J-RECS algorithm in Algorithm 4.1, which finds a $(1-\frac{1}{e})$-approximation of the optimal justifications. In Algorithm 4.1, we first compute the relevance score of all attributes based on (4.9). Then, we repeatedly find the product attribute from $\mathcal{A}_{(r)}$ with the greatest marginal gain, where $\mathcal{A}_{(r)}$ is the attributes of product r, and add it to $\mathcal{A}^*$ until we exhaust the given budget B.

Theorem 4.3.:
Algorithm 4.1 runs in $O(|E|)$ time with $O(|\mathcal{Q}|)$ processors, taking $O(|E||\mathcal{Q}|)$ steps in total, assuming that $BM < |V|$, where $M = \max(N_1, N_2)$ with N_1 and N_2 denoting the number of attribute types and topics, respectively.

Proof. Computing the relevance score in (4.9) for all attributes involves computing the PPR scores with respect to $|\mathcal{Q}| + 1$ personalization vectors. Using a power iteration with sparse matrix multiplications, PPR can be computed in $O(|E|)$ time. Since PPR computations are independent of each other, they can be completed in $O(|E|)$ time using $O(|\mathcal{Q}|)$ processors.

Computing $C^{\text{Max}}_{\text{Topic}}$ corresponds to the maximum coverage problem. As this is NP-hard, we use a greedy approximation algorithm [KT06], which takes $O(BM)$ time. Other terms for score normalization (e.g., $C^{\text{Max}}_{\text{Type}}$) can also be computed in $O(BM)$ time. Then we

select up to B justifications: Selecting one justification requires evaluating the marginal gain $J_r(\mathcal{A}^* \cup \{a\}) - J_r(\mathcal{A}^*)$ for all $a \in \mathcal{A}_{(r)} \setminus \mathcal{A}^*$. As evaluating the marginal gain with respect to each term in (4.16) takes $O(M)$, greedy selection takes $O(BM|\mathcal{A}_{(r)}|)$ steps in total. Given that $|\mathcal{A}_{(r)}| < |V|$ and assuming $BM < |V|$, which is true in most cases, the running time for the greedy selection is $O(|E|)$. ■

4.3 Evaluation Using Axioms

In this section, we evaluate different approaches to measure attribute relevance, using what we call *axioms*, which are a set of product graphs with an intuitive expected outcome. After describing axioms, we introduce other approaches to compute attribute relevance, and discuss how well axioms are satisfied by different approaches. Experimental settings used for this evaluation are given in Section 4.7.1.

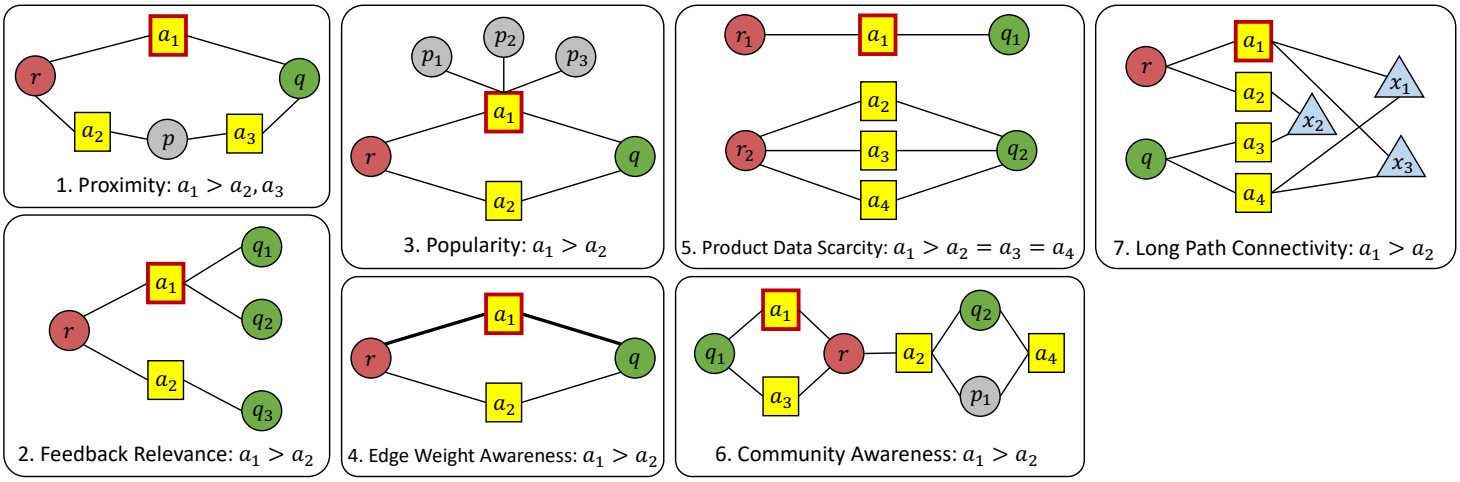


Figure 4.4: Axioms and expected relevance scores (4.9) of product attributes (see Section 4.3.1 for details).

4.3.1 Axioms

An axiom is a small product graph with expected relevance scores (4.9) for some product attributes. We use the axioms shown in Figure 4.4 to evaluate different methods for measuring attribute relevance. In Figure 4.4, squares denote product attributes, and circles denote products, where r is the recommended product, q is the product with positive user feedback, and p is a product the user had no interaction with.

1. Proximity. Attributes that are closer to products r and q should receive a higher relevance score. Given that a_1 is directly connected to r and q (i.e., a_1 is a shared attribute of r and q), while other attributes are three hops away from either r or q, we expect $R_r(a_1) > R_r(a_2), R_r(a_3)$.

2. Feedback Relevance. Attributes that receive more support from the products in $\mathcal{Q}$ should receive a higher relevance score. While both a_1 and a_2 are directly connected

Table 4.2: Our proposed J-RECS satisfies all axioms in Figure 4.4, while other alternatives fail at least one of them.

Method \ Axioms	1. Proximity	2. Feedback Relevance	3. Popularity	4. Edge Weight Awareness	5. Product Data Scarcity	6. Community Awareness	7. Long Path Connectivity
J-RECS	✓	✓	✓	✓	✓	✓	✓
ExpLOD [MNL+16]	✓					✓	
MP-AND [TF06]	✓	✓	✓	✓	✓		✓
MP-OR [TF06]	✓	✓	✓	✓	✓		✓
$BA^{Sink}_{q\to r}$ [TPF+10]	✓	✓		✓	✓	✓	
$BA^{Del}_{q\to r}$ [TPF+10]	✓	✓			✓		
$BA^{Sink}_{r\to q}$ [TPF+10]	✓	✓		✓	✓	✓	
$BA^{Del}_{r\to q}$ [TPF+10]	✓	✓			✓	✓	
MP-AND w/ HAR [LNY12] authority score	✓	✓	✓	✓	✓		
MP-OR w/ HAR [LNY12] authority score		✓	✓	✓	✓		
$BA^{Sink}_{q\to r}$ w/ HAR [LNY12] authority score	✓	✓			✓		
$BA^{Del}_{q\to r}$ w/ HAR [LNY12] authority score	✓				✓		
$BA^{Sink}_{r\to q}$ w/ HAR [LNY12] authority score	✓	✓		✓	✓	✓	
$BA^{Del}_{r\to q}$ w/ HAR [LNY12] authority score					✓	✓	✓

to r, a_1 is covered by a greater number of products in $\mathcal{Q}$. Thus, we expect $R_r(a_1) > R_r(a_2)$.

3. Popularity. More widely used attributes should receive a higher relevance score (e.g., consider explaining a movie recommendation with a popular actor vs. an actor who appeared only in that movie). Thus, we require that $R_r(a_1) > R_r(a_2)$.

4. Edge Weight Awareness. Attributes that are connected via edges of higher weight should receive a higher relevance score, as edge weight indicates the importance of each connection in the product graph. Thus, we expect $R_r(a_1) > R_r(a_2)$. Note that satisfying this axiom is important to reflect the relative importance among different attribute types. For instance, while country attributes would be one of the most popular ones in the movie product graph, it may not be a very interesting justification to the user. Based on this prior knowledge, we can downplay the country type if our method satisfies this axiom.

5. Product Data Scarcity. Attributes of the product that contain scarcer information should receive a higher relevance score. This axiom consists of two product graphs, in which each attribute is directly connected to r and q. While each graph contains only one product in $\mathcal{Q}$, the positive feedback expressed by $\mathcal{Q}$ is attributed to multiple attributes in the larger graph, leading to each one of them being a weaker evidence of user's preference than the other product's unique attribute. Thus, we expect $R_r(a_1) > R_r(a_i)$ for $i \neq 1$.

6. Community Awareness. When the recommended item belongs to one community (e.g., a group of shoes), it is desirable to put more weight on attributes belonging to the same community than on others (e.g., a group of electronic devices), as similar products tend to share more attributes (e.g., product details, keywords) with each other than with different products. In this axiom, there are two small communities of products and attributes, where r and a_2 act as a bridge between communities. Also, each community has only one product with positive feedback. Therefore, we expect $R_r(a_1) > R_r(a_2)$.

7. Long Path Connectivity. Even when attributes are not directly connected to r and q, attributes more strongly connected to r and q should receive a higher relevance score. Here, a_1 is more strongly linked to q via x_1 and x_3 than a_2, which is linked to q only via x_2. Thus, we expect $R_r(a_1) > R_r(a_2)$.

4.3.2 Baselines

Below we use the same notation as in (4.6) to specify the query nodes. Table 4.1 provides the definition of symbols.

4.3.2.1 Relevance Models

We consider three approaches that measure the relevance of attribute a, given recommendation r and products $q_1, \ldots, q_{|\mathcal{Q}_u|}$.

ExpLOD [MNL+16] assigns a high score to those attributes that are highly connected to the products in $\mathcal{Q}_u \cup \{r\}$, using the following formula:

$$\text{ExpLOD}(a) = \left(\alpha \cdot \frac{n_{a,\mathcal{Q}_u}}{|\mathcal{Q}_u|} + \beta \cdot n_{a,r} \right) \cdot \text{IDF}_a \tag{4.18}$$

where $n_{a,\mathcal{Q}_u}$ is the number of edges between product attribute a and the products in $\mathcal{Q}_u$, $n_{a,r}$ is the number of edges between a and r, and IDF_a is the reciprocal of the number of products that are described by attribute a.

Meeting Probability (MP) [TF06] assigns a high relevance score to an attribute that is close to the recommended item r and products $|\mathcal{Q}_u|$. MP has been successfully used in identifying nodes that have strong connections to the query nodes. We consider the following two MP scores.

$$\text{MP-AND}(a) \triangleq \prod_{p \in \mathcal{Q}_u \cup \{r\}} \text{PPR}(a|p{=}1) \tag{4.19}$$

$$\text{MP-OR}(a) \triangleq 1 - \prod_{p \in \mathcal{Q}_u \cup \{r\}} (1 - \text{PPR}(a|p{=}1)) \tag{4.20}$$

BASSET (BA) [TPF+10] aims to identify a small number of good gateway nodes between a source node s and a target node t in the given graph. Let $\text{PPR}^{\text{Sink}}_{\mathcal{I}}(t|s=1)$ denote the PPR score from source s to target t, after setting the nodes denoted by $\mathcal{I}$ as sinks (i.e., nodes with no outgoing edges). We also consider a related model, $\text{PPR}^{\text{Del}}_{\mathcal{I}}(t|s=1)$, in

which we delete the nodes denoted by $\mathcal{I}$, instead of making them sinks. Note that, in out setting, given a recommendation r and a product q, we can consider two directions of $q \to r$ and $r \to q$. For instance, BA score of attribute a between source q and target r using $\mathrm{PPR}^{\mathrm{Sink}}_{\mathcal{I}}$ is defined as:

$$\mathrm{BA}^{\mathrm{Sink}}_{q \to r}(a) \triangleq \sum_{i=1}^{|\mathcal{Q}_u|} \mathrm{PPR}(r|q_i = 1) - \mathrm{PPR}^{\mathrm{Sink}}_{\{a\}}(r|q_i = 1). \tag{4.21}$$

Three other options, $\mathrm{BA}^{\mathrm{Del}}_{q \to r}(a)$, $\mathrm{BA}^{\mathrm{Sink}}_{r \to q}(a)$, and $\mathrm{BA}^{\mathrm{Del}}_{r \to q}(a)$ are defined analogously.

4.3.2.2 Proximity Measure

A proximity measure provides a way to compute node proximity with respect to a query node. As we discuss in Sections 2.5 and 3.6, PPR and HAR are two important proximity measures.

PPR (Personalized PageRank) [Hav02] measures node-to-node proximity by the limiting probability distribution of a random walker biased towards a set of query nodes.

HAR [LNY12] is a generalization of SALSA [LM01] to handle multi-relational data, which enables users to specify the relative importance of relations. HAR has been shown to outperform SALSA [LM01] and HITS [Kle99] in identifying relevant results to the query input. HAR computes hub score and authority score with respect to a query entity. While we considered both scores as a proximity measure, due to space constraints, we report only the result obtained with authority score as hub score was mostly outperformed by authority score.

Among the relevance models introduced above, MP and BA internally use PPR. We evaluate variants of these baselines using HAR as their proximity measure (except for ExpLOD, which is not a random walk-based method).

4.3.3 Results

Table 4.2 summarizes which axioms are satisfied by different approaches to measure attribute relevance. A checkmark indicates that the expected outcome of the axiom has been achieved by the corresponding method. J-RECS is the only one that satisfies all axioms. Other alternatives fail at least one of the axioms; among them, MP is the next best one, failing only (6) Community Awareness axiom. In Section 4.4, we show that J-RECS also leads to more accurate justifications than MP in experiments using real-world product graphs.

ExpLOD does not satisfy most of the axioms, mainly due to the fact that it can only consider direct edges between products and attributes, failing to propagate information over the graph.

In general, BASSET (BA) led to worse results than J-RECS and MP, failing (1) Popularity, (6) Community Awareness, and (7) Long Path Connectivity axioms in many cases,

Table 4.3: Statistics of real-world product graphs.

Name	Product	# Products	# Nodes	# Edges
MOVIE-PG	Movie	4,803	308,304	1,329,428
CITATION-PG	Paper	11,941	111,007	724,962
CITATION-100M-PG	Paper	2,094,396	7,426,773	100,000,000

indicating that good gateway nodes may not serve well as a justification. Also, while HAR achieved a reasonably good result (for instance, with MP-AND), relevance models could satisfy more axioms using PPR as a proximity measure.

4.4 Evaluation Using Real-World Data

In this section, we address the following questions.

Q1. Justification Quality. How well does J-RECS justify recommendations?
Q2. Scalability. How does J-RECS scale up with the increase of the input size?
Q3. Relevance-Diversity Trade-Off. How does increasing the weight for diversity affect the relevance of justifications?

Experimental settings are given in Section 4.7.1.

4.4.1 Datasets

We construct product graphs from public datasets on movies and publications. Table 4.3 shows the statistics of our datasets.

MOVIE-PG consists of movies and movie-related attributes, such as actors, directors, crews, casts, and movie keywords, extracted from the TMDb 5000 movie dataset[1]. We also added positive reviews to MOVIE-PG, which gave 10/10 rating to the movies. Reviews were retrieved from the IMDb website[2], instead of TMDb, since more reviews were available on IMDb. We extracted keywords from each review, by filtering out those words whose tf-idf score is below a threshold, and connected them with the reviews.

CITATION-PG is a product graph of papers and related attributes, such as authors, citations, publication venues, and fields of study, constructed from the citation network dataset v12 [TZY+08][3]. In CITATION-PG, we included papers published at KDD, SIGMOD, and ICML, and their attributes, while excluding papers that were cited less than 5 times. We also created CITATION-100M-PG that contains 10^8 edges for scalability evaluation, which is also constructed from the same citation network dataset, and includes all venues and papers.

[1]https://www.kaggle.com/tmdb/tmdb-movie-metadata
[2]https://www.imdb.com/
[3]https://www.aminer.org/citation

4.4.2 Baselines

Among the baselines used in Section 4.3, we use MP-AND and MP-OR [TF06], which satisfied most axioms among baselines, and ExpLOD [MNL+16], which is a representative non-random walk based method for justifying recommendations. Among them, we exclude BA and HAR (which was used as a proximity measure) as they were mostly outperformed by other alternatives. We also include PR [PBMW99], which estimates attribute relevance by its PageRank score in the product graph.

4.4.3 Q1. Justification Quality

We evaluate the quality of justifications in two ways: automatic evaluation and qualitative analysis.

4.4.3.1 Automatic Evaluation

For an automatic and objective evaluation, we consider the task of user preference retrieval.

User Preference Retrieval. Among several attributes of the recommended product r, we want those that reflect the user's preference better than others to receive higher relevance scores and be used as a justification. In our two datasets, the review and the paper written by a user clearly reflect the user's preference. Thus, among the reviews and papers associated with recommended product r, it is desirable for the review and paper written by the user to get higher scores than others.

For MOVIE-PG, we select a positive review written by the user who wrote at least 10 positive reviews. Note that the movies for which user u wrote positive reviews correspond to Q_u. Similarly, for CITATION-PG, we select a reference written by the user who published at least 15 papers. In total, we randomly select 50 reviews and 50 citations.

Performance Evaluation. We evaluate preference retrieval results using the mean reciprocal rank (MRR). Let $\mathcal{A}_R$ denote the chosen attributes of a specific type (e.g., reviews or cited papers), written by different users. Let p_i refer to the product of i-th attribute in $\mathcal{A}_R$ (e.g., the product for which the i-th review was written), and rank_i be the rank position of the i-th attribute among the corresponding attributes of product p_i, where ranks are determined by the estimated relevance score computed with respect to r and the products Q_u that received positive feedback by user u who created the i-th attribute (e.g., movies that received positive reviews by user u). MRR is defined by

$$\text{MRR} = \frac{1}{|\mathcal{A}_R|} \sum_{i=1}^{|\mathcal{A}_R|} \frac{1}{\text{rank}_i}, \tag{4.22}$$

and higher values are better.

Results. Figure 4.1 shows the MMR on two datasets. J-RECS achieved the best MMR, which is up to 20.7% higher than the second best result achieved by MP-AND. While MP-OR outperformed MP-AND on CITATION-PG by a small margin, results show that

MP-OR's performance is more sensitive to the dataset than MP-AND. PR's performance is worse than J-RECS as it does not consider Q_u and r in measuring attribute relevance. Since each review applies to only one product, ExpLOD ends up assigning identical scores to all reviews, making it inapplicable to be used to retrieve user preferences on MOVIE-PG. On CITATION-PG, ExpLOD obtains even lower MMR than PR, which is due to the fact that ExpLOD computes relevance score based only on the direct connection between products and attributes, failing to propagate user preference over a graph.

Note that since we ignore the relevance of other reviews and citations, which is unknown to us, these results are a lower bound of the true MMR. As MMR considers the rank of the first relevant entity, the true MMR would be higher than the current result if there is another relevant attribute, ranked higher than the user's review and paper.

4.4.3.2 Qualitative Analysis

We present two case studies where we compare the results obtained with J-RECS and MP-AND on CITATION-PG. To see how the parameter ρ in (4.9) affects J-RECS, we report two results for J-RECS using $\rho = 0.5$ and $\rho = 0.9$.

Case 1. J-RECS and MP-AND are given the paper entitled "Rubik: Knowledge Guided Tensor Factorization and Completion for Health Data Analytics" [WCG+15] as a recommended item r, and the set Q_u with ten papers on matrix and tensor factorizations. Table 4.4 shows top-15 papers cited by the recommended paper, ordered by the relevance computed with J-RECS (4.9) and MP-AND (4.19). The papers in blue font deal with electronic health record (EHR) analysis, and those highlighted in cyan background color are two highly relevant papers that employ tensor factorization for EHR analysis. Among the citations, these highlighted papers are particularly relevant justifications, since they cut across two topics central to the recommended paper and the papers in Q_u, namely, EHR analysis and tensor factorization. In Table 4.4, these two highly relevant papers belong to the top-5 citations in both results of J-RECS, while they are ranked at 6th and 11th places in the result of MP-AND. Further, since EHR analysis is a major topic of the recommended paper, with $\rho = 0.9$, J-RECS gives more weight to EHR analysis than with $\rho = 0.5$, which boosts the ranking of the papers on EHR analysis.

Case 2. J-RECS and MP-AND are given the paper entitled "Towards Parameter-Free Data Mining" [KLR04] as a recommended item r, and the set Q_u with ten papers on time series analysis. Table 4.5 shows top-15 papers cited by the recommended paper, ordered by the relevance computed with J-RECS (4.9) and MP-AND (4.19). The citations in blue font are the papers relevant to the MDL principle. Among the cited publications, those on MDL are highly relevant justifications as MDL principle is central to the main idea of the recommended paper. At the same time, since Q_u contains papers on time series analysis, references on time series are relevant. In Table 4.5, J-RECS retrieves four papers on MDL (ranked at 2nd, 7th, 11th, and 13th places with $\rho = 0.9$), and eleven papers on time series. On the other hand, MP-AND retrieves only two papers on MDL (ranked at 7th and 14th places). Also, in this case, increasing ρ to 0.9 led to a more drastic change than in the previous case, boosting the ranking of the papers on MDL.

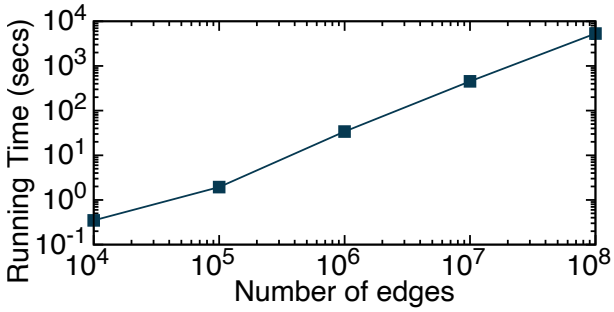


Figure 4.5: J-RECS exhibits near-linear scalability.

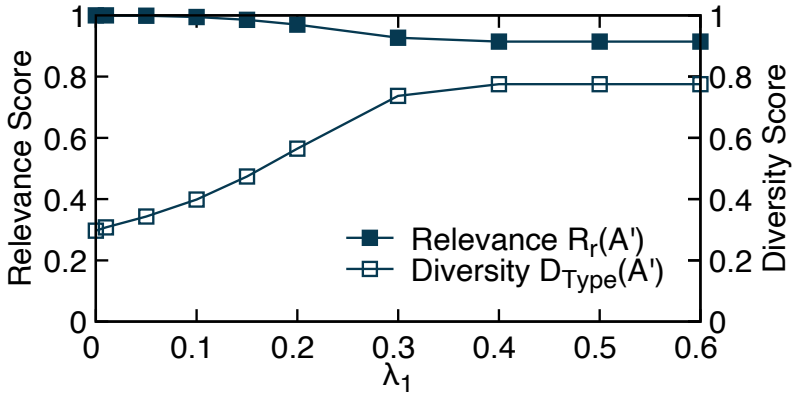


Figure 4.6: Relevance-diversity trade-off on MOVIE-PG. Varying λ_1 affects $R_r(\mathcal{A}')$ and $D_{\text{Type}}(\mathcal{A}')$ of the selected attributes $\mathcal{A}'$.

Overall, in these case studies, J-RECS produces qualitatively better results than MP-AND, which are more balanced in terms of the relevance to the recommendation, user preferences, and the diversity of paper topics.

4.4.4 Q2. Scalability

To evaluate the scalability of J-RECS, we created increasingly larger subgraphs of CITATION-100M-PG, with each subgraph being $10\times$ larger than the previous one. Figure 4.5 reports the running time of J-RECS on these product graphs of varying sizes, where the running time was averaged over three simulated users with at least ten products in his $\mathcal{Q}_u$. The results show that J-RECS achieves near-linear scalability, successfully scaling up to the largest graph with 10^8 edges.

4.4.5 Q3. Relevance-Diversity Trade-Off

We measure how varying λ_1, the weight for the attribute type diversity, affects the relevance score $R_r(\mathcal{A}')$ and the diversity score $D_{\text{Type}}(\mathcal{A}')$ of the selected attributes $\mathcal{A}'$. Specifically, we randomly selected 50 users from MOVIE-PG, who rated at least 10 movies, and generated 30 justifications varying λ_1 from 0 to 0.6. In Figure 4.6, we report relevance and diversity scores averaged over all users. Figure 4.6 shows that while there is a trade-off between relevance and diversity, it is possible to attain high relevance and diversity. As λ_1 is increased from 0 to 0.3, $D_{\text{Type}}(\mathcal{A}')$ increases by 148%, while $R_r(\mathcal{A}')$

decreases by 7%. Results indicate that setting λ_1 to an appropriate value can be beneficial in providing diverse and relevant justifications to the users.

4.5 Related Work

Explainable Recommendation. Methods for explainable recommendation can be grouped into embedded and post-hoc approaches. Embedded approaches [WHF+18, CZH+17, DQW+14, CCX+19, ZLZ+14] aim to develop interpretable models, such that explanations for the model decision can be naturally provided. While embedded methods have high model explainability, different explanation techniques need to be developed for different types of recommendation methods. Our framework is model-agnostic and can be applied to different recommenders flexibly. We refer the reader to [ZC20] for an in-depth review of embedded methods.

In post-hoc approaches, explanations and recommendations are generated from separate models. Post-hoc explanations are often item-based (e.g., "Customers who bought this item also bought" [TM15]), neighbor-based (e.g., "Your neighbors' rating for this item is" [HKR00]), and content-based (e.g., keywords [BM05], features [Tin07], tags [VSR09], or reviews [DLZ18, WCY+18]). Since these methods typically select an explanation based on one of manually defined templates or generate explanations using one type of data, the diversity of their explanations are limited by the form of such templates or the type of input data. On the other hand, J-RECS is a unified framework that can work with multiple types of data, and its diversity increases as we provide more data to the framework. ExpLOD [MNL+16] provides more diverse explanations than earlier methods by using linked open data cloud in a graph-based framework. However, ExpLOD and others such as [VSR09] produce justifications only using the items in the user profile, ignoring other items and their attributes relevant to the recommendation and the user profile. By using random walk-based node proximity, J-RECS utilizes both the user profile and other data relevant to it.

Node Importance. As we discuss in Sections 2.5 and 3.6, PageRank (PR) [PBMW99] measures node importance by considering the limiting probability of a random surfer that travels over a graph following any out-going edge with uniform probability. The original PR does not depend on the query, and personalized PR (PPR) [Hav02] followed PR to estimate query-dependent node importance. Random walk with restart (RWR) [TFP08, JPSK17] can be seen as a special case of PPR that considers one query node. HITS [Kle99] first retrieves a focused subgraph with respect to the search query, and computes hub and authority scores for each node in the focused subgraph. SALSA [LM01] can be considered as an improvement of HITS, which also computes hub and authority scores like HITS, while less susceptible to the tightly knit community (TKC) effect than HITS. HAR [LNY12] is a generalization of SALSA that deals with multi-relation data, and computes hub and authority scores for objects and relevance scores for relations, with respect to a query input. GENI (Chapter 2) and MultiImport (Chapter 3) are semi-supervised techniques to estimate node importance by considering both the graph structure and real-world signals of node popularity. Among the above methods, query-

sensitive ones can be used to measure node-to-node proximity, which have also been used to identify a subset of nodes or a subgraph, which have strong connections to the query nodes [FMT04, TF06] and are important in connecting source and target nodes [TPF+10, TPF+12]. In this work, we consider the effectiveness of these approaches for the task of recommendation justification, and use the most effective ones to define the relevance score, which best satisfy the axioms of good justifications.

4.6 Conclusion

In this chapter, we present a graph-based formulation of the problem of recommendation justification, and develop J-RECS, a unified model-agnostic framework which can produce concise, diverse, and personalized justifications in a principled manner, based on various types of product and user data. We show the effectiveness and efficiency of J-RECS in an evaluation using axioms and real-world data. In this chapter, we propose preference retrieval as one way of evaluating the justification quality. Developing additional automatic and objective evaluation metrics that can measure the quality of justifications from different perspectives will also be an important direction for future research on explainable recommendations.

4.7 Appendix

4.7.1 Experimental Settings

Machine. We ran experiments on a machine with 32 Intel Xeon CPU E7-8837 cores at 2.67GHz, and 1 TB of memory.

Parameters. For PPR [Hav02], we set the damping factor to 0.85. For HAR [LNY12], we set the weighting parameters α, β, and γ to 0.15. For J-RECS, we set $\rho = 0.5$ and budget B to 15, unless otherwise stated. For ExpLOD [MNL+16], we followed the settings used in [MNL+16], where the two weighting factors α and β were set to 0.5.

Table 4.4: **J-RECS works:** J-RECS produces qualitatively better results than MP-AND. The table shows the top-15 references cited by the paper "Rubik: Knowledge Guided Tensor Factorization and Completion for Health Data Analytics" [WCG+15], ordered by the relevance computed with J-RECS (with ρ set to 0.5 and 0.9) and MP-AND. Papers on electronic health record analysis are highlighted in blue font, and the two highly relevant papers that employ tensor factorization for electronic health record analysis are further highlighted in cyan background color.

J-RECS ($\rho = 0.5$)	**J-RECS** ($\rho = 0.9$)	**MP-AND** [TF06]
Positive tensor factorization	Tensor decompositions and applications	Tensor decompositions and applications
Tensor decompositions and applications	Positive tensor factorization	Distributed optimization and statistical learning via the alternating direction method of multipliers
Marble: high-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization	Marble: high-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization	Positive tensor factorization
On tensors, sparsity, and nonnegative factorizations	Limestone: high-throughput candidate phenotype generation via tensor factorization	Scalable tensor factorizations for incomplete data
Limestone: high-throughput candidate phenotype generation via tensor factorization	On tensors, sparsity, and nonnegative factorizations	Learning with tensors: a framework based on convex optimization and spectral regularization
Distributed optimization and statistical learning via the alternating direction method of multipliers	Scalable tensor factorizations for incomplete data	Marble: high-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization
Scalable tensor factorizations for incomplete data	Distributed optimization and statistical learning via the alternating direction method of multipliers	A block coordinate descent method for regularized multiconvex optimization with applications to nonnegative tensor factorization and completion
Tensor completion for estimating missing values in visual data	Tensor completion for estimating missing values in visual data	On tensors, sparsity, and nonnegative factorizations
Network discovery via constrained tensor analysis of fMRI data	Network discovery via constrained tensor analysis of fMRI data	Tensor completion for estimating missing values in visual data
Learning with tensors: a framework based on convex optimization and spectral regularization	Next-generation phenotyping of electronic health records	Network discovery via constrained tensor analysis of fMRI data
Next-generation phenotyping of electronic health records	Learning with tensors: a framework based on convex optimization and spectral regularization	Limestone: high-throughput candidate phenotype generation via tensor factorization
Square deal: lower bounds and improved relaxations for tensor recovery	Square deal: lower bounds and improved relaxations for tensor recovery	Next-generation phenotyping of electronic health records
FlexiFaCT: scalable flexible factorization of coupled tensors on Hadoop	FlexiFaCT: scalable flexible factorization of coupled tensors on Hadoop	Square deal: lower bounds and improved relaxations for tensor recovery
All-at-once optimization for coupled matrix and tensor factorizations	All-at-once optimization for coupled matrix and tensor factorizations	Convex tensor decomposition via structured Schatten norm regularization
A block coordinate descent method for regularized multiconvex optimization with applications to nonnegative tensor factorization and completion	Convex tensor decomposition via structured Schatten norm regularization	A new convex relaxation for tensor completion

Table 4.5: **J-RECS works:** J-RECS produces qualitatively better results than MP-AND. The table shows the top-15 references cited by the paper "Towards Parameter-Free Data Mining" [KLR04], ordered by the relevance computed with J-RECS (with ρ set to 0.5 and 0.9) and MP-AND. Papers on the minimum description length principle are highlighted in blue font.

J-RECS ($\rho = 0.5$)	**J-RECS** ($\rho = 0.9$)	**MP-AND** [TF06]
On the need for time series data mining benchmarks: a survey and empirical demonstration	On the need for time series data mining benchmarks: a survey and empirical demonstration	On the need for time series data mining benchmarks: a survey and empirical demonstration
Making time-series classification more accurate using learned constraints	An introduction to Kolmogorov complexity and its applications	Clustering of time series subsequences is meaningless: implications for previous and future research
Distance measures for effective clustering of ARIMA time-series	Making time-series classification more accurate using learned constraints	A symbolic representation of time series, with implications for streaming algorithms
Deformable Markov model templates for time-series pattern matching	Distance measures for effective clustering of ARIMA time-series	Making time-series classification more accurate using learned constraints
TSA-tree: a wavelet-based approach to improve the efficiency of multi-level surprise and trend queries on time-series data	Deformable Markov model templates for time-series pattern matching	Distance measures for effective clustering of ARIMA time-series
Mining the stock market: which measure is best?	TSA-tree: a wavelet-based approach to improve the efficiency of multi-level surprise and trend queries on time-series data	Mining the stock market: which measure is best?
Supporting content-based searches on time series via approximation	Modeling by shortest data description	Modeling by shortest data description
Clustering of time series subsequences is meaningless: implications for previous and future research	Mining the stock market: which measure is best?	Deformable Markov model templates for time-series pattern matching
An introduction to Kolmogorov complexity and its applications	A symbolic representation of time series, with implications for streaming algorithms	Indexing multi-dimensional time-series with support for multiple distance measures
A symbolic representation of time series, with implications for streaming algorithms	Clustering of time series subsequences is meaningless: implications for previous and future research	FastMap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets
Indexing multi-dimensional time-series with support for multiple distance measures	Inferring decision trees using the minimum description length principle	TSA-tree: a wavelet-based approach to improve the efficiency of multi-level surprise and trend queries on time-series data
Modeling by shortest data description	Supporting content-based searches on time series via approximation	Online novelty detection on temporal sequences
Inferring decision trees using the minimum description length principle	The similarity metric	Supporting content-based searches on time series via approximation
The similarity metric	Indexing multi-dimensional time-series with support for multiple distance measures	Inferring decision trees using the minimum description length principle
Online novelty detection on temporal sequences	Online novelty detection on temporal sequences	Graph-based anomaly detection

Chapter based on work published in ICDE 2017 [POK17] and the VLDB Journal [POK19].

Chapter 5

Fast and Scalable Distributed Boolean Tensor Factorization

How can we analyze tensors that are composed of 0's and 1's? How can we efficiently analyze such Boolean tensors with millions or even billions of entries? Boolean tensors often represent relationship, membership, or occurrences of events such as subject-relation-object tuples in knowledge base data (e.g., 'Seoul'-'is the capital of'-'South Korea'). Boolean tensor factorization (BTF) is a useful tool for analyzing binary tensors to discover latent factors from them. Furthermore, BTF is known to produce more interpretable and sparser results than normal factorization methods. Although several BTF algorithms exist, they do not scale up for large-scale Boolean tensors. In this chapter, we propose DBTF, a distributed method for Boolean CP (DBTF-CP) and Tucker (DBTF-TK) factorizations running on the Apache Spark framework. By distributed data generation with minimal network transfer, exploiting the characteristics of Boolean operations, and with careful partitioning, DBTF successfully tackles the high computational costs and minimizes the intermediate data. Experimental results show that DBTF-CP decomposes up to 16^3–$32^3\times$ larger tensors than existing methods in 82–180$\times$ less time, and DBTF-TK decomposes up to 8^3–$16^3\times$ larger tensors than existing methods in 86–129$\times$ less time. Furthermore, both DBTF-CP and DBTF-TK exhibit near-linear scalability in terms of tensor dimensionality, density, rank, and machines.

5.1 Introduction

How can we analyze tensors that are composed of 0's and 1's? How can we efficiently analyze such Boolean tensors that have millions or even billions of entries? Many real-world data can be represented as tensors, or multi-dimensional arrays. Among them, many are composed of only 0's and 1's. Those tensors often represent relationship, membership, or occurrences of events. Examples of such data include subject-relation-

object tuples in knowledge base data (e.g., 'Seoul'-'is the capital of'-'South Korea'), source IP-destination IP-port number-timestamp in network intrusion logs, and user1 ID-user2 ID-timestamp in friendship network data. Tensor factorizations are widely-used tools for analyzing tensors. CANDECOMP/PARAFAC (CP) and Tucker are two major tensor factorization methods [KB09]. These methods decompose a tensor into a sum of rank-1 tensors, from which we can find the latent structure of the data. Tensor factorization methods can be classified according to the constraint placed on the resulting rank-1 tensors [EM13b]. The unconstrained form allows entries in the rank-1 tensors to be arbitrary real numbers, where we find linear relationships between latent factors; when a non-negativity constraint is imposed on the entries, the resulting factors reveal parts-of-whole relationships.

What we focus on in this chapter is yet another approach with Boolean constraints, named Boolean tensor factorization (BTF) [Mie11], that has many interesting applications including latent concept discovery, clustering, recommendation, link prediction, and synonym finding. For example, low-rank BTF yields factor matrices whose columns correspond to latent concepts underlying the data, and applying a Boolean Tucker factorization to subject-predicate-object triples can find synonyms and uncover facts from the data [EM13a]. BTF requires that the input tensor, all factor matrices, and a core tensor are binary. Furthermore, BTF uses Boolean sum instead of normal addition, which means $1 + 1 = 1$ in BTF. When the data is inherently binary, BTF is an appealing choice as it can reveal Boolean structures and relationships underlying the binary tensor that are hard to be found by other factorizations. Also, BTF is known to produce more interpretable and sparser results than the unconstrained and the non-negativity constrained counterparts, though at the expense of increased computational complexity [Mie11, MM15].

While several algorithms have been developed for BTF [Mie11, EM13b, BGV13, LVMDBR99], they are not fast and scalable enough for large-scale tensors that have become widespread. For example, consider DBLP and NELL datasets, which are two different types of real-world tensors consisting of 1.3M to 77M non-zeros. In our experiments on these tensors, all of the state-of-the-art BTF methods get terminated due to out-of-memory errors, or

Table 5.1: Comparison of our proposed DBTF and existing methods for Boolean (a) CP and (b) Tucker factorizations in terms of whether a method is parallel (Par.) and distributed (Dist.). DBTF is the only approach that is parallel and distributed.

(a) Boolean CP Factorization

Method	Par.	Dist.
Walk'n'Merge [EM13b]	Yes	No
BCP_ALS [Mie11]	No	No
DBTF-CP	**Yes**	**Yes**

(b) Boolean Tucker Factorization

Method	Par.	Dist.
Walk'n'Merge [EM13b]	Yes	No
BTucker_ALS [Mie11]	No	No
DBTF-TK	**Yes**	**Yes**

failed at processing them within a reasonable amount of time. Even in the case where existing approaches could be applied, their performance is not enough for many practical applications. In order for BTF to be used for the analysis of large-scale tensors in practical settings, it is of great importance to overcome these limitations. In summary, the major challenges that need to be tackled for fast and scalable BTF are (1) how to minimize the computational costs involved with updating Boolean factor matrices, and (2) how to minimize the intermediate data that are generated in the process of factorization. Existing methods fail to solve both of these challenges.

In this chapter, we propose DBTF (Distributed Boolean Tensor Factorization), a distributed method for Boolean CP and Tucker factorizations running on the Apache Spark framework [ZCD+12]. DBTF tackles the high computational cost by reducing the operations involved with BTF in an efficient greedy algorithm, while minimizing the generation and shuffling of intermediate data. Also, DBTF exploits the characteristics of Boolean operations in solving both of the above problems. Due to the effective algorithm designed carefully with these ideas, DBTF achieves higher efficiency and scalability compared to existing methods. Table 5.1 shows a comparison of DBTF and existing methods in terms of whether a method is parallel and distributed. Note that DBTF is the only approach that is parallel and distributed.

The main contributions of this chapter are as follows:

- **Algorithm.** We propose DBTF, a distributed method for Boolean CP (DBTF-CP) and Tucker (DBTF-TK) factorizations, which is designed to scale up to large tensors by minimizing intermediate data and the number of operations for BTF, and carefully partitioning the workload.
- **Theory.** We provide an analysis of the proposed DBTF-CP and DBTF-TK in terms of time complexity, memory requirement, and the amount of shuffled data.
- **Experiment.** We present extensive empirical evidences for the scalability and performance of DBTF. We show that the proposed Boolean CP factorization method decomposes up to 16^3–$32^3\times$ larger tensors than existing methods in 82–180$\times$ less time, and the proposed Boolean Tucker factorization method decomposes up to 8^3–$16^3\times$ larger tensors than existing methods in 86–129$\times$ less time. We also show that DBTF successfully decomposes real-world tensors, such as DBLP and NELL, that cannot be processed with the state-of-the-art BTF methods.

The code of our method and the datasets used in this chapter are available at https://www.cs.cmu.edu/~namyongp/dbtf/.

The rest of this chapter is organized as follows. We present the preliminaries of CP and Tucker factorizations in normal and Boolean settings in Section 5.2. Then, we discuss related works in Section 5.3, and describe our proposed method for fast and scalable Boolean CP and Tucker factorization in Section 5.4. After presenting experimental results in Section 5.5, we conclude in Section 5.6.

5.2 Preliminaries

In this section, we provide the definition of Boolean arithmetic and present the notations and operations used for tensor decomposition. Next, we give the definitions of normal CP and Tucker decompositions, and those of Boolean CP and Tucker decompositions. After that, we introduce approaches for computing Boolean CP and Tucker decompositions. Symbols used in this chapter are summarized in Table 5.2.

5.2.1 Boolean Arithmetic

Given binary data, all operations involved with Boolean tensor factorization operate with Boolean arithmetic in which addition (Boolean OR which is denoted by $\vee$) and multiplication (Boolean AND which is denoted by $\wedge$) between two variables are defined as:

x	y	$x \wedge y$	$x \vee y$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

5.2.2 Notation

We denote tensors by boldface Euler script letters (e.g., $\mathcal{X}$), matrices by boldface capitals (e.g., $\mathbf{A}$), vectors by boldface lowercase letters (e.g., $\mathbf{a}$), and scalars by lowercase letters (e.g., a).

Tensors and Matrices. Tensor is a multi-dimensional array. The dimension of a tensor is also referred to as *mode*, *order*, or *way*. A matrix $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2}$ is a tensor of order two. A tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is an N-mode or N-way tensor. The $(i_1, i_2, \ldots, i_N)$-th entry of a tensor $\mathcal{X}$ is denoted by $x_{i_1 i_2 \ldots i_N}$. A colon in the subscript indicates taking all elements of that mode. For example, given a matrix $\mathbf{A}$, $\mathbf{a}_{:j}$ denotes the j-th column, and $\mathbf{a}_{i:}$ denotes the i-th row. The j-th column of $\mathbf{A}$, $\mathbf{a}_{:j}$, is also denoted more concisely as $\mathbf{a}_j$. A colon between two numbers in the subscript denotes taking all such elements whose indices in that mode lie between the given numbers. For instance, $\mathbf{A}_{(1:5)j}$ indicates the first five elements of $\mathbf{a}_j$. For a three-way tensor $\mathcal{X}$, $\mathbf{x}_{:jk}$, $\mathbf{x}_{i:k}$, and $\mathbf{x}_{ij:}$ are called column (mode-1), row (mode-2), and tube (mode-3) fibers, respectively. $|\mathcal{X}|$ denotes the number of non-zero elements in a tensor $\mathcal{X}$; $\|\mathcal{X}\|$ denotes the Frobenius norm of a tensor $\mathcal{X}$ and is defined as $\sqrt{\sum_{i,j,k} x_{ijk}^2}$.

Tensor Matricization/Unfolding. The mode-n matricization (or unfolding) of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, denoted by $\mathbf{X}_{(n)}$, is the process of unfolding $\mathcal{X}$ into a matrix by re-arranging its mode-n fibers to be the columns of the resulting matrix. For instance, a

Table 5.2: Table of symbols.

Symbol	Definition
$\mathcal{X}$	tensor (Euler script, bold letter)
$\mathbf{A}$	matrix (in uppercase, bold letter)
$\mathbf{a}$	column vector (lowercase, bold letter)
a	scalar (lowercase, italic letter)
R	rank (number of components)
$\mathcal{G}$	core tensor ($\in \mathbb{R}^{R_1 \times R_2 \times R_3}$)
$\mathbf{X}_{(n)}$	mode-n matricization of a tensor $\mathcal{X}$
$\lvert\mathcal{X}\rvert$	number of non-zeros in the tensor $\mathcal{X}$
$\lVert\mathcal{X}\rVert$	Frobenius norm of the tensor $\mathcal{X}$
$\mathbf{A}^\top$	transpose of matrix $\mathbf{A}$
$\circ$	outer product
$\otimes$	Kronecker product
$\odot$	Khatri-Rao product
$\mathbb{B}$	set of binary numbers, i.e., $\{0, 1\}$
$\vee$	Boolean sum of two binary tensors
$\bigvee$	Boolean summation of a sequence of binary tensors
$\boxtimes$	Boolean matrix product
I, J, K	dimensions of each mode of an input tensor $\mathcal{X}$
R_1, R_2, R_3	dimensions of each mode of a core tensor $\mathcal{G}$

three-way tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and its matricizations are mapped as follows:

$$\begin{aligned} x_{ijk} &\rightarrow [\mathbf{X}_{(1)}]_{ic} \text{ where } c = j + (k-1)J \\ x_{ijk} &\rightarrow [\mathbf{X}_{(2)}]_{jc} \text{ where } c = i + (k-1)I \\ x_{ijk} &\rightarrow [\mathbf{X}_{(3)}]_{kc} \text{ where } c = i + (j-1)I. \end{aligned} \tag{5.1}$$

Outer Product and Rank-1 Tensor. We use $\circ$ to denote the vector outer product. The three-way outer product of vectors $\mathbf{a} \in \mathbb{R}^I$, $\mathbf{b} \in \mathbb{R}^J$, and $\mathbf{c} \in \mathbb{R}^K$ is a tensor $\mathcal{X} = \mathbf{a} \circ \mathbf{b} \circ \mathbf{c} \in \mathbb{R}^{I \times J \times K}$ whose element (i, j, k) is defined as $(\mathbf{a} \circ \mathbf{b} \circ \mathbf{c})_{ijk} = a_i b_j c_k$. A three-way tensor $\mathcal{X}$ is rank-1 if it can be expressed as an outer product of three vectors.

Kronecker Product. The Kronecker product of matrices $\mathbf{A} \in \mathbb{R}^{I_1 \times J_1}$ and $\mathbf{B} \in \mathbb{R}^{I_2 \times J_2}$ produces a matrix of size $I_1 I_2$-by-$J_1 J_2$, which is defined as:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1J_1}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2J_1}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I_1 1}\mathbf{B} & a_{I_1 2}\mathbf{B} & \cdots & a_{I_1 J_1}\mathbf{B} \end{bmatrix}. \tag{5.2}$$

The Kronecker product of matrices $\mathbf{A} \in \mathbb{R}^{I_1 \times J_1}$ and $\mathbf{B} \in \mathbb{R}^{I_2 \times J_2}$ can also be expressed by vector-matrix Kronecker products as follows:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} \mathbf{a}_{1:} \otimes \mathbf{B} \\ \mathbf{a}_{2:} \otimes \mathbf{B} \\ \vdots \\ \mathbf{a}_{I_1:} \otimes \mathbf{B} \end{bmatrix}. \tag{5.3}$$

Khatri-Rao Product. The Khatri-Rao product (or column-wise Kronecker product) of matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$ and $\mathbf{B} \in \mathbb{R}^{J \times R}$ produces a matrix of size IJ-by-R, and is defined as:

$$\mathbf{A} \odot \mathbf{B} = [\mathbf{a}_{:1} \otimes \mathbf{b}_{:1} \; \mathbf{a}_{:2} \otimes \mathbf{b}_{:2} \; \ldots \; \mathbf{a}_{:R} \otimes \mathbf{b}_{:R}]. \tag{5.4}$$

Given matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$ and $\mathbf{B} \in \mathbb{R}^{J \times R}$, the Khatri-Rao product $\mathbf{A} \odot \mathbf{B}$ can also be expressed by vector-matrix Khatri-Rao products as follows:

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} \mathbf{a}_{1:} \odot \mathbf{B} \\ \mathbf{a}_{2:} \odot \mathbf{B} \\ \vdots \\ \mathbf{a}_{I_1:} \odot \mathbf{B} \end{bmatrix}. \tag{5.5}$$

Set of Binary Numbers. We use $\mathbb{B}$ to denote the set of binary numbers, that is, $\{0, 1\}$.

Boolean Summation. We use $\bigvee$ to denote the Boolean summation, in which a sequence of Boolean tensors or matrices is summed. The Boolean sum ($\vee$) of two binary tensors $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$ and $\mathcal{Y} \in \mathbb{B}^{I \times J \times K}$ is defined by:

$$(\mathcal{X} \vee \mathcal{Y})_{ijk} = x_{ijk} \vee y_{ijk}. \tag{5.6}$$

The Boolean sum of two binary matrices is defined analogously.

Boolean Matrix Product. The Boolean product of two binary matrices $\mathbf{A} \in \mathbb{B}^{I \times R}$ and $\mathbf{B} \in \mathbb{B}^{R \times J}$ is defined as:

$$(\mathbf{A} \boxtimes \mathbf{B})_{ij} = \bigvee_{k=1}^{R} a_{ik} b_{kj}. \tag{5.7}$$

5.2.3 Tensor Rank and Tensor Decompositions

5.2.3.1 Normal Tensor Rank and Tensor Decompositions

With the above notations, we first give the definitions of normal tensor rank, and normal CP and Tucker decompositions.

Definition 4.: (Tensor rank) The rank of a three-way tensor $\mathcal{X}$ is the smallest integer R such that there exist R rank-1 tensors whose sum is equal to the tensor $\mathcal{X}$, i.e.,

$$\mathcal{X} = \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r. \tag{5.8}$$

Definition 5.: (CP decomposition) Given a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and a rank R, find factor matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, and $\mathbf{C} \in \mathbb{R}^{K \times R}$ such that they minimize

$$\left\| \mathcal{X} - \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r \right\|. \tag{5.9}$$

CP decomposition can be expressed in a matricized form as follows [KB09]:

$$\begin{aligned} \mathbf{X}_{(1)} &\approx \mathbf{A}(\mathbf{C} \odot \mathbf{B})^\top \\ \mathbf{X}_{(2)} &\approx \mathbf{B}(\mathbf{C} \odot \mathbf{A})^\top \\ \mathbf{X}_{(3)} &\approx \mathbf{C}(\mathbf{B} \odot \mathbf{A})^\top. \end{aligned} \tag{5.10}$$

Definition 6.: (Tucker decomposition) Given a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and the dimensions of a core tensor R_1, R_2, and R_3, find factor matrices $\mathbf{A} \in \mathbb{R}^{I \times R_1}$, $\mathbf{B} \in \mathbb{R}^{J \times R_2}$, $\mathbf{C} \in \mathbb{R}^{K \times R_3}$, and a core tensor $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times R_3}$ such that they minimize

$$\left\| \mathcal{X} - \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \sum_{r_3=1}^{R_3} g_{r_1 r_2 r_3} \mathbf{a}_{r_1} \circ \mathbf{b}_{r_2} \circ \mathbf{c}_{r_3} \right\|. \tag{5.11}$$

Tucker decomposition can be expressed in a matricized form as follows [KB09]:

$$\begin{aligned} \mathbf{X}_{(1)} &\approx \mathbf{A}\mathbf{G}_{(1)}(\mathbf{C} \otimes \mathbf{B})^\top \\ \mathbf{X}_{(2)} &\approx \mathbf{B}\mathbf{G}_{(2)}(\mathbf{C} \otimes \mathbf{A})^\top \\ \mathbf{X}_{(3)} &\approx \mathbf{C}\mathbf{G}_{(3)}(\mathbf{B} \otimes \mathbf{A})^\top. \end{aligned} \tag{5.12}$$

5.2.3.2 Boolean Tensor Rank and Tensor Decompositions

We now give the definitions of Boolean tensor rank, and Boolean CP and Tucker decompositions. The definitions of Boolean tensor rank and Boolean tensor decompositions differ from their normal counterparts in the following two respects: 1) the tensor and factor matrices are binary; 2) Boolean sum is used where $1 + 1$ is defined to be 1.

Definition 7.: (Boolean tensor rank) The Boolean rank of a three-way binary tensor $\mathcal{X}$ is the smallest integer R such that there exist R rank-1 binary tensors whose Boolean summation is equal to the tensor $\mathcal{X}$, i.e.,

$$\mathcal{X} = \bigvee_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r. \tag{5.13}$$

Definition 8.: (Boolean CP decomposition) Given a binary tensor $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$ and a rank R, find binary factor matrices $\mathbf{A} \in \mathbb{B}^{I \times R}$, $\mathbf{B} \in \mathbb{B}^{J \times R}$, and $\mathbf{C} \in \mathbb{B}^{K \times R}$ such that they minimize

$$\left| \mathcal{X} - \bigvee_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r \right|. \tag{5.14}$$

By replacing the normal matrix product in Equation (5.10) with the Boolean matrix product, Boolean CP decomposition can be expressed in a matricized form as follows:

$$\begin{aligned} \mathbf{X}_{(1)} &\approx \mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top \\ \mathbf{X}_{(2)} &\approx \mathbf{B} \boxtimes (\mathbf{C} \odot \mathbf{A})^\top \\ \mathbf{X}_{(3)} &\approx \mathbf{C} \boxtimes (\mathbf{B} \odot \mathbf{A})^\top. \end{aligned} \tag{5.15}$$

Figure 5.1 illustrates the rank-R Boolean CP decomposition of a three-way tensor $\mathcal{X}$.

Definition 9.: (Boolean Tucker decomposition) Given a binary tensor $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$ and the dimensions of a core tensor R_1, R_2, and R_3, find binary factor matrices $\mathbf{A} \in \mathbb{B}^{I \times R_1}$, $\mathbf{B} \in \mathbb{B}^{J \times R_2}$, $\mathbf{C} \in \mathbb{B}^{K \times R_3}$, and a binary core tensor $\mathcal{G} \in \mathbb{B}^{R_1 \times R_2 \times R_3}$ such that they minimize

$$\left| \mathcal{X} - \bigvee_{r_1=1}^{R_1} \bigvee_{r_2=1}^{R_2} \bigvee_{r_3=1}^{R_3} g_{r_1 r_2 r_3} \mathbf{a}_{r_1} \circ \mathbf{b}_{r_2} \circ \mathbf{c}_{r_3} \right|. \tag{5.16}$$

By using Boolean matrix product in place of the normal matrix product in Equation (5.12), Boolean Tucker decomposition can be expressed in a matricized form as follows:

$$\begin{aligned} \mathbf{X}_{(1)} &\approx \mathbf{A} \boxtimes \mathbf{G}_{(1)} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top \\ \mathbf{X}_{(2)} &\approx \mathbf{B} \boxtimes \mathbf{G}_{(2)} \boxtimes (\mathbf{C} \otimes \mathbf{A})^\top \\ \mathbf{X}_{(3)} &\approx \mathbf{C} \boxtimes \mathbf{G}_{(3)} \boxtimes (\mathbf{B} \otimes \mathbf{A})^\top. \end{aligned} \tag{5.17}$$

Figure 5.2 illustrates the rank-R Boolean Tucker decomposition of a three-way tensor $\mathcal{X}$.

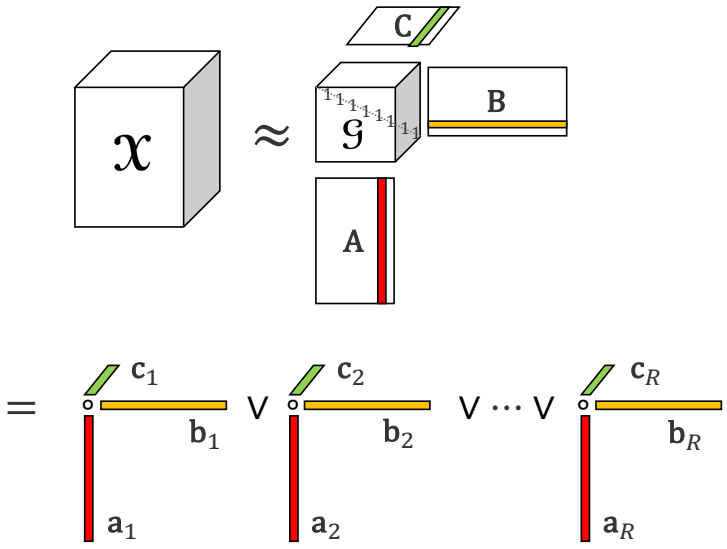


Figure 5.1: Rank-R Boolean CP decomposition of a three-way tensor $\mathcal{X}$. $\mathcal{X}$ is decomposed into three binary factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$.

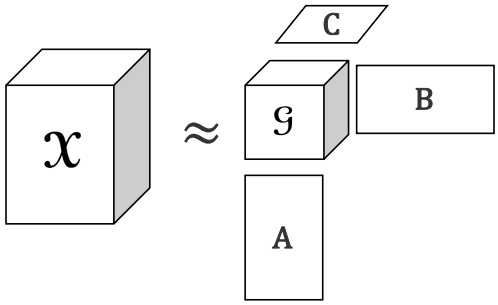


Figure 5.2: Rank-R Boolean Tucker decomposition of a three-way tensor $\mathcal{X}$. $\mathcal{X}$ is decomposed into a binary core tensor $\mathcal{G}$, and three binary factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$.

Algorithm 5.1: Boolean CP Decomposition Framework

Input: A three-way binary tensor $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$, rank R, and the maximum number of iterations T.
Output: Binary factor matrices $\mathbf{A} \in \mathbb{B}^{I \times R}$, $\mathbf{B} \in \mathbb{B}^{J \times R}$, and $\mathbf{C} \in \mathbb{B}^{K \times R}$.

1 initialize factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$
2 **for** $t \leftarrow 1..T$ **do**
3 update $\mathbf{A}$ such that $|\mathbf{X}_{(1)} - \mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top|$ is minimized
4 update $\mathbf{B}$ such that $|\mathbf{X}_{(2)} - \mathbf{B} \boxtimes (\mathbf{C} \odot \mathbf{A})^\top|$ is minimized
5 update $\mathbf{C}$ such that $|\mathbf{X}_{(3)} - \mathbf{C} \boxtimes (\mathbf{B} \odot \mathbf{A})^\top|$ is minimized
6 **if** *converged* **then**
7 break out of **for** loop
8 **return** $\mathbf{A}$, $\mathbf{B}$, *and* $\mathbf{C}$

Computing the Boolean CP and Tucker Decompositions. The alternating least squares (ALS) algorithm is the "workhorse" approach for normal CP and Tucker decompositions [KB09]. With a few changes, ALS projection heuristic provides frameworks for computing the Boolean CP and Tucker decompositions as shown in Algorithms 5.1 and 5.2.

The framework for Boolean CP decomposition (Algorithm 5.1) is composed of two parts: first, the initialization of factor matrices (line 1), and second, the iterative update of each factor matrix in turn (lines 3–5). At each step of the iterative update phase, the n-th factor matrix is updated given the mode-n matricization of the input tensor $\mathcal{X}$ with the goal of minimizing the difference between the input tensor $\mathcal{X}$ and the approximate tensor reconstructed from the factor matrices using Equation (5.15), while the other factor matrices are fixed. The framework for Boolean Tucker decomposition (Algorithm 5.2) is similar to that for Boolean CP decomposition, except for (1) the additional initialization and update of the core tensor in lines 2 and 7, respectively, and (2) the tensor reconstruction in lines 4–6 that involves the core tensor and the Kronecker product between factor matrices (Equation (5.17)) The convergence criterion for Algorithms 5.1 and 5.2 is either one of the following: (1) the number of iterations exceeds the maximum value T, or (2) the sum of

Algorithm 5.2: Boolean Tucker Decomposition Framework

Input: A three-way binary tensor $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$, dimensions of a core tensor R_1, R_2, and R_3, and the maximum number of iterations T.
Output: Binary factor matrices $\mathbf{A} \in \mathbb{B}^{I \times R_1}$, $\mathbf{B} \in \mathbb{B}^{J \times R_2}$, and $\mathbf{C} \in \mathbb{B}^{K \times R_3}$, and a core tensor $\mathcal{G} \in \mathbb{B}^{R_1 \times R_2 \times R_3}$.

initialize factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$
initialize the core tensor $\mathcal{G}$ such that $\left|\mathcal{X} - \bigvee_{r_1=1}^{R_1} \bigvee_{r_2=1}^{R_2} \bigvee_{r_3=1}^{R_3} g_{r_1 r_2 r_3} \mathbf{a}_{r_1} \circ \mathbf{b}_{r_2} \circ \mathbf{c}_{r_3}\right|$ is minimized
for $t \leftarrow 1..T$ **do**
    update $\mathbf{A}$ such that $|\mathbf{X}_{(1)} - \mathbf{A} \boxtimes \mathbf{G}_{(1)} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top|$ is minimized
    update $\mathbf{B}$ such that $|\mathbf{X}_{(2)} - \mathbf{B} \boxtimes \mathbf{G}_{(2)} \boxtimes (\mathbf{C} \otimes \mathbf{A})^\top|$ is minimized
    update $\mathbf{C}$ such that $|\mathbf{X}_{(3)} - \mathbf{C} \boxtimes \mathbf{G}_{(3)} \boxtimes (\mathbf{B} \otimes \mathbf{A})^\top|$ is minimized
    update $\mathcal{G}$ such that $\left|\mathcal{X} - \bigvee_{r_1=1}^{R_1} \bigvee_{r_2=1}^{R_2} \bigvee_{r_3=1}^{R_3} g_{r_1 r_2 r_3} \mathbf{a}_{r_1} \circ \mathbf{b}_{r_2} \circ \mathbf{c}_{r_3}\right|$ is minimized
    if *converged* **then**
        break out of **for** loop
return $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, *and* $\mathcal{G}$

absolute differences between the input tensor and the reconstructed one does not change significantly for two consecutive iterations (i.e., the difference between the two errors is within a small threshold).

Using the above frameworks, Miettinen [Mie11] proposed Boolean CP and Tucker decomposition algorithms named BCP_ALS and BTucker_ALS, respectively. However, since both methods are designed to run on a single machine, their scalability and performance are limited by the computing and memory capacity of a single machine. Also, the initialization scheme used in the two methods has high space and time requirements which are proportional to the squares of the number of columns of each unfolded tensor. Due to these limitations, BCP_ALS and BTucker_ALS cannot scale up to large-scale tensors.

Walk'n'Merge [EM13b] is a different approach for Boolean CP and Tucker factorizations. Representing the tensor as a graph, Walk'n'Merge performs random walks on it to identify dense blocks (which correspond to rank-1 tensors) and merge these blocks to get larger, yet dense blocks; Walk'n'Merge orders and selects blocks based on the Minimum Description Length (MDL) principle for the CP decomposition, and obtains the Tucker decomposition from the returned blocks by merging factors and adjusting the core tensor accordingly again using the MDL principle. As a result, the dimension of a core tensor cannot be controlled with Walk'n'Merge. While Walk'n'Merge is a parallel algorithm, its scalability is still limited for large-scale tensors. Since it is not a distributed method, Walk'n'Merge suffers from the same limitations of a single machine. Also, as the size of tensor increases, the running time of Walk'n'Merge rapidly increases as we show in Section 5.5.2.

5.3 Related Works

In this section, we review previous approaches for computing Boolean and normal tensor decompositions, and present related works on the partitioning of sparse tensors, and distributed computing frameworks.

5.3.1 Boolean Tensor Decomposition

Leenen et al. [LVMDBR99] proposed the first Boolean CP decomposition algorithm. Miettinen [Mie11] presented Boolean CP and Tucker decomposition methods along with a theoretical study of Boolean tensor rank and decomposition. In [BGV13], Belohlávek et al. presented a greedy algorithm for Boolean CP decomposition of three-way binary data. In the preliminary work of this chapter [POK17], Park et al. proposed a distributed method for Boolean CP factorization running on the Apache Spark framework. Erdős et al. [EM13b] proposed a parallel algorithm called Walk'n'Merge for scalable Boolean CP and Tucker decompositions, which performs random walks to find dense blocks (rank-1 tensors) and obtains final CP and Tucker decompositions from the returned blocks by employing the MDL principle. In [EM13a], Erdős et al. applied the Boolean Tucker decomposition method proposed in [EM13b] to discover synonyms and find facts from the subject-predicate-object triples. Finding closed itemsets in N-way binary tensor [CBRB09, JTT06] is a restricted form of Boolean CP decomposition, in which an error of representing 0's as 1's is not allowed. Metzler et al. [MM15] presented an algorithm for Boolean tensor clustering, which is another form of restricted Boolean CP decomposition where one of the factor matrices has exactly one non-zero per row.

5.3.2 Normal Tensor Decomposition

Many algorithms have been developed for normal CP and Tucker decompositions.

CP Decomposition. GigaTensor [KPHF12] is the first work for large-scale CP decomposition running on MapReduce. In [JJSK16], Jeon et al. proposed SCouT for scalable coupled matrix-tensor factorization. Recently, tensor decomposition methods proposed in [KPHF12, JPF+16, JJSK16, SJK15] have been integrated into a multi-purpose tensor mining library, BIGtensor [PJLK16]. Beutel et al. [BTK+14] proposed FlexiFaCT, a scalable MapReduce algorithm to decompose matrix, tensor, and coupled matrix-tensor using stochastic gradient descent. ParCube [PFS12] is a fast and parallelizable CP decomposition method that produces sparse factors by leveraging random sampling techniques. In [LCP+17], Li et al. proposed AdaTM, which adaptively chooses parameters in a model-driven framework for an optimal memoization strategy so as to accelerate the factorization process. Smith et al. [SPK16] and Karlsson et al. [KKU16] both developed alternating least squares (ALS) and coordinate descent (CCD++) methods for parallel CP factorizations; [SPK16] also explored parallel stochastic gradient descent (SGD) method. CDTF [SSK17] provides a scalable tensor factorization method that focuses on non-zero elements of a tensor.

Tucker Decomposition. De Lathauwer et al. [LMV00] proposed foundational work on N-dimensional Tucker-ALS algorithm. As conventional Tucker-ALS methods suffer

from limited scalability, many scalable Tucker methods have been developed. Kolda et al. [KS08] proposed MET (Memory Efficient Tucker), which avoids explicitly constructing intermediate data and maximizes performance while optimally using the available memory. S-HOT [OSP+17] further improved the scalability of MET [KS08] by employing on-the-fly computation and streaming non-zeros of a tensor from the disk. Smith et al. [SK17] accelerated the factorization process by removing computational redundancies with a compressed data structure. Jeon et al. [JPF+16] provided a scalable Tucker decomposition method running on the MapReduce framework. Kaya et al. [KU16] and Oh et al. [OPSK18] designed efficient Tucker algorithms for sparse tensors. Chakaravarthy et al. [CCJ+17] proposed optimized distributed Tucker decomposition method for dense input tensors.

5.3.3 Partitioning of Sparse Tensors

For distributed tensor factorization, it is essential to use efficient partitioning methods so as to maximize parallelism and minimize communication costs between machines. There are various partitioning approaches for decomposing sparse tensors on distributed platforms. DFacTo [CV14] and CDTF [SSK17] are two systems that perform a coarse-grained partitioning of the input tensor where independent, one-dimensional block partitionings are performed for each tensor mode. With a coarse-grained partitioning, each process has all the non-zeros required for computing its output; thus, the only necessary communication is to exchange updated factor rows at each iteration. However, it has the disadvantage that dense factor matrices need to be sent to all processes in their entirety. Hypergraph partitioning methods [KU15, KU16] reduce communication volume via a fine-grained partitioning of the input tensor, in which non-zeros are assigned to processes individually. However, hypergraph partitioning involves expensive preprocessing step, which often takes more time than the actual factorization. Recently, Cartesian (or medium-grained) partitioning methods [ABK16, SK16, ATA18] have gained interests due to reduced memory and communication costs, which divide an input tensor into a 3D grid, and factor matrices into corresponding groups of rows. All of the above partitioning methods have been developed for normal tensor factorization, where factor matrices are highly dense and, accordingly, incur a high memory usage and communication overhead. On the other hand, factor matrices in BTF are usually much sparser than the normal factor matrices due to Boolean constraint, and BTF methods can usually process smaller tensors than normal decomposition techniques due to high computational complexity. Considering these characteristics of BTF, DBTF adopts a coarse-grained, vertical partitioning for the unfolded tensor and performs a Cartesian partitioning of the input tensor, which we discuss in Section 5.4.5.

5.3.4 Distributed Computing Frameworks

MapReduce [DG04] is a distributed programming model for processing large datasets in a massively parallel manner. The advantages of MapReduce include massive scalability, fault tolerance, and automatic data distribution and replication. Hadoop [had] is an open-source implementation of MapReduce. Due to the advantages of MapReduce, many data

mining tasks [KPHF12, KTF09, PPMK16, PMK16, KTS+] have used Hadoop. However, due to intensive disk I/O, Hadoop is inefficient at executing iterative algorithms [KV13]. Apache Spark [ZCD+12, ZCF+10] is a distributed data processing framework that provides capabilities for in-memory computation and data storage. These capabilities enable Spark to perform iterative computations efficiently, which are common across many machine learning and data mining algorithms, and support interactive data analytics. Spark also supports various operations other than *map* and *reduce*, such as *join*, *filter*, and *groupBy*. Thanks to these advantages, Spark has been used in several domains recently [LRC+15, WMP+14, GTW+15, ZMU+16, KPJY16].

5.4 Proposed Method

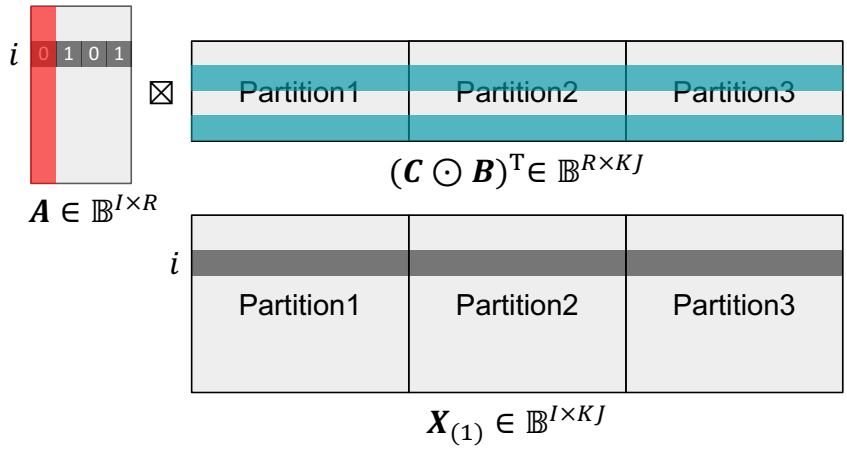
In this section, we describe DBTF, our proposed method for distributed Boolean CP (DBTF-CP) and Tucker (DBTF-TK) factorizations. There are several challenges to efficiently perform Boolean tensor factorization in a distributed environment.

1. **Minimize intermediate data.** The amount of intermediate data that are generated and shuffled across machines affects the performance of a distributed algorithm significantly. How can we minimize the intermediate data?
2. **Minimize the number of operations.** Boolean tensor factorization is an NP-hard problem [Mie11] with a high computational cost. How can we minimize the number of operations for factorizing Boolean tensors?
3. **Identify the characteristics of Boolean tensor factorization.** In contrast to the normal tensor factorization, Boolean tensor factorization applies Boolean operations to binary data. How can we utilize the characteristics of Boolean operations to design an efficient and scalable algorithm?

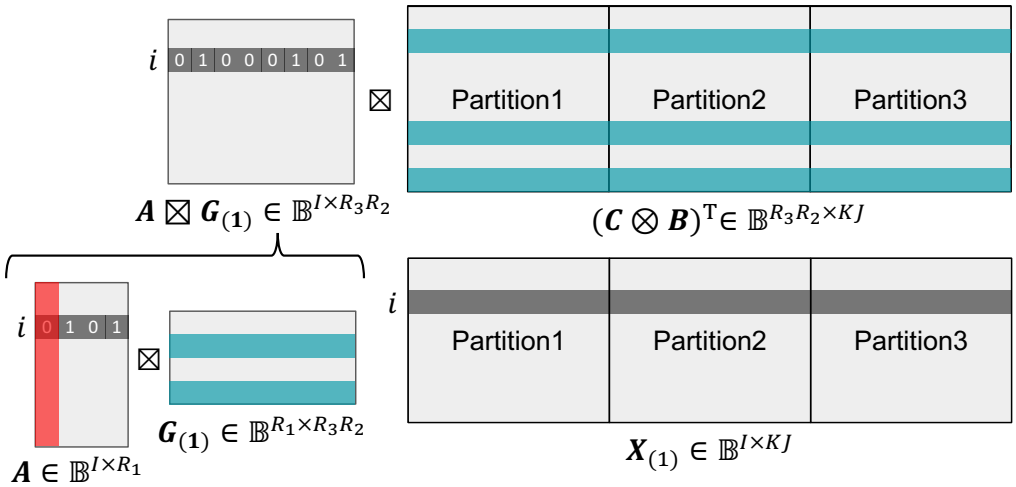
We address the above challenges with the following main ideas, which we describe in later subsections.

1. **Distributed generation and minimal transfer of intermediate data** remove redundant data generation and reduce the amount of data transfer (Section 5.4.3).
2. **Exploiting the characteristics of Boolean operation and Boolean tensor factorization** decreases the number of operations to update factor matrices (Section 5.4.4).
3. **Careful partitioning of the workload** facilitates reuse of intermediate results and minimizes data shuffling (Section 5.4.5).

We give an overview of how DBTF updates factor matrices (Section 5.4.1) and a core tensor (Section 5.4.2), and then describe how we address the aforementioned scalability challenges in detail (Sections 5.4.3 to 5.4.6). After that, we discuss implementation issues (Section 5.4.7) and provide a theoretical analysis of DBTF (Section 5.4.8). While DBTF-CP and DBTF-TK have a lot in common, DBTF-TK deals with some additional challenges. Accordingly, we organize this section such that Sections 5.4.1 to 5.4.5 describe ideas that apply to both DBTF-CP and DBTF-TK, and Sections 5.4.2 and 5.4.5.2 are dedicated to ideas that apply to DBTF-TK.



(a) Updating a factor matrix for Boolean CP factorization by DBTF-CP.



(b) Updating a factor matrix for Boolean Tucker factorization by DBTF-TK.

Figure 5.3: An overview of updating a factor matrix for (a) Boolean CP factorization by DBTF-CP, and (b) Boolean Tucker factorization by DBTF-TK. DBTF performs a column-wise update row by row. DBTF iterates over the rows of factor matrix for R (DBTF-CP) or R_1 (DBTF-TK) column (outer)-iterations in total, updating entries of each row in column c at column-iteration c ($1 \leq c \leq R$ for DBTF-CP, or $1 \leq c \leq R_1$ for DBTF-TK) to the values that result in a smaller reconstruction error. The red rectangle in $\mathbf{A}$ indicates the column c currently being updated; the gray rectangle in $\mathbf{A}$ refers to the row DBTF is visiting in row (inner)-iteration i; blue rectangles in $(\mathbf{C} \odot \mathbf{B})^\top$ or $(\mathbf{C} \otimes \mathbf{B})^\top$ are the rows that are Boolean summed to be compared against the i-th row of $\mathbf{X}_{(1)}$ (gray rectangle in $\mathbf{X}_{(1)}$). Vertical blocks in $(\mathbf{C} \odot \mathbf{B})^\top$, $(\mathbf{C} \otimes \mathbf{B})^\top$, and $\mathbf{X}_{(1)}$ represent partitions of the data (see Section 5.4.5 for details on partitioning).

5.4.1 Updating a Factor Matrix

DBTF is a distributed method for Boolean CP (DBTF-CP) and Tucker (DBTF-TK) factorizations based on the framework described in Algorithms 5.1 and 5.2, respectively. The core operation of DBTF-CP and DBTF-TK is updating factor matrices (lines 3–5 in Algorithm 5.1, and lines 4–6 in Algorithm 5.2). Since the update steps are similar, we focus on updating the factor matrix $\mathbf{A}$.

DBTF performs a column-wise update row by row. This is done with doubly nested loops, where the outer loop selects a column to update, and the inner loop iterates over the rows of a factor matrix, updating only those entries in the column selected by the outer loop. In other words, DBTF iterates over the rows of factor matrix for R (DBTF-CP) or R_1 (DBTF-TK) column (outer)-iterations in total, updating entries of each row in column c at column-iteration c ($1 \leq c \leq R$ for DBTF-CP, or $1 \leq c \leq R_1$ for DBTF-TK) to the values that result in a smaller reconstruction error. This is a greedy approach that updates each entry to the value that yields a better accuracy while all other entries in the same row are fixed; as a result, it does not consider all combinations of values for factor matrix elements. We also considered an exact approach that explores every possible value assignment, but preliminary tests showed that the greedy heuristic performs very closely to the exact search while being much faster than the exact search which takes exponential time with respect to R or R_1. Figure 5.3 shows an overview of how DBTF updates a factor matrix. In Figure 5.3, the red rectangle indicates the column c currently being updated, and the gray rectangle in $\mathbf{A}$ refers to the row DBTF is visiting in row (inner)-iteration i.

Updating a Factor Matrix in DBTF-CP. The objective of updating the factor matrix in DBTF-CP is to minimize the difference between the unfolded input tensor $\mathbf{X}_{(1)}$ and the approximate tensor $\mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top$. Let c refer to the column to be updated. Then, DBTF-CP computes $|\mathbf{X}_{(1)} - \mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top|$ for each of the possible values for the entries in column c (i.e., $\vec{a}_{:c}$), and updates the column c to the set of values that yield the smallest difference. In order to calculate the difference at row-iteration i, $[\mathbf{X}_{(1)}]_{i:}$ (gray rectangle in $\mathbf{X}_{(1)}$ of Figure 5.3) is compared against $[\mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top]_{i:} = \mathbf{a}_{i:} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top$ (Figure 5.3a). Then, an entry in a_{ic} is updated to the value that gives a smaller difference, i.e., $\left|[\mathbf{X}_{(1)}]_{i:} - \mathbf{a}_{i:} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top\right|$.

Updating a Factor Matrix in DBTF-TK. DBTF-TK updates the factor matrix such that the difference between $\mathbf{X}_{(1)}$ and $\mathbf{A} \boxtimes \mathbf{G}_{(1)} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top$ is minimized. DBTF-TK calculates the difference at row-iteration i by comparing $[\mathbf{X}_{(1)}]_{i:}$ against $[\mathbf{A} \boxtimes \mathbf{G}_{(1)} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top]_{i:} = [\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i:} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top$ (Figure 5.3b), and updates an entry in a_{ic} to the value resulting in a smaller difference, i.e., $\left|[\mathbf{X}_{(1)}]_{i:} - [\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i:} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top\right|$.

Lemma 5.1. Boolean Multiplication:
$\mathbf{a}_{i:} \boxtimes \mathbf{B}$ is the same as selecting rows of $\mathbf{B}$ that correspond to the indices of non-zeros of $\mathbf{a}_{i:}$, and performing a Boolean summation of those rows.

Proof. This follows directly from the definition of Boolean matrix product $\boxtimes$ (Equa-

tion (5.7)). ■

Consider Figure 5.3a as an example: Since $\mathbf{a}_{i:}$ is 0101 (gray rectangle in $\mathbf{A}$), $\mathbf{a}_{i:} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top$ is identical to the Boolean summation of the second and fourth rows of $(\mathbf{C} \odot \mathbf{B})^\top$ (blue rectangles). Similarly, in Figure 5.3b, $[\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i:} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top$ is the same as the Boolean summation of the second, sixth, and eighth rows of $(\mathbf{C} \otimes \mathbf{B})^\top$ (blue rectangles) as $[\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i:}$ is 01000101.

Note that an update of the i-th row of $\mathbf{A}$ does not depend on those of its other rows since $\mathbf{a}_{i:} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top$ or $[\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i:} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top$ needs to be compared only with $[\mathbf{X}_{(1)}]_{i:}$. Therefore, the determination of whether to update an entry of some row in $\vec{a}_{:c}$ to 0 or 1 can be made independently of the decisions for entries in other rows. Also, notice in Figure 5.3b that while it is factor matrix $\mathbf{A}$ that DBTF-TK tries to update, it is not the rows in $\mathbf{A}$ that determine which rows in $(\mathbf{C} \otimes \mathbf{B})^\top$ are to be summed as in Figure 5.3a, but those in the intermediate matrix $\mathbf{A} \boxtimes \mathbf{G}_{(1)}$.

Depending on the distribution of non-zeros in the input tensor, and how factor matrices and a core tensor have been initialized and updated, the factor matrix currently being updated may be updated to contain only zeros. When this happens, the intermediate matrix constructed with a Khatri-Rao (e.g., $(\mathbf{C} \odot \mathbf{B})^\top$) or a Kronecker product (e.g., $(\mathbf{C} \otimes \mathbf{B})^\top$) at the following iteration will consist of only zeros, and as a result, the difference between the approximate tensor and the input tensor will be always the same, regardless of how the factor matrix is updated. We handle this issue by providing the ability to upper bound the maximum percentage of zeros in the column being updated. When the percentage of zeros in the current column exceeds the given threshold, DBTF finds values different from the current assignments for a subset of rows, which will make the sparsity of the current column become less than the upper bound with the smallest increase in error, and updates those rows accordingly.

5.4.2 Updating a Core Tensor

Boolean Tucker factorization involves an additional task of updating a core tensor $\mathcal{G}$. How DBTF-TK updates a core tensor is based on BTucker_ALS [Mie11]. Below we describe the main observations used by DBTF-TK and BTucker_ALS for updating $\mathcal{G}$ and explain how DBTF-TK further reduces the amount of computation.

Given the definition of Boolean Tucker decomposition (Equation (5.16)), an (i, j, k)-th element of an approximate tensor $\tilde{\mathcal{X}}$ is computed as follows:

$$\tilde{x}_{ijk} = \bigvee_{r_1=1}^{R_1} \bigvee_{r_2=1}^{R_2} \bigvee_{r_3=1}^{R_3} g_{r_1 r_2 r_3} a_{i r_1} b_{j r_2} c_{k r_3}. \tag{5.18}$$

That is, every element of $\mathcal{G}$ is involved with the computation of $\tilde{x}_{ijk}$, and thus, flipping an element in $\mathcal{G}$ can affect the entire $\tilde{\mathcal{X}}$. However, we observe that the value of $g_{r_1 r_2 r_3}$ does not affect the product $g_{r_1 r_2 r_3} a_{i r_1} b_{j r_2} c_{k r_3}$ if $a_{i r_1} b_{j r_2} c_{k r_3}$ is 0. Therefore, only those (i, j, k)s for which $a_{i r_1} b_{j r_2} c_{k r_3} \neq 0$ are considered in DBTF-TK and BTucker_ALS. We also notice that,

as a result of Boolean sum, if there exists some (r_1, r_2, r_3) such that $g_{r_1 r_2 r_3} a_{i r_1} b_{j r_2} c_{k r_3} = 1$, then $\tilde{x}_{ijk} = 1$ regardless of values of other elements in $\mathcal{G}$.

Based on these observations, both methods compute the partial gain of flipping the (r_1, r_2, r_3)-th element for which there exists some (i, j, k) such that $a_{i r_1} b_{j r_2} c_{k r_3} = 1$, and update those elements having a positive gain.

- If $g_{r_1 r_2 r_3}$ and $\tilde{x}_{ijk}$ are both 0, then there exists no $(\alpha, \beta, \gamma) \neq (r_1, r_2, r_3)$ such that $g_{\alpha\beta\gamma} a_{i\alpha} b_{j\beta} c_{k\gamma} = 1$. If $x_{ijk} = 1$ in this case, setting $g_{r_1 r_2 r_3}$ to 1 results in a partial gain, since $g_{r_1 r_2 r_3} a_{i r_1} b_{j r_2} c_{k r_3}$ becomes 1, and $\tilde{x}_{ijk} = x_{ijk} = 1$.
- If $g_{r_1 r_2 r_3}$ is 1, $\tilde{x}_{ijk}$ is guaranteed to be 1. However, flipping $g_{r_1 r_2 r_3}$ back to 0 does not necessarily lead to a partial gain since there might be other $(\alpha, \beta, \gamma) \neq (r_1, r_2, r_3)$ such that $g_{\alpha\beta\gamma} a_{i\alpha} b_{j\beta} c_{k\gamma} = 1$. So in this case, under the condition that no such (α, β, γ) exists and x_{ijk} is 0, setting $g_{r_1 r_2 r_3}$ to 0 leads to a partial gain.

We further reduce the amount of computation by utilizing vectors s_I, s_J, and s_K, which contain the rowwise sum of entries in factor matrices $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$ that are in those columns selected by entries in $\mathcal{G}$. Let us assume that $a_{i r_1} b_{j r_2} c_{k r_3} = 1$ for some (i, j, k) and (r_1, r_2, r_3). First, when $g_{r_1 r_2 r_3} = 0$ and $x_{ijk} = 1$, we need to know whether $\tilde{x}_{ijk}$ is 0 or not in order to compute the partial gain. We observe that $\tilde{x}_{ijk} = 0$ if at least one of $s_I(i), s_J(j)$, and $s_K(k)$ is zero, since when this condition is satisfied, $a_{i\alpha} b_{j\beta} c_{k\gamma} = 0$ for any (α, β, γ) in $\mathcal{G}$. Second, if $g_{r_1 r_2 r_3} = 1$, $\tilde{x}_{ijk}$ is equal to 1. When $x_{ijk} = 1$ in this case, in order to compute the partial gain, we need to know whether there exists some $(\alpha, \beta, \gamma) \neq (r_1, r_2, r_3)$ that also contributes to $\tilde{x}_{ijk} = 1$. We note that if all of $s_I(i), s_J(j)$, and $s_K(k)$ are equal to one, then $g_{r_1 r_2 r_3}$ is the only element in $\mathcal{G}$ that turns on $\tilde{x}_{ijk}$, since otherwise, there exists at least one other element in $\mathcal{G}$ also contributing to $\tilde{x}_{ijk}$, which is impossible given that $s_I(i) = s_J(j) = s_K(k) = 1$. In both cases, vectors s_I, s_J, and s_K help us avoid visiting elements in $\mathcal{G}$ repeatedly, and enable DBTF-TK to skip the current (i, j, k) and the following ones for which no partial gain can be obtained.

While the above ideas allow the update of a core tensor $\mathcal{G}$, updating $\mathcal{G}$ in a distributed environment poses a challenge of how to distribute the workload among machines, which we describe in Section 5.4.5.2.

5.4.3 Distributed Generation and Minimal Transfer of Intermediate Data

The first challenge for performing Boolean tensor factorization in a distributed manner is how to generate and distribute the intermediate data efficiently. In particular, updating a factor matrix involves the following types of intermediate data: (1) a Khatri-Rao product of two factor matrices (e.g., $(\mathbf{C} \odot \mathbf{B})^\top$), (2) a Kronecker product of two factor matrices (e.g., $(\mathbf{C} \otimes \mathbf{B})^\top$), and (3) an unfolded tensor (e.g., $\mathbf{X}_{(1)}$).

Khatri-Rao and Kronecker Products. A naive method for processing the Khatri-Rao and Kronecker products is to construct the entire product first, and then distribute its partitions across machines. While Boolean factors are known to be sparser than the

normal counterparts with real-valued entries [MM15], performing the entire Khatri-Rao or Kronecker product is still an expensive operation. Also, since one of the two matrices involved in the product is always updated in the previous update procedure (Algorithms 5.1 and 5.2), prior Khatri-Rao or Kronecker products cannot be reused. Our idea is to distribute only the factor matrices, and then let each machine generate the part of the product it needs, which is possible according to the definition of Khatri-Rao product,

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{b}_{:1} & a_{12}\mathbf{b}_{:2} & \cdots & a_{1R}\mathbf{b}_{:R} \\ a_{21}\mathbf{b}_{:1} & a_{22}\mathbf{b}_{:2} & \cdots & a_{2R}\mathbf{b}_{:R} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}\mathbf{b}_{:1} & a_{I2}\mathbf{b}_{:2} & \cdots & a_{IR}\mathbf{b}_{:R} \end{bmatrix}, \tag{5.19}$$

and that of Kronecker product (Equation (5.2)). We notice from Equations (5.2) and (5.19) that a specific range of rows of Khatri-Rao or Kronecker product can be constructed if we have the two factor matrices and the corresponding range of row indices. With this change, we only need to broadcast relatively small factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ along with the index ranges assigned for each machine without having to materialize the entire product.

Unfolded Tensor. While the Khatri-Rao or Kronecker products are computed iteratively, matricizations of an input tensor need to be performed only once. However, in contrast to the Khatri-Rao and Kronecker products, we cannot avoid shuffling the entire unfolded tensor as we have no characteristics to exploit as in the case of Khatri-Rao or Kronecker product. Furthermore, unfolded tensors take up the largest space during the execution of DBTF. In particular, its row dimension quickly becomes very large as the sizes of factor matrices increase. Therefore, we partition the unfolded tensors in the beginning and do not shuffle them afterwards. We do vertical partitioning of the Khatri-Rao and Kronecker products and unfolded tensors as shown in Figure 5.3 (see Section 5.4.5 for more details on the partitioning of unfolded tensors).

5.4.4 Exploiting the Characteristics of Boolean Operation and Boolean Tensor Factorization

The second and the most important challenge for efficient and scalable Boolean tensor factorization is how to minimize the number of operations for updating factor matrices. In this subsection, we describe the problem in detail and present our solution.

5.4.4.1 Problem

Given our procedure to update factor matrices (Section 5.4.1), the two most frequently performed tasks are (1) computing the Boolean sums of selected rows of $(\mathbf{C} \odot \mathbf{B})^\top$ (CP factorization) or $(\mathbf{C} \otimes \mathbf{B})^\top$ (Tucker factorization), and (2) comparing the resulting row with the corresponding row of $\mathbf{X}_{(1)}$. Assuming that all factor matrices are of the same size, I-by-R, the first task takes $O(RI^2)$ or $O(R^2I^2)$ time (for CP and Tucker factorizations, respectively), and the second task takes $O(I^2)$ time. Since we compute the errors for both cases of when each factor matrix entry is set to 0 and 1, each task needs to be performed

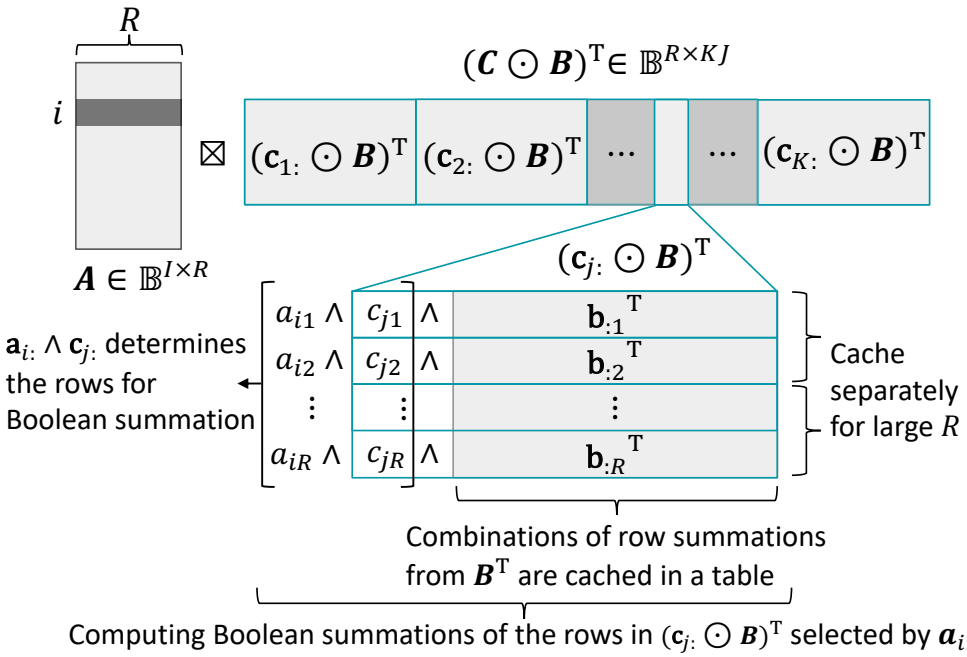


Figure 5.4: DBTF-CP reduces intermediate operations by exploiting the characteristics of Boolean CP factorization. Blue rectangles in $(\mathbf{C} \odot \mathbf{B})^{\top}$ correspond to K vector-matrix Khatri-Rao products, among which $(\mathbf{c}_{j:} \odot \mathbf{B})^{\top}$ is shown in detail. $\mathbf{B}^{\top}$ is the target for row summation. A Boolean vector $\mathbf{a}_{i:} \wedge \mathbf{c}_{j:}$ determines which rows in $\mathbf{B}^{\top}$ are to be summed to compute the row summation of $(\mathbf{c}_{j:} \odot \mathbf{B})^{\top}$. Combinations of the row summations of $\mathbf{B}^{\top}$ are cached. For large R, rows of $\mathbf{B}^{\top}$ are split into multiple, smaller groups, each of which is cached separately.

$2RI$ times to update a factor matrix of size I-by-R; then, updating all three factor matrices for T iterations performs each task $6TRI$ times in total. Due to high computational costs and a large number of repetitions, it is crucial to minimize the number of intermediate operations involved with these tasks.

5.4.4.2 Our Solution

Overview. We start with the following observations:

- By Lemma 5.1, DBTF computes the Boolean sum of selected rows of $(\mathbf{C} \odot \mathbf{B})^{\top}$ (DBTF-CP), or $(\mathbf{C} \otimes \mathbf{B})^{\top}$ (DBTF-TK). This amounts to performing a specific set of operations repeatedly, which we describe below.
- Khatri-Rao and Kronecker products can be expressed by vector-matrix (VM) Khatri-Rao and Kronecker products, respectively (Equations (5.3) and (5.5)).
- Given factor matrices of size I-by-R, there are 2^R and 2^{R^2} combinations of selecting rows from $(\mathbf{C} \odot \mathbf{B})^{\top} \in \mathbb{B}^{R \times I^2}$ and $(\mathbf{C} \otimes \mathbf{B})^{\top} \in \mathbb{B}^{R^2 \times I^2}$, respectively.

Our main idea is to exploit the characteristics of Boolean operation and Boolean tensor factorization as summarized in the above observations to reduce the number of

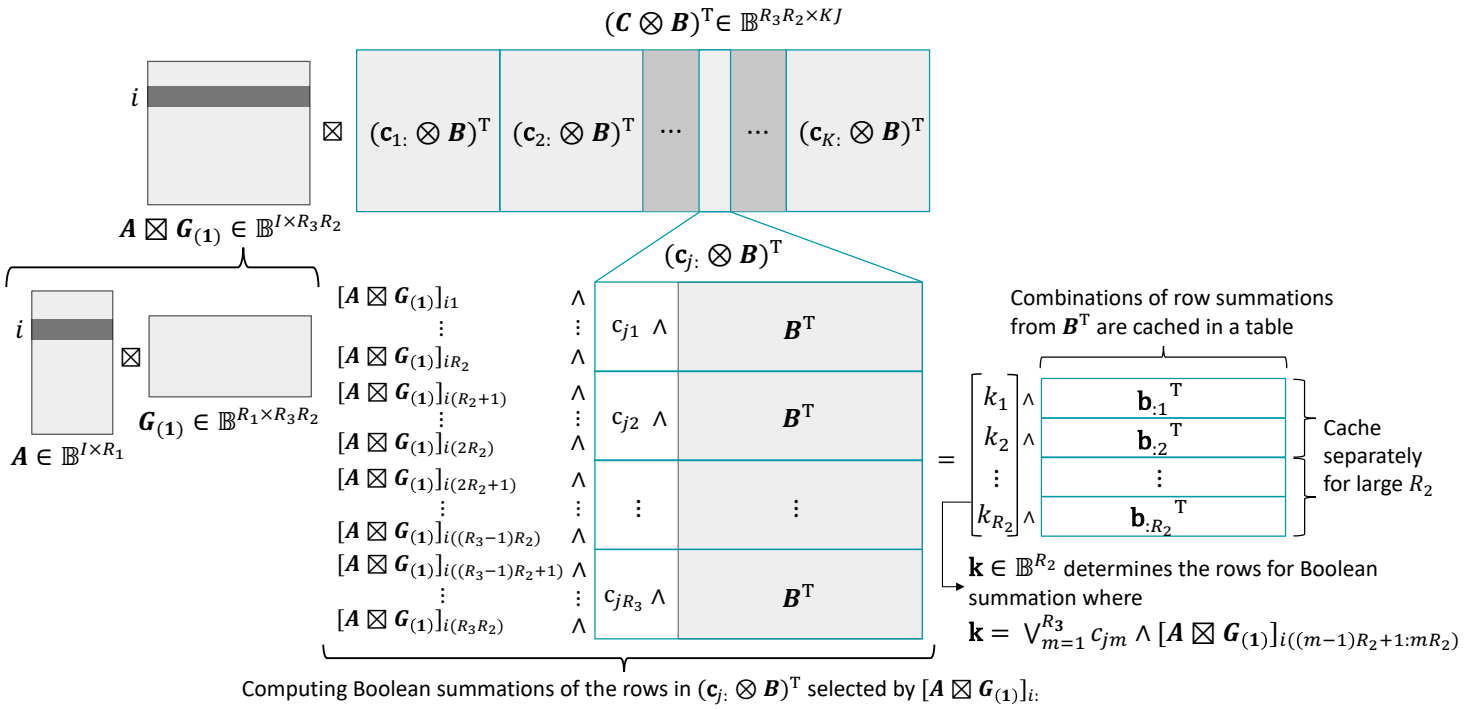


Figure 5.5: DBTF-TK reduces intermediate operations by exploiting the characteristics of Boolean Tucker factorization. Blue rectangles in $(\mathbf{C} \otimes \mathbf{B})^\top$ correspond to K vector-matrix Kronecker products, among which $(\mathbf{c}_{j:} \otimes \mathbf{B})^\top$ is shown in detail. $\mathbf{B}^\top$ is the target for row summation. A Boolean vector $\mathbf{k} = \bigvee_{m=1}^{R_3} c_{jm} \wedge [\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i((m-1)R_2+1:mR_2)}$ determines which rows in $\mathbf{B}^\top$ are to be summed to compute the row summation of $(\mathbf{c}_{j:} \otimes \mathbf{B})^\top$. Combinations of the row summations of $\mathbf{B}^\top$ are cached. For large R_2, rows of $\mathbf{B}^\top$ are split into multiple, smaller groups, each of which is cached separately.

intermediate steps to perform Boolean row summations. Figures 5.4 and 5.5 present an overview of our idea for Boolean CP and Tucker factorizations. We note that according to Equation (5.4),

$$(\mathbf{C} \odot \mathbf{B})^\top = [(\mathbf{c}_{1:} \odot \mathbf{B})^\top \ (\mathbf{c}_{2:} \odot \mathbf{B})^\top \ \cdots \ (\mathbf{c}_{K:} \odot \mathbf{B})^\top].$$

Similarly, we notice that by Equation (5.2),

$$(\mathbf{C} \otimes \mathbf{B})^\top = [(\mathbf{c}_{1:} \otimes \mathbf{B})^\top \ (\mathbf{c}_{2:} \otimes \mathbf{B})^\top \ \cdots \ (\mathbf{c}_{K:} \otimes \mathbf{B})^\top].$$

Blue rectangles in $(\mathbf{C} \odot \mathbf{B})^\top$ and $(\mathbf{C} \otimes \mathbf{B})^\top$ (Figures 5.4 and 5.5) correspond to K VM Khatri-Rao products, $[(\mathbf{c}_{1:} \odot \mathbf{B})^\top, \ldots, (\mathbf{c}_{K:} \odot \mathbf{B})^\top]$, and K VM Kronecker products, $[(\mathbf{c}_{1:} \otimes \mathbf{B})^\top, \ldots, (\mathbf{c}_{K:} \otimes \mathbf{B})^\top]$, respectively. Since a row of $(\mathbf{C} \odot \mathbf{B})^\top$ or $(\mathbf{C} \otimes \mathbf{B})^\top$ is made up of a sequence of K corresponding rows of VM Khatri-Rao or VM Kronecker products, the Boolean sum of the selected rows of $(\mathbf{C} \odot \mathbf{B})^\top$ or $(\mathbf{C} \otimes \mathbf{B})^\top$ can be constructed by summing up the same set of rows in each VM Khatri-Rao or VM Kronecker product, and concatenating the resulting rows into a single row.

Selecting Rows of $\mathbf{B}^\top$ in DBTF-CP. Assuming that the row $\mathbf{a}_{i:}$ is being updated as in Figure 5.4, we observe that computing Boolean row summations of each $(\mathbf{c}_{j:} \odot \mathbf{B})^\top$ amounts to summing up the rows in $\mathbf{B}^\top$ that are selected by the next two conditions. First, we choose all those rows of $\mathbf{B}^\top$ whose corresponding entries in $\mathbf{c}_{j:}$ are 1. Since all other rows are empty vectors by the definition of Khatri-Rao product (Equation (5.4)), they can be ignored in computing Boolean row summations. Second, we pick the set of rows from each $(\mathbf{c}_{j:} \odot \mathbf{B})^\top$ selected by the value of row $\mathbf{a}_{i:}$ as they are the targets of Boolean summation. Therefore, the value of Boolean AND ($\wedge$) between the rows $\mathbf{a}_{i:}$ and $\mathbf{c}_{j:}$ determines which rows are to be used for the row summation of $(\mathbf{c}_{j:} \odot \mathbf{B})^\top$.

Selecting Rows of $\mathbf{B}^\top$ in DBTF-TK. Assuming that the row $\mathbf{a}_{i:}$ is being updated, we can compute Boolean row summations of each $(\mathbf{c}_{j:} \otimes \mathbf{B})^\top$ by employing an approach similar to that used for Boolean CP factorization, which is to sum up those rows in $[\mathbf{B}\ \mathbf{B}\ \ldots\ \mathbf{B}]^\top \in \mathbb{B}^{R_3 R_2 \times J}$ that are selected by the non-zeros of $[\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i:}$ and $\mathbf{c}_{j:}$, as depicted in Figure 5.5. While straightforward, this approach is not efficient as in Boolean CP factorization as the number of rows of the intermediate $(\mathbf{c}_{j:} \otimes \mathbf{B})^\top$ is R_3 times that of $(\mathbf{c}_{j:} \odot \mathbf{B})^\top$. Furthermore, we observe in Figure 5.5 that $\mathbf{B}^\top$ is repeatedly involved in constructing $(\mathbf{c}_{j:} \otimes \mathbf{B})^\top$; therefore, if we know which rows of $\mathbf{B}^\top$ are to be selected by $[\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i:}$ and $\mathbf{c}_{j:}$, we can obtain the Boolean row summation immediately from $\mathbf{B}^\top$, without going over $\mathbf{B}^\top$ R_3 times. Among the rows of the m-th $\mathbf{B}^\top$ in Figure 5.5, which rows are to be summed is determined by the value of Boolean AND ($\wedge$) between c_{jm} and $[\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i((m-1)R_2+1:mR_2)}$. Thus, the Boolean OR ($\vee$) of all these Boolean ANDs determines which rows need to be summed together to obtain the Boolean row summation for $(\mathbf{c}_{j:} \otimes \mathbf{B})^\top$.

Caching. In computing a row summation of $(\mathbf{C} \odot \mathbf{B})^\top$ or $(\mathbf{C} \otimes \mathbf{B})^\top$, we repeatedly sum a subset of rows in $\mathbf{B}^\top$ selected by the aforementioned conditions for each VM Khatri-Rao or Kronecker product. Then, if we can reuse row summation results, we can avoid summing up the same set of rows again and again. DBTF precalculates combinations of row summations of $\mathbf{B}^\top$ and caches the results in a table in memory. This table maps a specific subset of selected rows in $\mathbf{B}^\top$ to its Boolean summation result. In summary, we use the followings as a key to this cache table:

- $\mathbf{a}_{i:} \wedge \mathbf{c}_{j:}$ for Boolean CP factorization
- $\bigvee_{m=1}^{R_3} c_{jm} \wedge [\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i((m-1)R_2+1:mR_2)}$ for Boolean Tucker factorization

An issue related to this approach is that the space required for the table increases exponentially with the rank size. Thus, when R becomes larger than a threshold value V, we divide rows evenly into $\lceil R/V \rceil$ smaller groups, construct smaller tables for each group, and then perform additional Boolean summation of rows that we obtain from the smaller tables.

Lemma 5.2. Number of Cache Tables:
Given R and V, the number of required cache tables is $\lceil R/V \rceil$, and each table is of size $2^{\lceil R/\lceil R/V \rceil \rceil}$.

For instance, when the rank R is 18 and V is set to 10, we create two tables of size 2^9, the first one storing possible summations of ${\mathbf{b}_{:1}}^\top, \ldots, \mathbf{b}_{:9}^\top$, and the second one storing those of ${\mathbf{b}_{:10}}^\top, \ldots, \mathbf{b}_{:18}^\top$. This provides a good trade-off between space and time: While it requires additional computations for row summations, it reduces the amount of memory used for the tables, and also the time to construct them, which also increases exponentially with R.

In addition to the cache table containing the row summation results of $\mathbf{B}^\top$, we build another cache table for Boolean Tucker factorization, which maps a set of rows of an unfolded core tensor (e.g., $\mathbf{G}_{(1)}$) to its Boolean summation result. Note that, in contrast to the case of Boolean CP factorization, the row $\mathbf{a}_{i:}$ is not directly used for computing a cache key in Boolean Tucker factorization. Instead, $\mathbf{a}_{i:}$ determines the set of rows of $\mathbf{G}_{(1)}$ that are to be summed, and the resulting row summation, $\mathbf{a}_{i:} \boxtimes \mathbf{G}_{(1)}$, is then used for constructing the cache key. In order to avoid summing up the same set of rows in $\mathbf{G}_{(1)}$ repeatedly, DBTF-TK also precomputes the combinations of row summations of $\mathbf{G}_{(1)}$ and stores them in an in-memory table. For large R, this additional table is also split into smaller ones in the same way as discussed above.

Note that the benefit of caching depends on a few factors such as the density of factors and a core tensor, the threshold value V, and the upper bound on the maximum percentage of zeros in columns of a factor matrix (Section 5.4.1). When factors and a core tensor are updated to be sparse, it is advisable to limit V to some small values such that we calculate combinations in a small amount of time, while avoiding computing too many combinations that are less likely to be used. When factorizing a dense tensor, it is recommended to try a higher value for V to benefit more from caching.

5.4.5 Careful Partitioning of the Workload

The third challenge is how to partition the workload effectively. A partition is a unit of workload distributed across machines. Partitioning is important since it determines the level of parallelism and the amount of shuffled data. Our goal is to fully utilize the available computing resources, while minimizing the amount of network traffic. In the following subsections, we describe how DBTF-CP and DBTF-TK partition unfolded tensors (Section 5.4.5.1), and how DBTF-TK partitions an input tensor (Section 5.4.5.2).

5.4.5.1 Unfolded Tensors

As introduced in Section 5.4.3, DBTF partitions the unfolded tensor vertically: A single partition covers a range of consecutive columns. The main reason for choosing vertical partitioning instead of horizontal one is because with vertical partitioning, each partition can perform Boolean summations of the rows assigned to it and compute their errors independently, with no need of communications between partitions. On the other hand, with horizontal partitioning, each partition needs to communicate with others to be able to compute the Boolean row summations. Furthermore, horizontal partitioning splits the dimensionality I, which is usually smaller than the product of dimensionalities KJ. Thus, the maximum number of partitions supported by horizontal partitioning

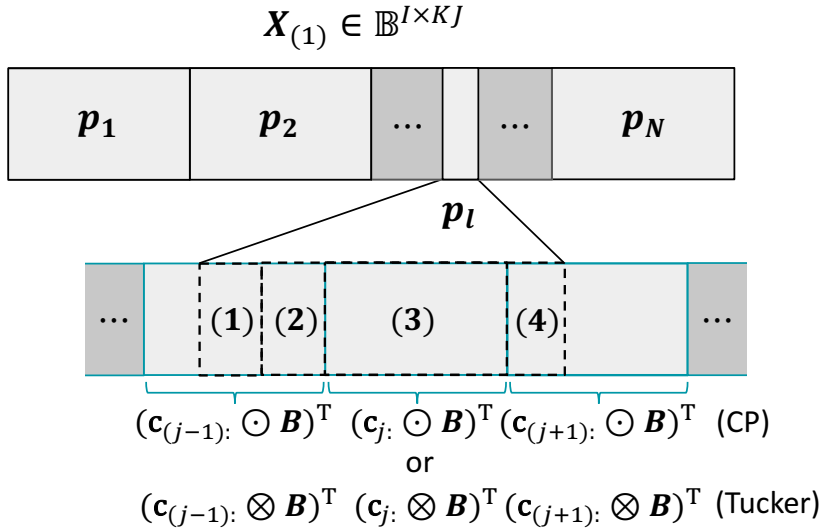


Figure 5.6: An overview of partitioning. DBTF partitions the unfolded input tensor vertically into a total of N partitions $p_1, p_2, \ldots, p_N$, among which the l-th partition p_l is shown in detail. A partition is further divided into "blocks" (rectangles in dashed lines) by the vertical boundaries between the underlying vector-matrix (VM) Khatri-Rao (CP factorization) or VM Kronecker (Tucker factorization) products, which correspond to blue rectangles. Numbers in p_l refer to the types of blocks a partition can be split into.

is normally smaller than that by vertical partitioning, which could lower the level of parallelism.

Since the workloads are vertically partitioned, each partition computes an error only for the vertically split part of the row distributed to it. Therefore, errors from all partitions should be considered together to make the decision of whether to update an entry to 0 or 1. DBTF collects from all partitions the errors for the entries in the column being updated and sets each one to the value with the smallest error.

DBTF further splits the vertical partitions of an unfolded tensor in a computation-friendly manner. By "computation-friendly," we mean structuring the partitions in such a way that facilitates an efficient computation of row summation results as discussed in Section 5.4.4. This is crucial since the number of operations performed directly affects the performance of DBTF. The target for row summation in DBTF is $\mathbf{B}^\top$ as shown in Figures 5.4 and 5.5. However, the horizontal length of each partition is not always the same as or a multiple of that of $\mathbf{B}^\top$. Depending on the number of partitions, and the size of $\mathbf{C}$ and $\mathbf{B}$, a partition may cross the vertical boundaries of multiple VM Khatri-Rao or Kronecker products, or may be shorter than one VM Khatri-Rao or Kronecker product.

Figure 5.6 presents an overview of our idea for computation-friendly partitioning in DBTF. DBTF partitions the unfolded input tensor into a total of N partitions p_1, p_2,

..., p_N, among which the l-th partition p_l is shown in detail. A partition is further divided into "blocks" (rectangles in dashed lines) by the vertical boundaries between the underlying Khatri-Rao (CP factorization) or Kronecker (Tucker factorization) products, which correspond to blue rectangles. Numbers in p_l refer to the types of blocks a partition can be split into. Since the target for row summation is $\mathbf{B}^\top$, with this organization, each block of a partition can efficiently obtain its row summation results.

Lemma 5.3. Block Types of a Partition:
A partition can have at most three types of blocks.

Proof. There are four different types of blocks—(1), (2), (3), and (4)—as shown in Figure 5.6. If the horizontal length of a partition is smaller than or equal to that of a single Khatri-Rao or Kronecker product, it can consist of up to two blocks. When the partition does not cross the vertical boundary between Khatri-Rao or Kronecker products, it consists of a single block, which corresponds to one of the four types (1), (2), (3), and (4). On the other hand, when the partition crosses the vertical boundary between products, it consists of two blocks of types (2) and (4).

If the horizontal length of a partition is larger than that of a single Khatri-Rao or Kronecker product, multiple blocks comprise the partition: Possible combinations of blocks are (2)+(3)*+(4), (3)$^+$+(4)$^?$, and (2)$^?$+(3)$^+$ where the star (*) superscript denotes that the preceding type is repeated zero or more times, the plus (+) superscript denotes that the preceding type is repeated one or more times, and the question mark (?) superscript denotes that the preceding type is repeated zero or one time. Thus, in all cases, a partition can have at most three types of blocks. ■

An issue with respect to the use of caching is that the horizontal length of blocks of types (1), (2), and (4) is smaller than that of a single Khatri-Rao or Kronecker product. If a partition has such blocks, we compute additional cache tables for the smaller blocks from the full-size one so that these blocks can also exploit caching. By Lemma 5.3, at most two smaller tables need to be computed for each partition, and each one can be built efficiently as constructing it requires only a single pass over the full-size cache.

Partitioning is a one-off task in DBTF. DBTF constructs these partitions in the beginning and caches the entire partitions for efficiency.

5.4.5.2 Input Tensor

DBTF-TK updates a core tensor $\mathcal{G}$ at each iteration (Algorithm 5.2). In DBTF-TK, an input tensor $\mathcal{X}$ is necessary for updating $\mathcal{G}$: As described in Section 5.4.2, the way DBTF-TK updates an element $g_{r_1r_2r_3}$ of a core tensor requires accessing an input tensor element x_{ijk} for all i, j, and k for which $a_{ir_1}b_{jr_2}c_{kr_3} = 1$.

For distributed computation, the workload of updating a core tensor $\mathcal{G}$ needs to be partitioned across the cluster. Considering that $\mathcal{G}$ is updated entry by entry in DBTF-TK, and updating each element of $\mathcal{G}$ requires accessing an input tensor $\mathcal{X}$ entry by entry,

we take into account two different workload partitioning approaches: (1) partitioning the core tensor $\mathcal{G}$ and (2) partitioning the input tensor $\mathcal{X}$. Partitioning the core tensor indicates that entries in different partitions of $\mathcal{G}$ are updated concurrently. However, in DBTF-TK, updating an entry (α, β, γ) in $\mathcal{G}$ involves accessing $(\alpha', \beta', \gamma') \neq (\alpha, \beta, \gamma)$. As each partition updates a different part of $\mathcal{G}$, different partitions may have different views of $\mathcal{G}$, which would result in an incorrect update. Thus, it is not possible to update different entries in $\mathcal{G}$ concurrently. By partitioning the input tensor, on the other hand, only one entry of $\mathcal{G}$ is updated at a time, while entries in different partitions of an input tensor $\mathcal{X}$ are processed in parallel. In this way, all partitions share the global view of the core tensor, and each partition computes the partial gain that can be obtained by flipping an entry (α, β, γ) in $\mathcal{G}$ with regard to the entries of an input tensor assigned to it.

DBTF-TK partitions the input tensor $\mathcal{X}$ into a total of N non-overlapping subtensors ${}_p\mathcal{X}_1, {}_p\mathcal{X}_1, \ldots, {}_p\mathcal{X}_N$ such that they satisfy the following conditions:

1. ${}_p\mathcal{X}_t$ is associated with three ranges, I_t, J_t, and K_t, such that $|I_i| \times |J_i| \times |K_i| \approx |I_j| \times |J_j| \times |K_j|$ for all $i, j \in [1 \,..\, N]$.
2. ${}_p\mathcal{X}_t$ contains all $x_{ijk} \in \mathcal{X}$ for $i \in I_t, j \in J_t$, and $k \in K_t$.
3. $\bigcup_{t=1}^{N} {}_p\mathcal{X}_t = \mathcal{X}$, and ${}_p\mathcal{X}_i \cap {}_p\mathcal{X}_j = \emptyset$ for all $i, j \in [1 \,..\, N]$ $(i \neq j)$.

With the input tensor partitioned as above, the parallel updates of a core tensor $\mathcal{G}$ in N partitions are synchronized on the update of each entry in $\mathcal{G}$.

5.4.6 Putting Things Together

In this section, we present algorithms for DBTF and provide a brief description of their relationships: DBTF-CP is given in Algorithms 5.3 to 5.7, and DBTF-TK is presented in Algorithms 5.4 and 5.6 to 5.11. The "**distributed**" keyword in the algorithm indicates that the marked section is performed in a fully distributed manner. We also briefly summarize what data are transferred across the network.

5.4.6.1 Partitioning

DBTF-CP and DBTF-TK first partition the unfolded input tensors (lines 1–3 in Algorithms 5.3 and 5.8): Each unfolded tensor is vertically partitioned and then cached across machines (Algorithm 5.4). In addition to the unfolded input tensor, DBTF-TK also partitions and caches the input tensor (Algorithm 5.9).

5.4.6.2 Updating Factor Matrices and a Core Tensor

DBTF-CP and DBTF-TK initialize L sets of factor matrices (line 6 in Algorithm 5.3 and line 7 in Algorithm 5.8, respectively). Instead of initializing a single set of factor matrices, DBTF allows initializing multiple sets as better initial factor matrices could lead to more accurate factorization. DBTF-CP updates all of them in the first iteration and runs the following iterations with the factor matrices that obtained the smallest error (lines 7–8 in Algorithm 5.3). After initializing factor matrices, DBTF-TK also prepares core tensors for each set of factor matrices (line 9 in Algorithm 5.8) and finds the set of factor matrices

Algorithm 5.3: DBTF-CP Algorithm

Input: a three-way binary tensor $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$, rank R, the maximum number of iterations T, the number of sets of initial factor matrices L, the number of partitions N, and a threshold value V to limit the size of a single cache table.
Output: binary factor matrices $\mathbf{A} \in \mathbb{B}^{I \times R}$, $\mathbf{B} \in \mathbb{B}^{J \times R}$, and $\mathbf{C} \in \mathbb{B}^{K \times R}$.

```
1  pX(1) ← PartitionUnfoldedTensor(X(1), N)
2  pX(2) ← PartitionUnfoldedTensor(X(2), N)
3  pX(3) ← PartitionUnfoldedTensor(X(3), N)
4  for t ← 1, ..., T do
5      if t = 1 then
6          initialize L sets of factor matrices (A1, B1, C1), ..., (AL, BL, CL) randomly
             where Ai ∈ B^{I×R}, Bi ∈ B^{J×R}, and Ci ∈ B^{K×R} for i = 1, 2, ..., L
7          apply UpdateFactors to each set, and find the set s_min with the smallest error
8          (A, B, C) ← s_min
9      else
10         (A, B, C) ← UpdateFactors(A, B, C)
11     if converged then
12         break out of for loop
13 return A, B, C
14 Function UpdateFactors(A, B, C)
       /* minimize |X(1) − A ⊠ (C ⊙ B)^⊤| */
15     A ← UpdateFactorCP(pX(1), A, C, B, V)
       /* minimize |X(2) − B ⊠ (C ⊙ A)^⊤| */
16     B ← UpdateFactorCP(pX(2), B, C, A, V)
       /* minimize |X(3) − C ⊠ (B ⊙ A)^⊤| */
17     C ← UpdateFactorCP(pX(3), C, B, A, V)
18     return A, B, C
```

Algorithm 5.4: `PartitionUnfoldedTensor`

Input: an unfolded binary tensor $\mathbf{X} \in \mathbb{B}^{P \times Q}$, and the number of partitions N.
Output: a partitioned unfolded tensor ${}_p\mathbf{X} \in \mathbb{B}^{P \times Q}$.

distributed (D): split $\mathbf{X}$ into non-overlapping partitions $p_1, p_2, \ldots, p_N$ such that $[p_1\ p_2\ \ldots\ p_N] \in \mathbb{B}^{P \times Q}$, and $\forall i \in \{1, ..., N\}, p_i \in \mathbb{B}^{P \times H}$ where $\left\lfloor \frac{Q}{N} \right\rfloor \leq H \leq \left\lceil \frac{Q}{N} \right\rceil$

```
2  pX ← [p1 p2 ... pN]
3  D: foreach p' ∈ pX do
4      further split p' into a set of blocks divided by the boundaries of underlying pointwise
         vector-matrix products as depicted in Figure 5.6 (see Section 5.4.5)
5  D: cache pX across machines
6  return pX
```

and a core tensor with the smallest error (lines 10–11 in Algorithm 5.8). In each iteration, factor matrices are updated one at a time, while the other two are fixed (lines 15–17 in Algorithm 5.3 and lines 19–21 in Algorithm 5.8). In DBTF-TK, the core tensor is also updated before factor matrices are updated (line 13 in Algorithm 5.8).

Updating a Factor Matrix. The procedures for updating a factor matrix are shown in Algorithm 5.5 (DBTF-CP) and Algorithm 5.10 (DBTF-TK). Note that their core operations—computing a Boolean row summation and its error—are performed in a fully distributed manner (lines 7–9 in Algorithm 5.5, and lines 7–11 in Algorithm 5.10). DBTF caches combinations of Boolean row summations of a factor matrix (Algorithm 5.6) at the beginning of `UpdateFactorCP` and `UpdateFactorTK` to avoid repeatedly computing them. DBTF-TK additionally caches the combinations of row summation results of an unfolded core tensor (line 2 in Algorithm 5.10). DBTF-CP and DBTF-TK fetch the cached Boolean summation results in an almost identical manner, except that they use different cache keys (line 7 in Algorithm 5.5, and line 9 in Algorithm 5.10). DBTF collects errors computed across machines and updates the current column DBTF is visiting (lines 10–14 in Algorithm 5.5, and lines 12–16 in Algorithm 5.10). Boolean factors are repeatedly updated until convergence, that is, until the reconstruction error does not decrease significantly, or a maximum number of iterations has been reached.

Updating a Core Tensor. The procedure for updating a core tensor is given in Algorithm 5.11. DBTF-TK computes the partial gain of flipping an element of a core tensor $\mathcal{G}$ in a fully distributed fashion (lines 4–25 in Algorithm 5.11) and determines its value using the sum of collected gains (lines 26–27 in Algorithm 5.11).

5.4.6.3 Network Transfer

In DBTF-CP and DBTF-TK, the following data are sent to each machine: Partitions of unfolded tensors are distributed across machines once in the beginning, and factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are broadcast to each machine at each iteration. DBTF-TK sends out further data: Partitions of an input tensor are distributed once in the beginning, a core tensor is transferred when factor matrices are updated, and the rowwise sum of entries in factor matrices is distributed when a core tensor is updated.

In both DBTF-CP and DBTF-TK, machines send intermediate errors back to the driver node for the update of columns of a factor matrix. In DBTF-TK, each machine additionally sends the partial gain back to the driver node in order to update a core tensor.

5.4.7 Implementation

In this section, we discuss practical issues pertaining to the implementation of DBTF on Spark. We use sparse representation for tensors and matrices, storing only non-zero elements, except for those factor matrices to which we apply Boolean AND operation to compute a cache key, which we represent as an array of *BitSet*. An input tensor is loaded as an RDD (*Resilient Distributed Datasets*) [ZCD+12], and unfolded using RDD's *map* function. We apply *map* and *combineByKey* operations to unfolded tensors for partitioning: *map* transforms an unfolded tensor into a pair RDD whose key is a partition

Algorithm 5.5: `UpdateFactorCP`

Input: a partitioned unfolded tensor ${}_p\mathbf{X} \in \mathbb{B}^{P \times QS}$, factor matrices $\mathbf{A} \in \mathbb{B}^{P \times R}$ (factor matrix to update), $\mathbf{M}_f \in \mathbb{B}^{Q \times R}$ (first matrix for the Khatri-Rao product), and $\mathbf{M}_s \in \mathbb{B}^{S \times R}$ (second matrix for the Khatri-Rao product), a threshold value V to limit the size of a single cache table, and the maximum percentage Z of zeros in the column being updated.

Output: an updated factor matrix $\mathbf{A}$.

`AugmentPartitionWithRowSummations`$({}_p\mathbf{X}, \mathbf{M}_s, V)$
`/* iterate over columns and rows of` $\mathbf{A}$ `*/`
for *column iter* $c \leftarrow 1 \dots R$ **do**
  for *row* $r \leftarrow 1 \dots P$ **do**
    for $a_{rc} \leftarrow 0, 1$ **do**
      distributed: foreach *partition* $p' \in {}_p\mathbf{X}$ **do**
        foreach *block* $b \in p'$ **do**
          compute the cache key $\mathbf{k} \leftarrow \mathbf{a}_{r:} \wedge [\mathbf{M}_f]_{i:}$ where i is the row index of $\mathbf{M}_f$ such that block b is within the vertical boundaries of underlying $([\mathbf{M}_f]_{i:} \odot \mathbf{M}_s)^\top$
          $\mathbf{v} \leftarrow$ using $\mathbf{k}$, fetch the cached Boolean row summation that corresponds to $\mathbf{a}_{r:} \boxtimes ([\mathbf{M}_f]_{i:} \odot \mathbf{M}_s)^\top$
          compute the error between the fetched row $\mathbf{v}$ and the corresponding part of ${}_p\mathbf{x}_{r:}$
  collect errors for the entries of column $\mathbf{a}_{:c}$ from all blocks (for both cases of when each entry is set to 0 and 1)
  for *row* $r \leftarrow 1 \dots P$ **do** `/* update` $\mathbf{a}_{:c}$ `*/`
    update a_{rc} to the value that yields a smaller error (i.e., $\left|\mathbf{x}_{r:} - \mathbf{a}_{r:} \boxtimes (\mathbf{M}_f \odot \mathbf{M}_s)^\top\right|$)
  if *the percentage of zeros in* $\mathbf{a}_{:c} > Z$ **then**
    find new values for a subset of rows which will make $\mathbf{a}_{:c}$ to obey Z with the smallest increase in error, and update those rows accordingly.
return $\mathbf{A}$

Algorithm 5.6: `AugmentPartitionWithRowSummations`

Input: a partitioned unfolded tensor ${}_p\mathbf{X} \in \mathbb{B}^{P \times QS}$, a matrix for caching $\mathbf{M}_c \in \mathbb{B}^{S \times R}$, and a threshold value V to limit the size of a single cache table.

distributed: foreach *partition* $p' \in {}_p\mathbf{X}$ **do**
  $\mathbf{T}_i \leftarrow$ `GenerateRowSummations`$(\mathbf{M}_c, V)$
  foreach *block* $b \in p'$ **do**
    if block b is of the type (1), (2), or (4) as shown in Figure 5.6, vertically slice $\mathbf{T}_i$ such that the sliced one corresponds to block b
    cache (the sliced) $\mathbf{T}_i$ if not cached, and augment partition p' with it

Algorithm 5.7: `GenerateRowSummations`

Input: a matrix for caching $\mathbf{M}_c \in \mathbb{B}^{S \times R}$, and a threshold value V to limit the size of a single cache table.

Output: a table $\mathbf{T}_m$ that contains mappings from a set of rows in $\mathbf{M}_c$ to its summation result

1 $\mathbf{T}_m \leftarrow$ all combinations of row summations of $\mathbf{M}_c$ (if $S > V$, divide the rows of $\mathbf{M}_c$ evenly into smaller groups of rows, and generate combinations of row summations from each one separately)

2 **return** $\mathbf{T}_m$

ID; *combineByKey* groups non-zeros by partition ID and organizes them into blocks. In Tucker factorization, we similarly use *map* and *groupByKey* operations, to divide the input tensor RDD into partitions, in which non-zeros are organized as a set, since DBTF-TK queries the existence of tensor entries in updating the core tensor. Partitioned unfolded tensor RDDs and the input tensor RDD are then *persist*ed in memory. We create a pair RDD containing combinations of row summations, which is keyed by partition ID and *join*ed with the partitioned unfolded tensor RDD. This *join*ed RDD is processed in a distributed manner using *mapPartitions* operation. In obtaining the key to the table for row summations, we use bitwise AND operation for efficiency. At the end of column-wise iteration, a driver node *collect*s errors computed from each partition to update columns. DBTF-TK updates the core tensor entry by entry. In each iteration, executors process the partitioned input tensor in parallel using *foreachPartition* operation, computes the partial gains of flipping the current core tensor entry, and aggregates them using an *accumulator*. In order to upper bound the maximum percentage of zeros in the columns of the factor matrix being updated, we use a priority queue in which an inverse of the error of each candidate value (binary values assigned to the entries in each row which belong to the column being updated) is used as a priority. While the percentage of zeros is greater than the threshold, an element with the highest priority is popped off the priority queue and replaces the corresponding, current value provided that it decreases the percentage of zeros.

5.4.8 Analysis

We analyze the proposed method in terms of time complexity, memory requirement, and the amount of shuffled data. We use the following symbols in the analysis: R (rank or the dimension of each mode of a core tensor), M (number of machines), T (number of maximum iterations), N (number of partitions), V (maximum number of rows for caching, and Z (maximum percentage of zeros in the columns being updated). For the sake of simplicity, we assume an input tensor $\mathcal{X} \in \mathbb{B}^{I \times I \times I}$ and a core tensor $\mathcal{G} \in \mathbb{B}^{R \times R \times R}$, and that DBTF initializes a single set of factor matrices and a core tensor. Also, based on symbol definitions, we make simplifying assumptions that $N \ll I$, $R \ll I$, and $R^2 \leq I$.

Algorithm 5.8: DBTF-TK Algorithm

Input: a three-way binary tensor $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$, dimensions of a core tensor R_1, R_2, and R_3, the maximum number of iterations T, the number of sets of initial factor matrices L, the number of partitions N, and a threshold value V to limit the size of a single cache table.

Output: binary factor matrices $\mathbf{A} \in \mathbb{B}^{I \times R_1}$, $\mathbf{B} \in \mathbb{B}^{J \times R_2}$, and $\mathbf{C} \in \mathbb{B}^{K \times R_3}$, and a core tensor $\mathcal{G} \in \mathbb{B}^{R_1 \times R_2 \times R_3}$.

$_p\mathbf{X}_{(1)} \leftarrow$ `PartitionUnfoldedTensor`$(\mathbf{X}_{(1)}, N)$
$_p\mathbf{X}_{(2)} \leftarrow$ `PartitionUnfoldedTensor`$(\mathbf{X}_{(2)}, N)$
$_p\mathbf{X}_{(3)} \leftarrow$ `PartitionUnfoldedTensor`$(\mathbf{X}_{(3)}, N)$
$_p\mathcal{X} \leftarrow$ `PartitionInputTensor`$(\mathcal{X}, N)$
for $t \leftarrow 1, \ldots, T$ **do**
  if $t = 1$ **then**
    initialize L sets of factor matrices $(\mathbf{A}_1, \mathbf{B}_1, \mathbf{C}_1), \ldots, (\mathbf{A}_L, \mathbf{B}_L, \mathbf{C}_L)$ randomly where $\mathbf{A}_i \in \mathbb{B}^{I \times R_1}, \mathbf{B}_i \in \mathbb{B}^{J \times R_2},$ and $\mathbf{C}_i \in \mathbb{B}^{K \times R_3}$ for $i = 1, 2, \ldots, L$
    initialize L sets of core tensors $\mathcal{G}_1, \ldots, \mathcal{G}_L$ randomly
    $\mathcal{G}_i \leftarrow$ `UpdateCore`$(_p\mathcal{X}, \mathcal{G}_i, \mathbf{A}_i, \mathbf{B}_i, \mathbf{C}_i)$ for $i = 1, 2, \ldots, L$
    apply `UpdateFactors` to each set $(\mathbf{A}_i, \mathbf{B}_i, \mathbf{C}_i, \mathcal{G}_i)$ for $i = 1, 2, \ldots, L$, and find the set s_{min} with the smallest error
    $(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathcal{G}) \leftarrow s_{min}$
  else
    $\mathcal{G} \leftarrow$ `UpdateCore`$(_p\mathcal{X}, \mathcal{G}, \mathbf{A}, \mathbf{B}, \mathbf{C})$
    $(\mathbf{A}, \mathbf{B}, \mathbf{C}) \leftarrow$ `UpdateFactors`$(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathcal{G})$
  if *converged* **then**
    break out of **for** loop
return $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathcal{G}$
Function `UpdateFactors`$(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathcal{G})$
  `/* minimize` $\left|\mathbf{X}_{(1)} - \mathbf{A} \boxtimes \mathbf{G}_{(1)} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top\right|$ `*/`
  $\mathbf{A} \leftarrow$ `UpdateFactorTucker`$(_p\mathbf{X}_{(1)}, \mathbf{A}, \mathbf{C}, \mathbf{B}, \mathbf{G}_{(1)}, V)$
  `/* minimize` $\left|\mathbf{X}_{(2)} - \mathbf{B} \boxtimes \mathbf{G}_{(2)} \boxtimes (\mathbf{C} \otimes \mathbf{A})^\top\right|$ `*/`
  $\mathbf{B} \leftarrow$ `UpdateFactorTucker`$(_p\mathbf{X}_{(2)}, \mathbf{B}, \mathbf{C}, \mathbf{A}, \mathbf{G}_{(2)}, V)$
  `/* minimize` $\left|\mathbf{X}_{(3)} - \mathbf{C} \boxtimes \mathbf{G}_{(3)} \boxtimes (\mathbf{B} \otimes \mathbf{A})^\top\right|$ `*/`
  $\mathbf{C} \leftarrow$ `UpdateFactorTucker`$(_p\mathbf{X}_{(3)}, \mathbf{C}, \mathbf{B}, \mathbf{A}, \mathbf{G}_{(3)}, V)$
  return $\mathbf{A}, \mathbf{B}, \mathbf{C}$

5.4.8.1 Analysis of DBTF-CP

All proofs of the following lemmas appear in Section 5.7.

Lemma 5.4. Time Complexity of DBTF-CP:
The time complexity of DBTF-CP is $O\big(TI^3R\lceil\frac{R}{V}\rceil + TN\lceil\frac{R}{V}\rceil 2^{\lceil R/\lceil R/V\rceil\rceil} I\big)$.

Lemma 5.5. Memory Requirement of DBTF-CP:
The memory requirement of DBTF-CP is $O(|\mathcal{X}| + NI\lceil\frac{R}{V}\rceil 2^{\lceil R/\lceil R/V\rceil\rceil} + MRI)$.

Algorithm 5.9: `PartitionInputTensor`

Input: a three-way binary tensor $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$, and the number of partitions N.
Output: a partitioned input tensor ${}_p\mathcal{X} \in \mathbb{B}^{I \times J \times K}$.

1 **distributed (D):** split $\mathcal{X}$ into ${}_p\mathcal{X}$, which consists of non-overlapping subtensors ${}_p\mathcal{X}_1, {}_p\mathcal{X}_1, \ldots, {}_p\mathcal{X}_N$ where (1) ${}_p\mathcal{X}_t$ is associated with three ranges, I_t, J_t, and K_t, such that $|I_i| \times |J_i| \times |K_i| \approx |I_j| \times |J_j| \times |K_j|$ for all $i, j \in [1 \,..\, N]$; (2) ${}_p\mathcal{X}_t$ contains all $x_{ijk} \in \mathcal{X}$ for $i \in I_t, j \in J_t$, and $k \in K_t$; and (3) $\bigcup_{t=1}^{N} {}_p\mathcal{X}_t = \mathcal{X}$ and ${}_p\mathcal{X}_i \cap {}_p\mathcal{X}_j = \emptyset$ for all $i, j \in [1 \,..\, N]$ $(i \neq j)$
2 **D:** cache ${}_p\mathcal{X}$ across machines
3 **return** ${}_p\mathcal{X}$

Lemma 5.6. Shuffled Data for Partitioning in DBTF-CP:
The amount of shuffled data for partitioning an input tensor $\mathcal{X}$ is $O(|\mathcal{X}|)$.

Lemma 5.7. Shuffled Data After Partitioning in DBTF-CP:
The amount of shuffled data after the partitioning of an input tensor $\mathcal{X}$ is $O(TRI(M+N))$.

5.4.8.2 Analysis of DBTF-TK

All proofs of the following lemmas appear in Section 5.7.

Lemma 5.8. Time Complexity of DBTF-TK:
The time complexity of DBTF-TK is $O\big(TI^3R^3 + TN\lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V \rceil \rceil} I\big)$.

Lemma 5.9. Memory Requirement of DBTF-TK:
The memory requirement of DBTF-TK is $O(|\mathcal{X}| + (N+M)I\lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V \rceil \rceil})$.

Lemma 5.10. Shuffled Data for Partitioning in DBTF-TK:
The amount of shuffled data for partitioning an input tensor $\mathcal{X}$ is $O(|\mathcal{X}|)$.

Lemma 5.11. Shuffled Data After Partitioning in DBTF-TK:
The amount of shuffled data after the partitioning of an input tensor $\mathcal{X}$ is $O(TRI(M + N) + TR^3(MI + N) + TMR^2\lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V \rceil \rceil})$.

5.5 Experiments

In this section, we experimentally evaluate our proposed method DBTF. We aim to answer the following questions.

- **Q1 Data Scalability (Section 5.5.2).** How well do DBTF and other methods scale up with respect to the following aspects of an input tensor: number of non-zeros, dimensionality, density, and rank?
- **Q2 Machine Scalability (Section 5.5.3).** How well does DBTF scale up with respect to the number of machines?
- **Q3 Reconstruction Error (Section 5.5.4).** How accurately do DBTF and other methods factorize the given tensor?

Algorithm 5.10: `UpdateFactorTucker`

Input: a partitioned unfolded tensor ${}_p\mathbf{X} \in \mathbb{B}^{P \times QS}$, factor matrices $\mathbf{A} \in \mathbb{B}^{P \times R}$ (factor matrix to update), $\mathbf{M}_f \in \mathbb{B}^{Q \times R_f}$ (first matrix for the Kronecker product), and $\mathbf{M}_s \in \mathbb{B}^{S \times R_s}$ (second matrix for the Kronecker product), an unfolded core tensor $\mathbf{G}$, a threshold value V to limit the size of a single cache table, and the maximum percentage Z of zeros in the column being updated.

Output: an updated factor matrix $\mathbf{A}$.

`AugmentPartitionWithRowSummations`$({}_p\mathbf{X}, \mathbf{M}_s, V)$
$\mathbf{T}_g \leftarrow$ `GenerateRowSummations`$(\mathbf{G}, V)$
`/* iterate over columns and rows of` $\mathbf{A}$ `*/`
for *column iter* $c \leftarrow 1 \dots R$ **do**
  for *row* $r \leftarrow 1 \dots P$ **do**
    for $a_{rc} \leftarrow 0, 1$ **do**
      distributed: foreach *partition* $p' \in {}_p\mathbf{X}$ **do**
        $\mathbf{g} \leftarrow$ using $\mathbf{a}_{r:}$, fetch the cached Boolean row summation from $\mathbf{T}_g$ that corresponds to $\mathbf{a}_{r:} \boxtimes \mathbf{G}$
        foreach *block* $b \in p'$ **do**
          compute the cache key $\mathbf{k} \leftarrow \bigvee_{m=1}^{R_f} [\mathbf{M}_f]_{im} \wedge \mathbf{g}_{((m-1)R_s+1:mR_s)}$ where i is the row index of $\mathbf{M}_f$ such that block b is within the vertical boundaries of underlying $([\mathbf{M}_f]_{i:} \otimes \mathbf{M}_s)^\top$
          $\mathbf{v} \leftarrow$ using $\mathbf{k}$, fetch the cached Boolean row summation that corresponds to $[\mathbf{a}_{r:} \boxtimes \mathbf{G}] \boxtimes ([\mathbf{M}_f]_{i:} \otimes \mathbf{M}_s)^\top$
          compute the error between the fetched row $\mathbf{v}$ and the corresponding part of ${}_p\mathbf{x}_{r:}$
  collect errors for the entries of columns $\mathbf{a}_{:c}$ from all blocks (for both cases of when each entry is set to 0 and 1)
  for *row* $r \leftarrow 1 \dots P$ **do** `/* update` $\mathbf{a}_{:c}$ `*/`
    update a_{rc} to the value that yields a smaller error (i.e., $\left|\mathbf{x}_{r:} - \mathbf{a}_{r:} \boxtimes \mathbf{G} \boxtimes (\mathbf{M}_f \otimes \mathbf{M}_s)^\top\right|$)
  if *the percentage of zeros in* $\mathbf{a}_{:c} > Z$ **then**
    find new values for a subset of rows which will make $\mathbf{a}_{:c}$ to obey Z with the smallest increase in error, and update those rows accordingly.
return $\mathbf{A}$

We introduce the datasets, baselines, and experimental environment in Section 5.5.1. After that, we answer the above questions in Sections 5.5.2 to 5.5.4.

5.5.1 Experimental Settings

5.5.1.1 Datasets

We use both real-world and synthetic tensors to evaluate the proposed method. The tensors used in experiments are listed in Table 5.3. For real-world tensors, we use

Algorithm 5.11: `UpdateCore`

Input: a partitioned input tensor ${}_p\mathcal{X} \in \mathbb{B}^{I \times J \times K}$, a core tensor $\mathcal{G} \in \mathbb{B}^{R_1 \times R_2 \times R_3}$, and factor matrices $\mathbf{A} \in \mathbb{B}^{I \times R_1}$, $\mathbf{B} \in \mathbb{B}^{J \times R_2}$, and $\mathbf{C} \in \mathbb{B}^{K \times R_3}$.

Output: an updated core tensor $\mathcal{G}$.

```
1  for (r1, r2, r3) ∈ [1 .. R1] × [1 .. R2] × [1 .. R3] do
2      gain ← 0
3      s_I, s_J, s_K ← rowwise sum of entries in A, B, and C
         that are in those columns selected by entries in G
4      distributed: foreach subtensor _pX_t ∈ _pX do
5          I_t, J_t, K_t ← three ranges associated with _pX_t
6          if g_{r1 r2 r3} = 0 then
7              foreach i ∈ I_t such that a_{i r1} = 1 do
8                  c_I ← s_I(i) = 0
9                  foreach j ∈ J_t such that b_{j r2} = 1 do
10                     c_IJ ← c_I or s_J(j) = 0
11                     foreach k ∈ K_t such that c_{k r3} = 1 do
12                         c_IJK ← c_IJ or s_K(k) = 0
13                         if c_IJK and _px_{ijk} = 1 then
14                             gain ← gain + 1
15                             break out of all foreach loops
16         else
17             foreach i ∈ I_t such that a_{i r1} = 1 do
18                 if s_I(i) ≠ 1 then continue
19                 foreach j ∈ J_t such that b_{j r2} = 1 do
20                     if s_J(j) ≠ 1 then continue
21                     foreach k ∈ K_t such that c_{k r3} = 1 do
22                         if s_K(k) ≠ 1 then continue
23                         if _px_{ijk} = 0 then
24                             gain ← gain + 1
25                             break out of all foreach loops
26     if gain > 0 then
27         g_{r1 r2 r3} ← 1 − g_{r1 r2 r3}
28 return G
```

Facebook, DBLP, CAIDA-DDoS-S, CAIDA-DDoS-L, NELL-S, and NELL-L. Facebook[1] is temporal relationship data between users. DBLP[2] is a record of DBLP publications. CAIDA-DDoS[3] datasets are traces of network attack traffic. NELL datasets are knowledge

[1] http://socialnetworks.mpi-sws.org/data-wosn2009.html
[2] http://www.informatik.uni-trier.de/~ley/db/
[3] http://www.caida.org/data/passive/ddos-20070804_dataset.xml

Table 5.3: Summary of real-world and synthetic tensors used for experiments. B: billion, M: million, K: thousand.

Name	I	J	K	Non-Zeros
Facebook	64K	64K	870	1.5M
DBLP	418K	3.5K	50	1.3M
CAIDA-DDoS-S	9K	9K	4K	22M
CAIDA-DDoS-L	9K	9K	393K	331M
NELL-S	15K	15K	29K	77M
NELL-L	112K	112K	213K	18M
Synthetic-scalability	2^6∼2^{13}	2^6∼2^{13}	2^6∼2^{13}	26K∼5.5B
Synthetic-CP-error	100	100	100	6.5K∼240K
Synthetic-TK-error	100	100	100	1.5K∼45K

base tensors. S (small) and L (large) suffixes indicate the relative size of the dataset.

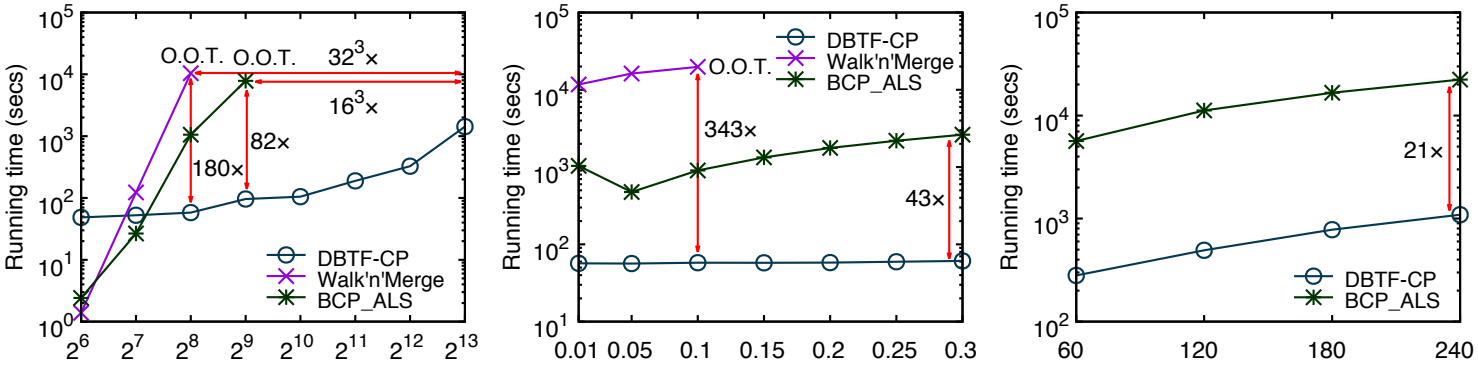
We prepare two different sets of synthetic tensors, one for scalability tests and another for reconstruction error tests. For scalability tests, we generate random tensors, varying the following aspects: (1) dimensionality and (2) density. We vary one aspect while fixing others to see how scalable DBTF and other methods are with respect to a particular aspect. For reconstruction error tests, we generate three random factor matrices, construct a noise-free tensor from them, and then add noise to this tensor, while varying the following aspects: (1) factor matrix density, (2) rank, (3) additive noise level, and (4) destructive noise level. When we vary one aspect, others are fixed. The amount of noise is determined by the number of 1's in the noise-free tensor. For example, 10% additive noise indicates that we add 10% more 1's to the noise-free tensor, and 5% destructive noise means that we delete 5% of the 1's from the noise-free tensor.

5.5.1.2 Baselines

In experiments for Boolean CP factorization, we compare DBTF-CP with Walk'n'Merge [EM13b] and BCP_ALS [Mie11]. We also implemented an algorithm for Boolean CP decomposition of three-way binary data presented in [BGV13], but found its results to be much worse than Walk'n'Merge and BCP_ALS (e.g., it takes three orders of magnitude more time than BCP_ALS and Walk'n'Merge for a tensor of size $I{=}J{=}K{=}2^7$). So we omit reporting its results for the sake of clarity. In experiments for Boolean Tucker factorization, we compare DBTF-TK with Walk'n'Merge [EM13b] and BTucker_ALS [Mie11].

5.5.1.3 Environment

DBTF is implemented on the Apache Spark framework. We run experiments on a cluster with 17 machines, each of which is equipped with an Intel Xeon E3-1240v5 CPU (quad-core with hyper-threading at 3.50GHz) and 32GB RAM. The cluster runs Apache Spark v2.2.0, and consists of a driver node and 16 worker nodes. In the experiments for DBTF, we use 16 executors, and each executor uses 8 cores. The amount of memory for the

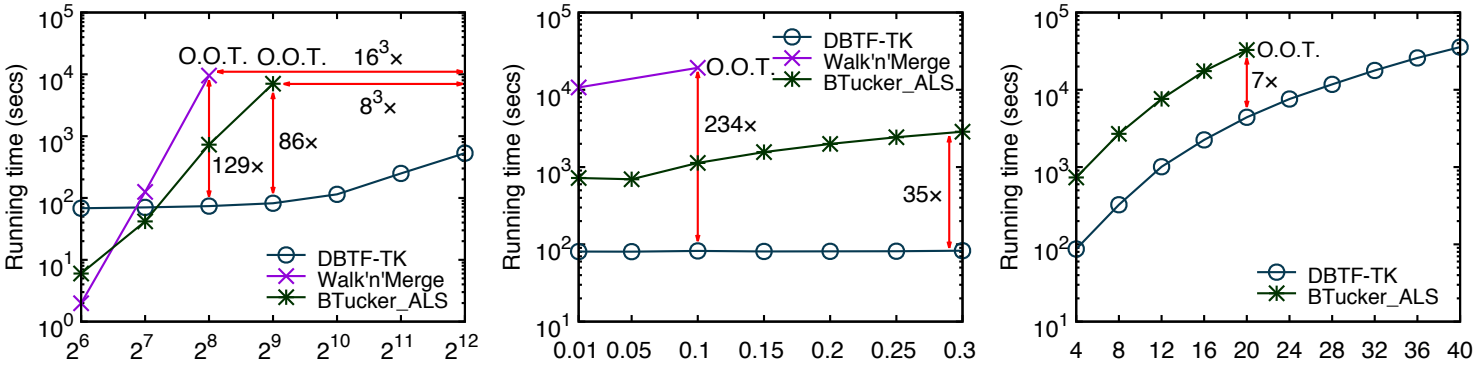


(a) Dimensionality. (b) Density. (c) Rank.

Figure 5.7: The scalability of DBTF-CP and other methods with respect to the dimensionality and density of a tensor, and the rank of CP decomposition. O.O.T.: Out Of Time (takes more than 6 hours). DBTF-CP decomposes up to 16^3–$32^3\times$ larger tensors than existing methods in 82–180$\times$ less time (Figure 5.7a). Overall, DBTF-CP achieves 21–343$\times$ speedup and exhibits near-linear scalability with regard to all data aspects.

driver and each executor process is set to 16GB and 25GB, respectively. DBTF parameters L, V, and Z are set to 1, 15, and 0.95, respectively, and T is set to 10 for scalability tests and 20 for reconstruction error tests (see Algorithms 5.3, 5.5 to 5.8 and 5.10 for details on these parameters). We run Walk'n'Merge, BCP_ALS, and BTucker_ALS on one machine in the cluster. For the CP factorization by Walk'n'Merge, we use the original implementation[4] provided by the authors. However, the open-source implementation of Walk'n'Merge does not contain code for the Tucker factorization, which is obtained based on the CP factorization output of Walk'n'Merge. We implement the missing part, which is to merge the factor matrices returned from Walk'n'Merge and adjust the core tensor accordingly. We run Walk'n'Merge with the same parameter settings as described in [EM13b]. The minimum size of blocks is set to 4-by-4-by-4. The length of random walks is set to 5. In reconstruction error tests, the merging threshold t is set to $1-(n_d+0.05)$ for CP factorization, and it is set to $1-(n_d+0.5)$ for Tucker factorization where n_d is the destructive noise level of an input tensor. Since Walk'n'Merge did not find blocks when it performs Tucker factorization with the threshold value used for CP factorization, we used smaller values for Tucker factorization. For scalability tests, t is set to 0.2 for both types of factorizations. Other parameters are set to the default values. We implement BCP_ALS and BTucker_ALS using the open-source code of ASSO[5][MMG+08]. For ASSO, the threshold value for discretization is set to 0.7; default values are used for other parameters.

[4] http://people.mpi-inf.mpg.de/~pmiettin/src/walknmerge.zip
[5] http://people.mpi-inf.mpg.de/~pmiettin/src/DBP-progs/



(a) Dimensionality. (b) Density. (c) Core Size $R_1 = R_2 = R_3$.

Figure 5.8: The scalability of DBTF-TK and other methods with respect to the dimensionality and density of a tensor, and the core size of Tucker decomposition. O.O.T.: Out Of Time (takes more than 12 hours). DBTF-TK decomposes up to 8^3–$16^3\times$ larger tensors than existing methods in 86–129× less time (Figure 5.8a). Overall, DBTF-TK achieves 7–234× speedup and exhibits near-linear scalability with regard to all data aspects.

5.5.2 Data Scalability

We evaluate the data scalability of DBTF and other methods for Boolean CP and Tucker factorizations using both synthetic random tensors (Sections 5.5.2.1 and 5.5.2.2) and real-world tensors (Sections 5.5.2.3 and 5.5.2.4).

5.5.2.1 Boolean CP Factorization on Synthetic Data

We evaluate the data scalability of DBTF-CP, Walk'n'Merge, and BCP_ALS on synthetic tensors with respect to the dimensionality, density, and rank of a tensor. Experiments are allowed to run for up to 6 hours, and those running longer than that are marked as O.O.T. (Out Of Time).

Dimensionality. We increase the dimensionality $I{=}J{=}K$ of each mode from 2^6 to 2^{13}, while setting the tensor density to 0.01 and the rank R to 10. As shown in Figure 5.7a, DBTF-CP successfully decomposes tensors of size $I{=}J{=}K{=}2^{13}$, while Walk'n'Merge and BCP_ALS run out of time when $I{=}J{=}K \geq 2^9$ and $\geq 2^{10}$, respectively. Notice that the running time of Walk'n'Merge and BCP_ALS increases rapidly with the dimensionality: DBTF-CP decomposes the largest tensors Walk'n'Merge and BCP_ALS can process $180\times$ and $82\times$ faster than each method. DBTF-CP is slower than other methods for small tensors of 2^6 and 2^7 scale, because the overhead of running a distributed algorithm on Spark (e.g., code and data distribution, network I/O latency, etc) dominates the running time in these cases.

Density. We increase the tensor density from 0.01 to 0.3, while fixing $I{=}J{=}K$ to 2^8 and the rank R to 10. As shown in Figure 5.7b, DBTF-CP decomposes tensors of all densities and exhibits near constant performance regardless of the density. BCP_ALS also scales up to 0.3 density. On the other hand, Walk'n'Merge runs out of time when

the density increases over 0.1. In terms of running time, DBTF-CP runs $343\times$ faster than Walk'n'Merge, and $43\times$ faster than BCP_ALS. Also, the performance gap between DBTF-CP and BCP_ALS grows wider for tensors with greater density.

Rank. We increase the rank R of a tensor from 60 to 240, while fixing $I{=}J{=}K$ to 2^8 and the tensor density to 0.01. V is set to 15 in all experiments. Walk'n'Merge is excluded from this experiment since its running time is constant across different ranks. As shown in Figure 5.7c, both methods scale up to rank 240, and the running time increases almost linearly as the rank increases. When the rank is 240, DBTF-CP is $21\times$ faster than BCP_ALS.

5.5.2.2 Boolean Tucker Factorization on Synthetic Data

We evaluate the data scalability of DBTF-TK, Walk'n'Merge, and BTucker_ALS. Experiments that run longer than 12 hours are marked as O.O.T. (Out Of Time).

Dimensionality. We increase the dimensionality $I{=}J{=}K$ of each mode from 2^6 to 2^{12} while setting the tensor density to 0.01 and the core size $R_1{=}R_2{=}R_3{=}4$ (Figure 5.8a). While DBTF-TK is slower than BTucker_ALS and Walk'n'Merge for small tensors of 2^6 and 2^7 scale due to the overhead associated with a distributed system, the running time of BTucker_ALS and Walk'n'Merge increases much more rapidly than that of DBTF-TK. As a result, DBTF-TK is the only method that successfully decomposes tensors of $I{=}J{=}K{=}2^{12}$, while Walk'n'Merge and BTucker_ALS run out of time when $I{=}J{=}K \geq 2^9$ and $\geq 2^{10}$, respectively. Furthermore, DBTF-TK decomposes the largest tensors that Walk'n'Merge and BTucker_ALS can handle $129\times$ and $86\times$ faster than each method.

Density. We increase the tensor density from 0.01 to 0.3, while fixing $I{=}J{=}K$ to 2^8 and the core size $R_1{=}R_2{=}R_3$ to 4. Figure 5.8b shows that DBTF-TK decomposes tensors of all densities, and its running time remains almost the same as the density increases. BTucker_ALS also scales up to the tensor with 0.3 density. However, Walk'n'Merge runs out of time when the density becomes greater than 0.1, and even when it is 0.05. In terms of running time, DBTF-TK runs $234\times$ and $35\times$ faster than Walk'n'Merge and BTucker_ALS, respectively.

Rank. We increase the core size $R_1{=}R_2{=}R_3$ from 4 to 40, while fixing $I{=}J{=}K$ to 2^8 and the tensor density to 0.01. V is set to 15 in all experiments. As shown in Figure 5.8c, DBTF-TK scales up to the largest core size, while BTucker_ALS fails to scale up to core size greater than 20. Note that Walk'n'Merge is not shown in the figure since it does not allow users to specify the core size; instead, it automatically determines the core size according to the MDL principle. In terms of running time, DBTF-TK is faster than BTucker_ALS for all core sizes, with DBTF-TK being $7\times$ faster than BTucker_ALS when core size $R_1{=}R_2{=}R_3$ is 20.

While DBTF-TK outperforms all baselines, the largest core size $R_1{=}R_2{=}R_3 = 40$ for Tucker factorization is much smaller than the largest rank size $R = 240$ used for CP factorization. This is because Tucker factorization is much more expensive than CP factorization as it involves all steps of CP factorization, and also performs steps to update

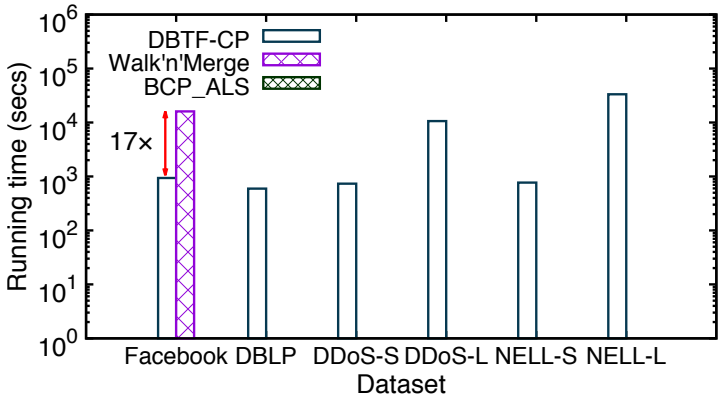


Figure 5.9: The scalability of DBTF-CP and other methods on the real-world datasets. Notice that only DBTF-CP scales up to all datasets, while Walk'n'Merge processes only Facebook, and BCP_ALS fails to process all datasets. DBTF-CP runs $17\times$ faster than Walk'n'Merge on Facebook. An empty bar denotes that the corresponding method runs out of time (> 12 hours) or memory while decomposing the dataset.

a core tensor, which is the most costly operation that takes time proportional to the cube of core size $R_1{=}R_2{=}R_3$ (see Lemma 5.8).

5.5.2.3 Boolean CP Factorization on Real-World Data

Figure 5.9 shows the running time of DBTF-CP, Walk'n'Merge, and BCP_ALS on real-world datasets. We set the maximum running time to 12 hours, and R to 10 for DBTF-CP and BCP_ALS. Among three methods, DBTF-CP is the only one that scales up for all datasets. Walk'n'Merge decomposes only Facebook and runs out of time for all other datasets; BCP_ALS fails to handle real-world tensors as it causes out-of-memory errors for all datasets, except for DBLP for which BCP_ALS runs out of time. Also, DBTF-CP runs $17\times$ faster than Walk'n'Merge on Facebook.

5.5.2.4 Boolean Tucker Factorization on Real-World Data

Figure 5.10 reports the running time of DBTF-TK, Walk'n'Merge, and BTucker_ALS on real-world tensors. We run each experiment for at most 12 hours with $R_1{=}R_2{=}R_3 = 4$. In Figure 5.10, only DBTF-TK is shown as it is the only method that scales up to all datasets. While Walk'n'Merge finds blocks within the time limit for Facebook data, it runs out of time while merging factors and adjusting the core tensor. BTucker_ALS runs out of time for DBLP and causes out-of-memory errors for all other datasets.

5.5.3 Machine Scalability

We measure the machine scalability of DBTF-CP and DBTF-TK by increasing the number of machines from 4 to 16 and report T_4/T_M where T_M is the running time using M machines.

Boolean CP Factorization. We use the synthetic tensor of size $I{=}J{=}K{=}2^{12}$ and of

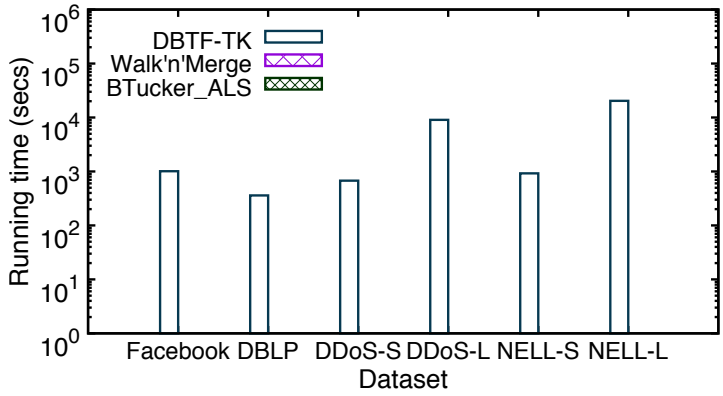
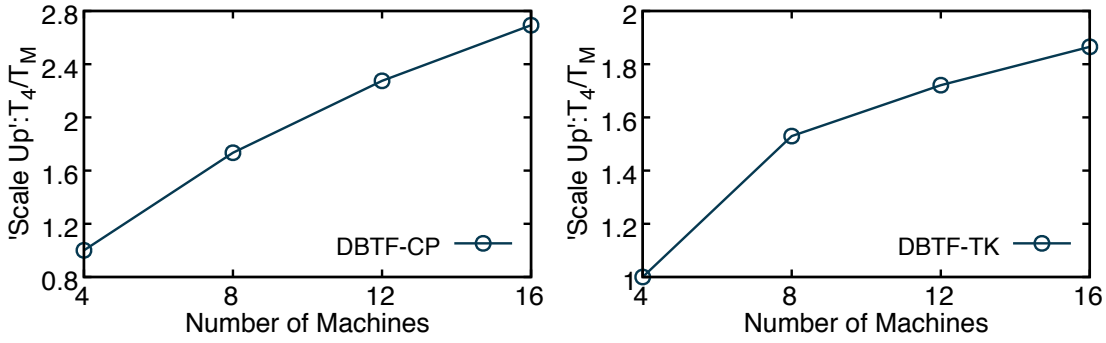


Figure 5.10: The scalability of DBTF-TK and other methods on the real-world datasets. Notice that only DBTF-TK scales up to all datasets, while Walk'n'Merge and BTucker_ALS fail to process all datasets as they run out of time (> 12 hours) or memory while decomposing the dataset.



(a) CP Factorization.

(b) Tucker Factorization.

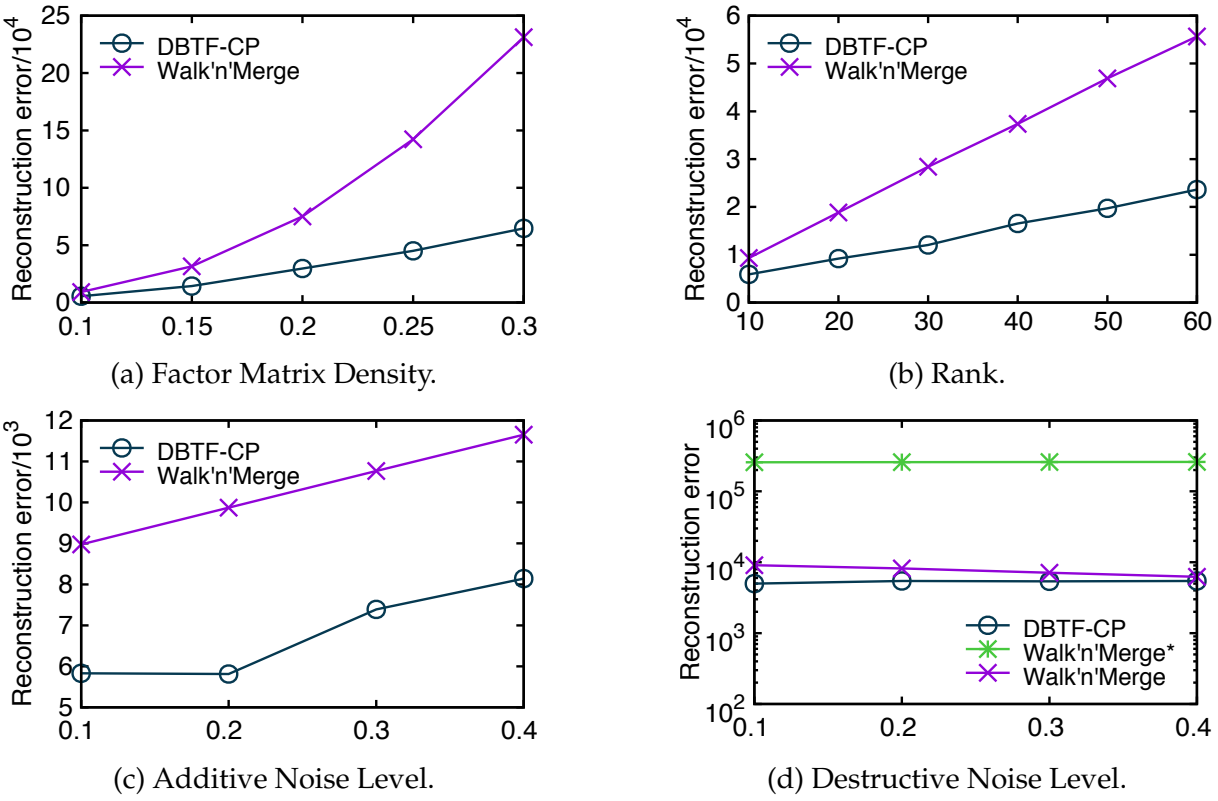
Figure 5.11: The scalability of DBTF-CP and DBTF-TK with respect to the number of machines. T_M means the running time using M machines. Notice that the running time scales up near linearly.

density 0.01; we set the rank R to 10. Figure 5.11a shows that DBTF-CP scales up near linearly. Overall, DBTF-CP achieves $2.69\times$ speedup, as the number of machines increases fourfold.

Boolean Tucker Factorization. We use the synthetic tensor of size $I{=}J{=}K{=}2^{11}$ and of density 0.01; we set the core size $R_1{=}R_2{=}R_3$ to 4. Figure 5.11b shows that DBTF-TK shows near-linear scalability, achieving $1.87\times$ speedup when the number of machines is increased from 4 to 16.

5.5.4 Reconstruction Error

We evaluate the accuracy of DBTF in terms of reconstruction error, which is defined as $|\mathcal{X} - \mathcal{X}'|$ where $\mathcal{X}$ is an input tensor and $\mathcal{X}'$ is a reconstructed tensor. Tensors of size



(a) Factor Matrix Density.

(b) Rank.

(c) Additive Noise Level.

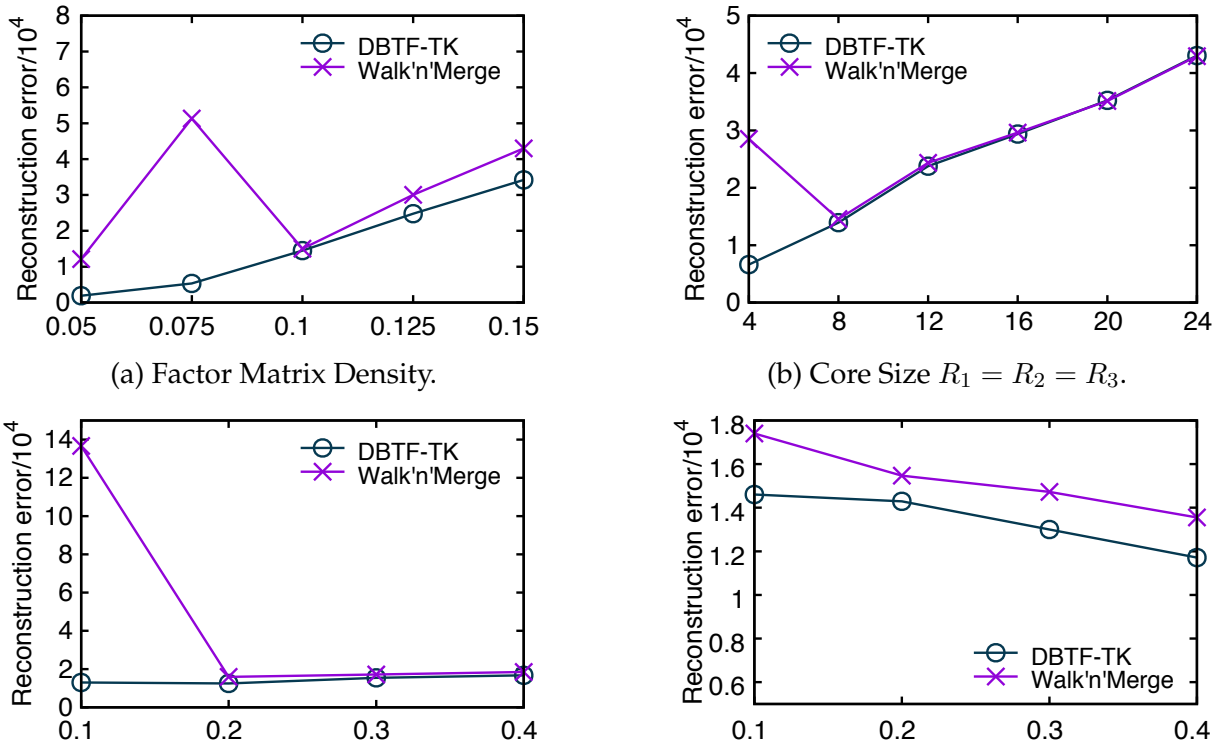
(d) Destructive Noise Level.

Figure 5.12: The reconstruction error of DBTF-CP and other methods with respect to factor matrix density, rank, additive noise level, and destructive noise level. Walk'n'Merge* in d refers to the version of Walk'n'Merge which executes the merging phase. Notice that the reconstruction errors of DBTF-CP are smaller than those of Walk'n'Merge for all aspects.

$I{=}J{=}K{=}100$ are used in experiments. We run each configuration three times and report the average of the results. We compare DBTF with Walk'n'Merge as they take different approaches for Boolean CP and Tucker decompositions, and exclude BCP_ALS and BTucker_ALS as DBTF, BCP_ALS, and BTucker_ALS are based on the same Boolean decomposition frameworks.

5.5.4.1 Boolean CP Factorization

We measure reconstruction errors, varying one of the four different data aspects—factor matrix density (0.1), rank (10), additive noise level (0.1), and destructive noise level (0.1)—while fixing the others to the default values. The values in the parentheses are the default settings for each aspect. For Walk'n'Merge, we report the reconstruction error computed from the blocks obtained before the second part of the merging phase [EM13b], since the subsequent merging procedure significantly increased the reconstruction error when applied to our synthetic tensors. Figure 5.12d shows the difference between the version of Walk'n'Merge with the second part of merging procedure (Walk'n'Merge*)



(a) Factor Matrix Density.

(b) Core Size $R_1 = R_2 = R_3$.

(c) Additive Noise Level.

(d) Destructive Noise Level.

Figure 5.13: The reconstruction error of DBTF-TK and other methods with respect to factor matrix density, core size, additive noise level, and destructive noise level. Across all aspects, the reconstruction errors of DBTF-TK are smaller than or close to those of Walk'n'Merge. Note that, in contrast to DBTF-TK, Walk'n'Merge shows performance fluctuations in a few cases.

and the one without it (Walk'n'Merge).

Factor Matrix Density. We increase the density of factor matrices from 0.1 to 0.3. As shown in Figure 5.12a, the reconstruction error of DBTF-CP is smaller than that of Walk'n'Merge for all densities. In particular, as the density increases, the gap between DBTF-CP and Walk'n'Merge widens.

Rank. We increase the rank of a tensor from 10 to 60. As shown in Figure 5.12b, the reconstruction errors of both methods increase in proportion to the rank. This is an expected result since, given a fixed density, the increase in the rank of factor matrices leads to increased number of non-zeros in the input tensor. Notice that the reconstruction error of DBTF-CP is smaller than that of Walk'n'Merge for all ranks.

Additive Noise Level. We increase the additive noise level from 0.1 to 0.4. As shown in Figure 5.12c, the reconstruction errors of both methods increase in proportion to the additive noise level. While the relative accuracy improvement obtained with DBTF-CP tends to decrease as the noise level increases, the reconstruction error of DBTF-CP is

smaller than that of Walk'n'Merge for all additive noise levels.

Destructive Noise Level. We increase the destructive noise level from 0.1 to 0.4. Figure 5.12d shows that DBTF-CP produces more accurate results than Walk'n'Merge across all destructive noise levels. As the destructive noise level increases, the reconstruction error of DBTF-CP slightly increases, while that of Walk'n'Merge decreases; as a result, the gap between two methods becomes smaller. Destructive noise makes the factorization harder by sparsifying tensors and introducing noises at the same time.

5.5.4.2 Boolean Tucker Factorization

We measure reconstruction errors, varying one of the following data aspects—factor matrix density (0.1), core size (R_1=R_2=R_3=8), additive noise level (0.2), and destructive noise level (0.2)—while fixing the others to their default values. The values in the parentheses are the default settings for each aspect.

Factor Matrix Density. We increase the density of factor matrices from 0.05 to 0.15. As shown in Figure 5.13a, the reconstruction error of DBTF-TK is smaller than or close to that of Walk'n'Merge across all densities. Note that, in contrast to DBTF-TK, Walk'n'Merge shows performance fluctuation when the density is 0.075.

Rank. We increase the core size $R_1 = R_2 = R_3$ from 4 to 24. As shown in Figure 5.13b, the reconstruction errors of both methods increase in proportion to the core size, and DBTF-TK and Walk'n'Merge exhibit similar performance.

Additive Noise Level. We increase the additive noise level from 0.1 to 0.4. Figure 5.13c shows that both methods perform similarly as the noise level increases, except when the additive noise level is 0.1, in which case the reconstruction error of Walk'n'Merge is approximately $10\times$ greater than that of DBTF-TK.

Destructive Noise Level. We increase the destructive noise level from 0.1 to 0.4. Figure 5.13d shows that the reconstruction errors of both methods decrease as the noise level is increased, and DBTF-TK is consistently more accurate than Walk'n'Merge for all destructive noise levels.

5.6 Conclusion

In this chapter, we propose DBTF, a distributed method for Boolean CP (DBTF-CP) and Tucker (DBTF-TK) factorizations running on the Apache Spark framework. By distributed data generation with minimal network transfer, exploiting the characteristics of Boolean operations, and with careful partitioning, DBTF successfully tackles the high computational costs and minimizes the intermediate data. Experimental results show that DBTF-CP decomposes up to 16^3–$32^3\times$ larger tensors than existing methods in 82–$180\times$ less time, and DBTF-TK decomposes up to 8^3–$16^3\times$ larger tensors than existing methods in 86–$129\times$ less time. Furthermore, both DBTF-CP and DBTF-TK exhibit near-linear scalability in terms of tensor dimensionality, density, rank, and the number of machines.

5.7 Appendix

5.7.1 Proof of Lemma 5.4

Proof. Algorithm 5.3 is composed of three operations: (1) partitioning (lines 1–3), (2) initialization (line 6), and (3) updating factor matrices (lines 7 and 10).

(1) After unfolding an input tensor $\mathcal{X}$ into $\mathbf{X}$, DBTF-CP splits $\mathbf{X}$ into N partitions, and further divides each partition into a set of blocks (Algorithm 5.4). Unfolding takes $O(|\mathcal{X}|)$ time as each entry can be mapped in constant time (Equation (5.1)), and partitioning takes $O(|\mathbf{X}|)$ time since determining which partition and block an entry of $\mathbf{X}$ belongs to is also a constant-time operation. It takes $O(|\mathcal{X}|)$ time in total.
(2) Random initialization of factor matrices takes $O(IR)$ time.
(3) The update of a factor matrix (Algorithm 5.5) consists of the following steps (i, ii, iii, and iv):

i. Caching row summations of a factor matrix (line 1). By Lemma 5.2, the number of cache tables is $\lceil R/V \rceil$, and the maximum size of a single cache table is $2^{\lceil R/\lceil R/V \rceil \rceil}$. Each row summation can be obtained in $O(I)$ time via incremental computations that use prior row summation results. Hence, caching row summations for N partitions takes $O(N\lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V \rceil \rceil} I)$.
ii. Fetching cached row summations (lines 7–8). The number of constructing row summations and computing errors to update a factor matrix is $2IR$. An entire row summation is constructed by fetching row summations from the cache tables $O(\max(I, N))$ times across N partitions. If $R \leq V$, a row summation can be constructed by a single access to the cache. If $R > V$, multiple accesses are required to fetch row summations from $\lceil \frac{R}{V} \rceil$ tables. Also, constructing a cache key requires $O(\min(V, R))$ time. Thus, fetching a cached row summation takes $O(\lceil \frac{R}{V} \rceil \min(V, R) \max(I, N))$ time. When $R > V$, there is an additional cost to sum up $\lceil \frac{R}{V} \rceil$ row summations, which is $O((\lceil \frac{R}{V} \rceil - 1)I^2)$. In total, the time complexity for this step is $O(IR \left[\lceil \frac{R}{V} \rceil \min(V, R) \max(I, N) + (\lceil \frac{R}{V} \rceil - 1)I^2 \right])$. Simplifying terms, we get $O(I^3 R \lceil \frac{R}{V} \rceil)$.
iii. Computing the error for the fetched row summation (line 9). It takes $O(I^2)$ time to calculate an error of one row summation with regard to the corresponding row of the unfolded tensor. For each column entry, DBTF-CP constructs row summations ($\mathbf{a}_{r:} \boxtimes (\mathbf{M}_f \odot \mathbf{M}_s)^\top$ in Algorithm 5.5) twice (for $a_{rc} = 0$ and 1). Therefore, given a rank R, this step takes $O(I^3 R)$ time.
iv. Updating a factor matrix (lines 10–14). Updating an entry in a factor matrix requires summing up errors for each value collected from N partitions, which takes $O(N)$ time. Updating all entries takes $O(NIR)$ time. In case the percentage of zeros in the column being updated is greater than Z, an additional step is performed to make the sparsity of the column less than Z, which takes $O(I \log(I))$ time as all $2I$ values may need to be fetched in the order of increasing error in the worst case. Since we have R columns, this additional step takes $O(RI \log(I))$ in total. Thus, step iv takes $O(NIR + RI \log(I))$ time.

After simplifying terms, DBTF-CP's time complexity is $O\big(TI^3 R \lceil \frac{R}{V} \rceil + TN \lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V \rceil \rceil} I\big)$.

At each iteration, the dominating term is $O(I^3R)$ that comes from fetching row summation and calculating its error (steps ii and iii), which is an $O(I^2)$ operation that is performed $2IR$ times. Note that the worst-case time complexity for this error calculation is $O(I^2)$ even when the input tensor is sparse because the time for this operation depends not only on the non-zeros in the row of an input tensor, but also on the non-zeros in the corresponding row of the intermediate matrix product (e.g., $(\mathbf{C} \odot \mathbf{B})^\top$), which could be full of non-zeros in the worst case. However, given sparse tensors in practice, factor matrices are updated to be sparse such that the reconstructed tensor gets closer to the sparse input tensor, which makes the time required for the dominating operation much less than $O(I^2)$.

■

5.7.2 Proof of Lemma 5.5

Proof. For the decomposition of an input tensor $\mathcal{X} \in \mathbb{B}^{I \times I \times I}$, DBTF-CP stores the following four types of data in memory at each iteration: (1) partitioned unfolded input tensors ${}_p\mathbf{X}_{(1)}, {}_p\mathbf{X}_{(2)}$, and ${}_p\mathbf{X}_{(3)}$, (2) row summation results, (3) factor matrices $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$, and (4) errors for the entries of a column being updated.

(1) While partitioning of an unfolded tensor by DBTF-CP structures it differently from the original one, the total number of elements does not change after partitioning. Thus, ${}_p\mathbf{X}_{(1)}, {}_p\mathbf{X}_{(2)}$, and ${}_p\mathbf{X}_{(3)}$ require $O(|\mathcal{X}|)$ memory.
(2) By Lemma 5.2, the total number of row summations of a factor matrix is $O(\lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V \rceil \rceil})$. By Lemma 5.3, each partition has at most three types of blocks. Since an entry in the cache table uses $O(I)$ space, the total amount of memory used for row summation results is $O(NI\lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V \rceil \rceil})$. Note that since Boolean factor matrices are normally sparse, many cached row summations are not normally dense. Therefore, the actual amount of memory used is usually smaller than the stated upper bound.
(3) Since $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$ are broadcast to each machine, they require $O(MRI)$ memory in total.
(4) Each partition stores two errors for the entries of the column being updated, which takes $O(NI)$ memory.

■

5.7.3 Proof of Lemma 5.6

Proof. DBTF-CP unfolds an input tensor $\mathcal{X}$ into three different modes, $\mathbf{X}_{(1)}$, $\mathbf{X}_{(2)}$, and $\mathbf{X}_{(3)}$, and then partitions each one: unfolded tensors are shuffled across machines so that each machine has a specific range of consecutive columns of unfolded tensors. In the process, the entire data can be shuffled, depending on the initial distribution of the data. Thus, the amount of data shuffled for partitioning $\mathcal{X}$ is $O(|\mathcal{X}|)$. ■

5.7.4 Proof of Lemma 5.7

Proof. Once the three unfolded input tensors $\mathbf{X}_{(1)}$, $\mathbf{X}_{(2)}$, and $\mathbf{X}_{(3)}$ are partitioned, they are cached across machines, and are not shuffled. In each iteration, DBTF-CP broadcasts three factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ to each machine, which takes $O(MRI)$ space in sum. With only these three matrices, each machine generates the part of row summation it needs to process. Also, in updating a factor matrix of size I-by-R, DBTF-CP collects from all partitions the errors for both cases of when each entry of the factor matrix is set to 0 and 1. This process involves transmitting $2IR$ errors from each partition to the driver node, which takes $O(NIR)$ space in total. Accordingly, the total amount of data shuffled for T iterations after partitioning $\mathcal{X}$ is $O(TRI(M+N))$. ■

5.7.5 Proof of Lemma 5.8

Proof. Algorithm 5.8 is composed of four operations: (1) partitioning (lines 1–4), (2) initialization (lines 7–8), (3) updating factor matrices (lines 10 and 14), and (4) updating a core tensor (lines 9 and 13).

(1) Partitioning of an input tensor $\mathcal{X}$ into ${}_p\mathbf{X}_{(1)}$, ${}_p\mathbf{X}_{(2)}$, and ${}_p\mathbf{X}_{(3)}$ (lines 1–3) takes $O(|\mathcal{X}|)$ time as in DBTF-CP. Similarly, partitioning of $\mathcal{X}$ into ${}_p\mathcal{X}$ (line 4) takes $O(|\mathcal{X}|)$ time since determining which partition an entry of $\mathcal{X}$ belongs to can be done in constant time.

(2) Randomly initializing factor matrices and a core tensor takes $O(IR)$ and $O(R^3)$ time, respectively.

(3) The update of a factor matrix (Algorithm 5.10) consists of the following steps (i, ii, iii, and iv):

i. Caching row summations (lines 1–2). Caching row summations of a factor matrix takes $O(N\lceil\frac{R}{V}\rceil 2^{\lceil R/\lceil R/V\rceil\rceil}I)$ as in DBTF-CP. Caching row summations of an unfolded core tensor requires $O(\lceil\frac{R}{V}\rceil 2^{\lceil R/\lceil R/V\rceil\rceil}R^2)$ as a single row summation can be computed in $O(R^2)$ time. Assuming $R^2 \leq I$, this step requires $O(N\lceil\frac{R}{V}\rceil 2^{\lceil R/\lceil R/V\rceil\rceil}I)$ time.

ii. Fetching cached row summations (lines 9–10). (a) Row summations of an unfolded core tensor are fetched $O(IR)$ times in each partition. If $R \leq V$, a row summation can be obtained with one access to the cache. If $R > V$, multiple accesses are required to fetch row summations from $\lceil\frac{R}{V}\rceil$ tables, and there is an additional cost to sum up $\lceil\frac{R}{V}\rceil$ row summations. In sum, this operation takes $O(NIR[\lceil\frac{R}{V}\rceil + (\lceil\frac{R}{V}\rceil - 1)R^2])$ time. (b) Fetching a cached row summation of a factor matrix is identical to that in DBTF-CP, except for the computation of cache key, which takes $O(R^2)$ time. Therefore, this operation takes $O(IR[\lceil\frac{R}{V}\rceil R^2 \max(I,N) + (\lceil\frac{R}{V}\rceil - 1)I^2])$ in total. Simplifying (a) and (b) under the assumption that $R^2 \leq I$ and $\max(I,N) = I$, the time complexity for this step reduces to $O(I^3R\lceil\frac{R}{V}\rceil)$.

iii. Computing the error for the fetched row summation (line 11). This step takes the same time as in DBTF-CP, which is $O(I^3R)$.

iv. Updating a factor matrix (lines 12–16). This step takes the same time as in DBTF-CP, which is $O(NIR + RI\log(I))$.

(4) For the update of a core tensor (Algorithm 5.11), two operations are repeatedly performed for each core tensor entry. First, rowwise sum of entries in factor matrices are computed (line 3), which takes $O(IR)$ time. Second, DBTF-TK determines whether flipping the core tensor entry would improve accuracy (lines 4–25). This step takes $O(I^3)$ time in the worst case when the factor matrices are full of non-zeros. In sum, it takes $O(I^3R^3)$ to update a core tensor.

In sum, DBTF-TK takes $O\big(TI^3R^3 + TN\lceil\frac{R}{V}\rceil 2^{\lceil R/\lceil R/V\rceil\rceil} I\big)$ time. ■

5.7.6 Proof of Lemma 5.9

Proof. In order to decompose an input tensor $\mathcal{X} \in \mathbb{B}^{I\times I\times I}$, DBTF-TK stores the following five types of data in memory at each iteration: (1) partitioned input tensors ${}_p\mathbf{X}_{(1)}, {}_p\mathbf{X}_{(2)}$, ${}_p\mathbf{X}_{(3)}$, and ${}_p\mathcal{X}$, (2) row summation results, (3) a core tensor $\mathcal{G}$, (4) factor matrices $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$, and (5) errors for the entries of a column being updated.

(1) Since partitioning does not change the total number of elements, ${}_p\mathbf{X}_{(1)}, {}_p\mathbf{X}_{(2)}, {}_p\mathbf{X}_{(3)}$, and ${}_p\mathcal{X}$ require $O(|\mathcal{X}|)$ memory.
(2) Two types of row summation results are maintained in DBTF-TK: the first for the factor matrix (e.g., $\mathbf{B}^\top$), and the second for the unfolded core tensors (e.g., $\mathbf{G}_{(1)}$). Note that, given R number of rows, the total number of row summations to be cached is $O(\lceil\frac{R}{V}\rceil 2^{\lceil R/\lceil R/V\rceil\rceil})$ by Lemma 5.2. First, the cache tables for the factor matrix are the same as those used in DBTF-CP; thus, they use $O(NI\lceil\frac{R}{V}\rceil 2^{\lceil R/\lceil R/V\rceil\rceil})$ memory. Second, across M machines, the cache tables for the unfolded core tensor require $O(MR^2\lceil\frac{R}{V}\rceil 2^{\lceil R/\lceil R/V\rceil\rceil})$ as a single entry uses $O(R^2)$ space. Assuming $R^2 \leq I$, $O((N+M)I\lceil\frac{R}{V}\rceil 2^{\lceil R/\lceil R/V\rceil\rceil})$ memory is required in total for row summation results.
(3) Since the core tensor $\mathcal{G}$ is broadcast to each machine, $O(MR^3)$ is required.
(4) Factor matrices require $O(MRI)$ memory as in DBTF-CP.
(5) $O(NI)$ memory is required since each partition stores two errors for each entry of the column being updated as in DBTF-CP.

■

5.7.7 Proof of Lemma 5.10

Proof. In DBTF-TK, an input tensor $\mathcal{X}$ is partitioned in four different ways, where the first three are ${}_p\mathbf{X}_{(1)}, {}_p\mathbf{X}_{(2)}$, and ${}_p\mathbf{X}_{(3)}$ that are used for updating factor matrices, and the last one is ${}_p\mathcal{X}$ that is used for updating a core tensor. Each machine is assigned non-overlapping partitions of the input tensor. The entire data can be shuffled in the worst case, depending on the data distribution. Thus, the total amount of data shuffled for partitioning $\mathcal{X}$ is $O(|\mathcal{X}|)$. ■

5.7.8 Proof of Lemma 5.11

Proof. As in DBTF-CP, partitioned input tensors ${}_p\mathbf{X}_{(1)}$, ${}_p\mathbf{X}_{(2)}$, ${}_p\mathbf{X}_{(3)}$, and ${}_p\mathcal{X}$ are shuffled only once in the beginning. After that, DBTF-TK performs data shuffling at each iteration in order to update (1) factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, and (2) a core tensor $\mathcal{G}$.

(1) In updating factor matrices, DBTF-TK uses all data used in DBTF-CP, which is $O(TRI(M+N))$. Also, DBTF-TK broadcasts the tables containing the combinations of row summations of three unfolded core tensors ($\mathbf{G_{(1)}}, \mathbf{G_{(2)}}$, and $\mathbf{G_{(3)}}$) to each machine at every iteration. Since, given R, the total number of row summations to be cached is $O(\lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V \rceil \rceil})$, and each row summation uses $O(R^2)$ space, broadcasting these tables overall requires $O(TMR^2 \lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V \rceil \rceil})$.

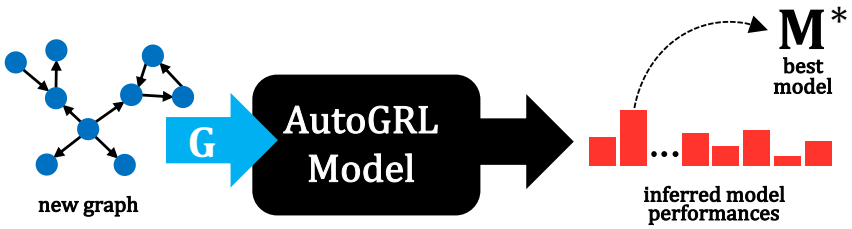
(2) DBTF-TK broadcasts the rowwise sum of entries in factor matrices to each machine when a core tensor $\mathcal{G}$ is updated, which takes $O(MIR^3)$ space in each iteration. Also, in updating an element of $\mathcal{G}$, DBTF-TK aggregates partial gains computed from each partition, which requires $O(NR^3)$ for each iteration.

Accordingly, the total amount of data shuffled for T iterations after partitioning $\mathcal{X}$ is $O(TRI(M+N) + TR^3(MI+N) + TMR^2 \lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V \rceil \rceil})$. ■

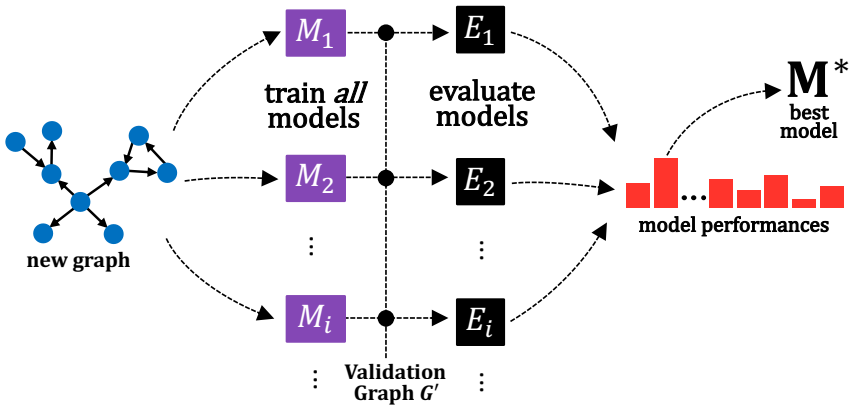
Chapter 6

Fast Automatic Model Selection for Graph Representation Learning

Given a graph learning task such as link prediction (LP) on a new graph dataset, how can we automatically select the best LP method as well as its hyperparameters (collectively called a model)? Model selection for graph learning has been largely ad hoc. A typical approach has been to apply popular methods to new datasets, but this is often suboptimal. Also, systematically comparing models on the new graph quickly becomes too costly, or even impractical. In this chapter, we develop the *first meta-learning approach for automatic graph representation learning*, called AUTOGRL, which automatically infers a good model for the new graph *without requiring any model training or evaluations*. AUTOGRL capitalizes on the prior performances of a large body of existing methods on benchmark graph datasets, and carries over this prior experience to automatically select the best model to use for the new graph. To capture the similarity across graphs from different domains, we introduce specialized structural meta-graph features that quantify the structural characteristics of a graph. Then we design a meta-graph that represents the relations among models and graphs, and develop a graph meta-learner operating on the meta-graph, which estimates the relevance of each model to different graphs. Through extensive experiments, we show that using AUTOGRL to select a method for the new graph significantly outperforms consistently applying popular LP methods as well as several existing meta-learners, while being extremely fast at test time compared to the model selection based on exhaustive evaluation. While we use LP as the task of interest, AUTOGRL is naturally applicable to other graph learning tasks, *e.g.*, node classification and edge regression, among many others.



(a) AUTOGRL is a meta-learned GRL model selection approach.



(b) Costly naive approach for selection of the best GRL model.

Figure 6.1: Overview of AUTOGRL compared to existing naive approach. Given an unseen graph G and a large space of models $\mathcal{M}$ to search over, AUTOGRL efficiently infers the best model $M^* \in \mathcal{M}$ without ever having to train a single model from $\mathcal{M}$ on the new graph G. This is in contrast to first training each model $M \in \mathcal{M}$, evaluating each one on a hold-out dataset, and then selecting the best model. Notably, in practical settings where time and budget is limited and thus costly naive approach cannot be used, AUTOGRL is extremely effective as it infers the best model nearly instantaneously.

6.1 Introduction

Given a graph learning task such as link prediction on a new graph dataset, how can we automatically select the best method as well as its hyperparameters (collectively called a model), in particular, *without performing model training or evaluations on the new graph*? Graph learning (*i.e.*, machine learning on graphs) has been receiving increasing attention in recent years [XSY+21, ZCZ22], and has shown successes across a large array of applications, including traffic forecasting [JL21], recommendation [FML+19], ranking [PKD+19], bioinformatics [STZ+20], drug discovery [LCH17], and anomaly detection [CCL+21]. However, as more graph learning methods are developed for various tasks, it becomes increasingly difficult to determine which method, and also which hyperparameter settings to use for a given graph.

Selecting a method and its hyperparameters (*i.e.*, model selection) for graph learning has been largely ad hoc to date. A typical approach is to simply apply popular graph

learning models to new graphs, often with the default hyperparameter values. However, it is well known that there is no universal learning algorithm that performs the best on *all* problem instances [WM97], and such consistent model selection is often suboptimal. At the other extreme lies "naive model selection" (Figure 6.1b), where all candidate models are trained on the new graph data, and then evaluated on a hold-out validation graph, and finally, the best performing model for this new graph is selected. This approach is very costly in terms of the runtime and computational cost associated with training all possible models whenever a new graph arrives. Thus, it is highly impractical for use in the real-world where model selection needs to be done nearly instantaneously as new data continuously arrive. There exist smarter strategies, such as Bayesian hyperparameter optimization [SLA12, WCZ+19], to enable a more efficient model selection by carefully evaluating a relatively small number of hyperparameter configurations. However, these methods mainly focus on finding the best hyperparameter setting of a single learning algorithm, while our model space $\mathcal{M}$ includes a wide variety of learning algorithms and their hyperparameters in general. Also, evaluating even just a few hyperparameter settings of each method in $\mathcal{M}$ on a real-world graph easily takes several orders of magnitude more time and resources than "training and evaluation-free" model selection.

In this chapter, we tackle the model selection problem for graph learning systematically, focusing on link prediction, which is a representative graph learning task. To that end, we develop AUTOGRL, *the first automatic graph learning framework* to the best of our knowledge that selects an effective model to employ for the new graph *without requiring any model training or evaluation*, as depicted in Figure 6.1a. AUTOGRL is a meta-learning based approach that stands on the prior performances of a large body of existing graph learning methods on extensive benchmark graph datasets. The high-level idea of AUTOGRL is to estimate a candidate model's performance on the new graph based on its performances on *similar* existing graphs. Once AUTOGRL is trained, we can infer the best model for any unseen graph at a very low computational cost.

Our meta-learning problem for graphs requires learning similarities between graphs based on characteristic dataset features (namely meta-features). Note that this step is often not needed for traditional meta-learning problems that deal with non-graph data, as features for those non-graph objects (*e.g.*, features such as age, gender, location, etc for users) may often be readily available. Even when no input features are available, the task is much more challenging for graph data as graphs not only have different number of nodes and edges, but also have widely varying connectivity patterns. Moreover, the high complexity and irregularity of graphs make the construction of meta-features for graphs computationally more costly than for non-graph (*e.g.*, i.i.d., tabular) datasets. To handle these challenges, we design specialized *meta-graph features* that effectively characterize major structural properties of real-world graphs, and can be computed efficiently.

To estimate the model performance, AUTOGRL learns to embed models and graphs in the shared latent space such that their embeddings reflect the graph-to-model affinity. Specifically, we design a multi-relational graph called *meta-graph*, which represents the

relations among models and graphs, and develop a graph meta-learner operating on this meta-graph, which is optimized to leverage meta-graph features and prior model performances into producing model and graph embeddings that can be effectively used to estimate the best performing model for the given graph.

In summary, the key contributions are as follows.

- **Problem Formulation.** We formulate the problem of *training and evaluation-free* model selection for graph learning, where model space encompasses a large array of graph learning algorithms and their hyperparameter configurations.
- **Framework for Automatic Graph Learning.** We propose AUTOGRL, the first approach to automatic graph learning to the best of our knowledge, which infers the best graph learning model for a new unseen graph in near real-time, without ever having to run different models as done in traditional model selection. AUTOGRL draws on the prior performances of various existing models on benchmark graph datasets, and can be used for different graph learning tasks, *e.g.*, link prediction and node classification.
- **Specialized Meta-Graph Features.** We design specialized meta-graph features for meta-learning on graphs. The meta-graph features effectively capture structural characteristics of a graph, enabling an effective and efficient quantification of graph similarity.
- **Effectiveness and Efficiency.** Through extensive experiments on the benchmark environment that we have built, we show that using AUTOGRL to select a model for various new graphs performs significantly better than always employing popular state-of-the-art models, as well as several existing meta-learning techniques tailored for our problem setting. Furthermore, AUTOGRL is highly efficient, incurring negligible runtime overhead (<1 second) at inference time.

6.2 Problem Formulation

In this work, we consider the problem of fast automatic model selection for a new unseen graph from a set of heterogeneous graph learning models, without requiring model evaluations and user intervention—hence *fast* and *automatic*. In comparison to traditional meta-learning problems where a model denotes a single method and its associated hyperparameters, a model in the graph meta-learning problem is more broadly defined to be

$$\begin{aligned}\text{model } M = \{&(\text{graph embedding method}, \text{hyperparameters}),\\ &(\text{predictor}, \text{hyperparameters})\}\end{aligned}$$

as graph learning tasks normally involve two steps: (1) the graph is first flattened by embedding it into a lower-dimensional space using a graph representation learning (embedding) method, and (2) the node embeddings are then used as input into the predictor for the downstream application like link prediction. Both steps require learning a method with specific hyperparameters. Hence, there can be many models that use the same embedding method (and also the same predictor), but have different hyperparameters.

Given a training meta-corpus of n graph datasets $\boldsymbol{\mathcal{G}} = \{G_1, \ldots, G_n\}$, m models $\boldsymbol{\mathcal{M}} = \{M_1, \ldots, M_m\}$ for graph learning tasks, and ground truth labels Y in the case of supervised tasks, we derive performance matrix $\mathbf{P} \in \mathbb{R}^{n \times m}$ where P_{ij} is the performance (*e.g.*, accuracy, average precision[1]) of model j on graph i. Our graph meta-learning problem for fast automatic model selection is defined as follows.

Problem 6.1. Fast Automatic Selection of Graph Learning Models:
Given (i) an unseen test graph $G_{\text{test}} \notin \boldsymbol{\mathcal{G}}$, and (ii) a performance matrix $\mathbf{P} \in \mathbb{R}^{n \times m}$ of m models $\boldsymbol{\mathcal{M}} = \{M_1, \ldots, M_m\}$ on n graphs $\boldsymbol{\mathcal{G}} = \{G_1, \ldots, G_n\}$, **infer** the best model $M^* \in \boldsymbol{\mathcal{M}}$ to employ on G_{test} without training or evaluating any model in the model set $\boldsymbol{\mathcal{M}}$ and requiring user intervention.

6.3 Framework

In this section, we present AUTOGRL, our meta-learning based framework that solves Problem 6.1. AUTOGRL operates by leveraging prior performances of a large body of existing methods on benchmark graphs to efficiently and automatically select the best model for a new graph. AUTOGRL consists of the following two phases: (1) *offline meta-training* phase (Section 6.3.1) that trains a meta-learner using observed graphs $\boldsymbol{\mathcal{G}}$ and model performances $\mathbf{P}$, and (2) *online model prediction* phase (Section 6.3.2), which selects the model with the highest estimated performance on the new graph. A summary of notations used in this work is provided in Table 6.1.

6.3.1 Offline Meta-Training

Meta-learning leverages prior experience from related learning tasks to do a better job on the new task. When the new task is similar to some historical learning tasks, then the knowledge from those similar tasks can be transferred and applied to the new task. Thus effectively capturing the similarity between an input task and observed ones is a fundamentally important problem to be addressed for successful meta-learning. In meta-learning, the similarity between learning tasks is modeled using *meta-features*, *i.e.*, characteristic features of the learning task that can be used to quantify the task similarity.

Meta-Graph Features. Given the graph learning model selection problem (where new graphs correspond to new learning tasks), AUTOGRL captures the graph similarity by extracting *meta-graph features* such that they reflect the structural properties of the graph. Notably, since graphs have irregular structure, with different number of nodes and edges, AUTOGRL designs meta-graph features to be of the same size for any arbitrary graph such that they can be easily compared using meta-graph features. We use the symbol $\mathbf{m} \in \mathbb{R}^d$ to denote the fixed-size meta-graph feature vector for graph G, and defer the details of how AUTOGRL computes $\mathbf{m}$ to Section 6.3.3.

[1]The evaluation metric used to obtain the performance matrix $\mathbf{P}$ is completely interchangeable, and can be replaced with another metric of interest.

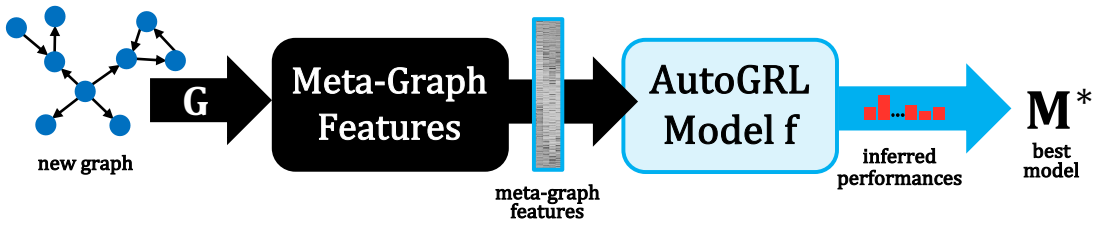


Figure 6.2: Given a new graph G, AUTOGRL extracts meta-graph features that capture the structural characteristics of graph G, and applies a meta-learned model to them, which efficiently infers the best model $M^* \in \boldsymbol{\mathcal{M}}$ for G, with no model evaluation.

Model Performance Estimation. To estimate how well a model would perform on a given graph, AUTOGRL represent models and graphs in the latent k-dimensional space, and captures the graph-to-model affinity using the dot product similarity between the two representations $\mathbf{h}_{G_i}$ and $\mathbf{h}_{M_j}$ of the i-th graph G_i and j-th model M_j, respectively, such that $p_{ij} \approx \langle \mathbf{h}_{G_i}, \mathbf{h}_{M_j} \rangle$ where p_{ij} is the performance of model M_j on graph G_i. Then to obtain the latent representation $\mathbf{h}$, we design a learnable function $f(\cdot)$ that takes in relevant information on models and graphs from the meta-graph features $\mathbf{m}$ and the prior knowledge (*i.e.*, model performances $\mathbf{P}$ and observed graphs $\boldsymbol{\mathcal{G}}$). Below in this section, we focus on the inputs to the function $f(\cdot)$, and defer the details of $f(\cdot)$ to Section 6.3.4.

We first factorize performance matrix $\mathbf{P}$ into latent graph factors $\mathbf{U} \in \mathbb{R}^{n \times k}$ and model factors $\mathbf{V} \in \mathbb{R}^{m \times k}$, and take the model factor $\mathbf{V}_j \in \mathbb{R}^k$ (the j-th row of $\mathbf{V}$) as the input representation of model M_j. Then, AUTOGRL obtains the latent embedding $\mathbf{h}_{M_j}$ of model M_j by $\mathbf{h}_{M_j} = f(\mathbf{V}_j)$. For graphs, more information is available since we have both meta-graph features $\mathbf{m}$ and meta-train graph factors $\mathbf{U}$. However, while we have the same number of models during training and inference, we observe new graphs during inference, and thus cannot obtain the graph factor $\mathbf{U}_{\text{test}}$ for the test graph as for the train graphs since matrix factorization (MF) is transductive by construction (*i.e.*, existing models' performance on the test graph is needed to get latent factors for the test graph directly via MF). We handle this issue by learning an estimator $\phi : \mathbb{R}^d \mapsto \mathbb{R}^k$ that maps the meta-graph features $\mathbf{m}$ into the latent factors of meta-train graphs obtained via MF above, *i.e.*, for graph G_i with $\mathbf{m}$, $\phi(\mathbf{m}) = \widehat{\mathbf{U}}_i \approx \mathbf{U}_i$, and use this estimated graph factor. We then combine both inputs ($[\mathbf{m}; \phi(\mathbf{m})] \in \mathbb{R}^{d+k}$), and apply linear transformation to make the input representation of graph G_i to be of the same size as that of model M_j, obtaining the latent embedding of graph G_i to be $\mathbf{h}_{G_i} = f(\mathbf{W}[\mathbf{m}; \phi(\mathbf{m})])$ where $\mathbf{W} \in \mathbb{R}^{k \times (d+k)}$ is a weight matrix. Thus in AUTOGRL, the performance p_{ij} of model M_j on graph G_i with meta-graph features $\mathbf{m}$ is estimated as

$$p_{ij} \approx \widehat{p}_{ij} = \langle f(\mathbf{W}[\mathbf{m}; \phi(\mathbf{m})]), f(\mathbf{V}_j) \rangle. \tag{6.1}$$

Meta-Learning Objective. For tasks where the goal is to estimate real values, such as accuracy, the mean squared error (MSE) is a typical choice for the loss function. While

Table 6.1: Summary of notations.

$\boldsymbol{\mathcal{G}}$	set of graphs $\{G_1, \ldots, G_n\}$ in the training set
n	number of graphs in training set $n = \lvert\boldsymbol{\mathcal{G}}\rvert$
G_{test}	new unseen test graph $G_{\text{test}} \notin \boldsymbol{\mathcal{G}}$
$\boldsymbol{\mathcal{M}}$	model set $\{M_1, \ldots, M_m\}$ to search over
m	number of models to search over $m = \lvert\boldsymbol{\mathcal{M}}\rvert$
Ψ	set of structural meta-node/edge feature extractors
Σ	set of meta-graph feature extractors
$\mathbf{M}$	meta-graph feature matrix where $\mathbf{M} \in \mathbb{R}^{n \times d}$
d	number of meta-graph features
$\mathbf{m}_{\text{test}}$	meta-graph feature vector for the new unseen test graph G_{test}
k	embedding size
$\mathbf{P}$	performance matrix of m models on n graphs
$\mathbf{U}$	latent graph factors obtained by factorizing $\mathbf{P}$ ($\mathbf{P} \approx \mathbf{U}\mathbf{V}^\intercal$)
$\mathbf{V}$	latent model factors obtained by factorizing $\mathbf{P}$ ($\mathbf{P} \approx \mathbf{U}\mathbf{V}^\intercal$)
$f(\cdot)$	learnable embedding function for models and graphs

MSE is easy to optimize and effective for regression, it does not directly concern with the ranking quality. On the other hand, in our problem setup, the goal is to accurately rank models for each graph, rather than estimating the performance itself, which makes MSE a suboptimal choice. In particular, model selection problem focuses on finding the model with the highest performance on the given graph. Therefore, among rank-based learning objectives, we choose to apply the top-1 probability [CQL+07] to our problem. Let $\widehat{\mathbf{P}}_i \in \mathbb{R}^m$ be the i-th row of $\widehat{\mathbf{P}}$ (*i.e.*, estimated performance of all m models on graph G_i's). Given $\widehat{\mathbf{P}}_i$, the top-1 probability $p_{\text{top1}}^{\widehat{\mathbf{P}}_i}(j)$ of j-th model M_j in the model set $\boldsymbol{\mathcal{M}}$ represents the probability of M_j to be ranked at the top of the list, *i.e.*, all models in $\boldsymbol{\mathcal{M}}$, given model performance $\widehat{\mathbf{P}}_i$, and is defined as

$$p_{\text{top1}}^{\widehat{\mathbf{P}}_i}(j) = \frac{\pi(\widehat{p}_{ij})}{\sum_{k=1}^{m} \pi(\widehat{p}_{ik})} = \frac{\exp(\widehat{p}_{ij})}{\sum_{k=1}^{m} \exp(\widehat{p}_{ik})}. \tag{6.2}$$

Here $\pi(\cdot)$ is a strictly increasing positive function, which we define to be an exponential function. Given that the top-1 probability $p_{\text{top1}}^{\widehat{\mathbf{P}}_i}(j)$ for all $j = 1, \ldots, m$ forms a probability distribution over all m models, we obtain two probability distributions by applying top-1 probability to the true performance $\mathbf{P}_i$ and estimated performance $\widehat{\mathbf{P}}_i$ of m models, and optimize AUTOGRL such that the distance between the two resulting distributions gets decreased. Using the cross entropy as the distance metric, we minimize the following loss over all n meta-train graphs $\boldsymbol{\mathcal{G}}$:

$$L(\mathbf{P}, \widehat{\mathbf{P}}) = -\sum_{i=1}^{n} \sum_{j=1}^{m} p_{\text{top1}}^{\mathbf{P}_i}(j) \log\left(p_{\text{top1}}^{\widehat{\mathbf{P}}_i}(j)\right) \tag{6.3}$$

6.3.2 Online Model Prediction

In the meta-training phase, AUTOGRL learns estimators $f(\cdot)$ and $\phi(\cdot)$, as well as weight matrix $\mathbf{W}$ and latent model factors $\mathbf{V}$. Given a new graph G_{test}, AUTOGRL first computes the meta-graph features $\mathbf{m}_{\text{test}} \in \mathbb{R}^d$ as we discuss in Section 6.3.3. Then $\mathbf{m}_{\text{test}}$ is regressed to obtain the (approximate) latent graph factors $\widehat{\mathbf{U}}_{\text{test}} = \phi(\mathbf{m}_{\text{test}}) \in \mathbb{R}^k$. Recall that the model factors $\mathbf{V}$ learned in the meta-training stage can be directly used for model prediction. Then model M_j's performance on test graph G_{test} can be estimated by applying Equation (6.1) with $\mathbf{m}_{\text{test}}$. Finally, the model that has the highest estimated performance is selected by AUTOGRL as the best model M^*, *i.e.*,

$$M^* \leftarrow \underset{M_j \in \mathcal{M}}{\arg\max} \left\langle f(\mathbf{W}[\mathbf{m}_{\text{test}}; \phi(\mathbf{m}_{\text{test}})]), f(\mathbf{V}_j) \right\rangle \tag{6.4}$$

Note that model selection using Equation (6.4) depends only on the meta-graph features $\mathbf{m}_{\text{test}}$ of the test graph and other pretrained estimators and latent factors that AUTOGRL learned in the meta-training phase. As no model training or evaluation is involved, model prediction by AUTOGRL is fast, taking negligible runtime compared to the time to train the selected model as our experiments show in Section 6.4.3. Further, model prediction process is fully automatic it does not require users to choose or fine-tune any values at test time. Figure 6.2 shows an overview of model prediction process, and Algorithm 6.1 lists steps for offline meta-training and online model prediction.

6.3.3 Structural Meta-Graph Features

Meta-graph features are a crucial component of our meta-learning approach AUTOGRL since they capture the important structural characteristics of an arbitrary graph dataset. Such meta-graph features enables AUTOGRL to quantify and leverage the similarity between meta-train graphs during training. It is important that a sufficient and representative set of graph meta-features are used to capture the important structural properties of graphs from a wide variety of different domains, including biological, technological, information, and social networks to name a few [RA15].

In this work, we are unable to leverage the commonly used and simple statistical meta-features used by previous work on model selection-based meta-learning, as these statistical functions cannot be used directly over irregular and complex graph datasets. To address this problem, we introduce the notion of meta-graph features and develop a general framework for computing them on any arbitrary unseen graph.

Meta-graph features in AUTOGRL are derived in two steps, which is shown in Figure 6.3. First, we apply a set of structural meta-feature extractors $\Psi = \{\psi_1, \ldots, \psi_q\}$ to the input graph G, obtaining $\Psi(G) = \{\psi_1(G), \ldots, \psi_q(G)\}$. Applying $\psi \in \Psi$ to G yields a vector or a distribution of values for the nodes (or edges) in the graph, such as degree distribution and PageRank scores. That is, in the example given in Figure 6.3, ψ_1 can be a degree distribution, ψ_2 can be PageRank scores of all nodes, and so on. Specifically, we use both local and global structural feature extractors. To capture the local structural properties around a node or an edge, we compute node degree, number of wedges, triangles

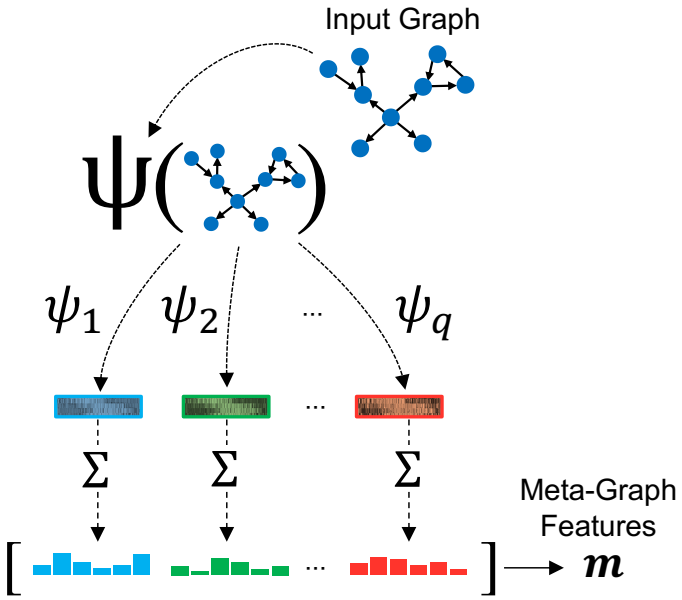


Figure 6.3: Meta-graph features in AUTOGRL are derived in two main steps: (1) Structural meta-feature extractors $\Psi = \{\psi_1, \ldots, \psi_q\}$ produce a set of $\Psi(G) = \{\psi_1(G), \ldots, \psi_q(G)\}$ structural feature matrices. (2) The set Σ of global statistical meta-graph extractors summarize $\Psi(G)$ into a fixed-size meta-graph feature vector $\mathbf{m}$. See Section 6.3.3 for more details.

centered at each node, and also frequency of triangles for every edge. To capture the global structural properties of a node, we derive the PageRank score, eccentricity, and k-core number of each node in the graph. We present more details of different sets of meta-feature extractors used in experiments in Sections 6.4.1.4 and 6.4.4.

Let ψ denote the local structural extractors for nodes. Given a graph $G_i = (V_i, E_i)$ and ψ, we obtain an $|V_i|$-dimensional node vector $\mathbf{x}_i = \psi(G_i)$. Since any two graphs G_i and G_j are likely to have a different number of nodes and edges, the resulting structural feature matrices $\psi(G_i)$ and $\psi(G_j)$ for these graphs are also likely to be of different sizes as the rows of these matrices correspond to nodes or edges. Thus, in general, these structural feature-based representations of the graphs cannot be used directly to derive similarity between graphs.

Now, to address this issue, we apply the set Σ of *global* statistical meta-graph extractors to every $\psi_i(G)$, $\forall i = 1, \ldots, q$, which summarizes each $\psi_i(G)$ to a vector. Specifically, $\Sigma(\psi_i(G))$ applies each of the statistical functions in Σ (*e.g.*, mean, kurtosis, etc) to the distribution $\psi_i(G)$, which computes a real number that summarizes the given feature distribution $\psi_i(G)$ from different statistical point of view, producing a vector $\Sigma(\psi_i(G)) \in \mathbb{R}^{|\Sigma|}$. Then we obtain the meta-graph feature vector $\mathbf{m}$ of graph G by concatenating the

resulting meta-graph feature vectors:

$$\mathbf{m} = [\Sigma(\psi_1(G)) \ \cdots \ \Sigma(\psi_q(G))] \in \mathbb{R}^d. \tag{6.5}$$

Table 6.8 lists the global statistical functions Σ used in this work to derive meta-graph features. Further, in addition to the node- and edge-level structural features, we also compute global graph statistics (scalars directly derived from the graph, *e.g.*, we use density and degree assortativity coefficient in this work), and append them to $\mathbf{m}$, *i.e.*, the node- or edge-level structural features obtained above.

Most importantly, given any arbitrary graph G', the proposed approach is guaranteed to output a fixed d-dimensional meta-graph feature vector characterizing it. Hence, the structural similarity of any two graphs G and G' can be quantified using a similarity function over $\mathbf{m}$ and $\mathbf{m}'$, respectively.

6.3.4 Embedding Models and Graphs

Given the informative context (*i.e.*, input features) of models and graphs that AUTOGRL learns from model performances $\mathbf{P}$ and meta-graph features $\mathbf{M}$ (Section 6.3.1), how can we use it to effectively learn model and graph embeddings that capture graph-to-model affinity? We note that similar entities can make each other's context more accurate and informative. For instance, in our problem setup, similar models tend to have similar performance distributions over graphs, and likewise similar graphs are likely to exhibit similar affinity to different models. With this consideration, we model the task as a graph representation learning problem, where we construct a graph called *meta-graph* that connects similar models and graphs, and learn the model and graph embeddings over it.

Meta-Graph. We define meta-graph to be a multi-relational graph with two types of nodes (*i.e.*, models and graphs) where edges connect similar model nodes and graph nodes. To measure similarity among graphs and models, we utilize the latent graph and model factors ($\mathbf{U}$ and $\mathbf{V}$, respectively) obtained by factorizing $\mathbf{P}$, as well as the meta-graph features $\mathbf{M}$. More precisely, we use the estimated graph factor $\widehat{\mathbf{U}}$ instead of $\mathbf{U}$ to let the same graph construction process work for new graphs. Note that this gives us two types of features for graph nodes (*i.e.*, $\widehat{\mathbf{U}}$ and $\mathbf{M}$), and one type of features for model nodes (*i.e.*, $\mathbf{V}$). To let different features influence the embedding step differently when needed, we connect graph nodes and model nodes using five type of edges: M-g2g, P-g2g, P-m2m, P-g2m, P-m2g where g and m denote the type of nodes that an edge connects (graph and model, respectively), and M and P denote that the edge is based on meta-graph features and model performance, respectively. For example, M-g2g and P-g2g edges connect two graph nodes that are similar in terms of $\mathbf{M}$ and $\widehat{\mathbf{U}}$, respectively. Then for each edge type, we construct a k-NN graph by connecting nodes to their top-k similar nodes, where node-to-node similarity is defined as the cosine similarity between the corresponding node features. For instance, for P-g2m edge type, graph nodes and model nodes are linked based on the similarity between $\widehat{\mathbf{U}}$ and $\mathbf{V}$.

Learning Over Meta-Graph. Given the meta-graph $\mathcal{G}_{\text{train}}$ that contains meta-train graphs and models, graph neural networks (GNNs) provide an effective framework to embed models and graphs via (weighted) neighborhood aggregation. However, since the graph structure of meta-graph is induced by simple k-NN search, some of the neighbors may not provide the same amount of information as others, or may even provide noisy information. We found it helpful to perform attentive neighborhood aggregation, so more informative neighbors can be given more weights. To this end, we choose to use attentive GNNs designed for multi-relational networks, and specifically use HGT [HDWS20] in experiments. Then the embedding function $f(\cdot)$ in Section 6.3.1 is defined to be $f(\mathbf{h}) = \text{HGT}(\mathbf{h}, \mathcal{G}_{\text{train}})$ during training, which transforms the input node feature $\mathbf{h}$ into an embedding via attentive neighborhood aggregation over $\mathcal{G}_{\text{train}}$.

Inference Over Meta-Graph. For inference at test time, we extend $\mathcal{G}_{\text{train}}$ to be a larger meta-graph $\mathcal{G}_{\text{test}}$ that additionally contains test graph nodes and edges between test graph nodes, and existing graphs and models in $\mathcal{G}_{\text{train}}$. The extension is done in the same way as for the training phase, by finding top-k similar nodes. Then the embedding for the inference can be done by $f(\mathbf{h}) = \text{HGT}(\mathbf{h}, \mathcal{G}_{\text{test}})$.

6.4 Experiments

The experiments are designed to answer the following questions:

- **RQ1 (Model Selection Accuracy):** How accurately do AUTOGRL and baselines select the best model for the new graph?
- **RQ2 (Model Selection Efficiency):** In comparison to the current practice of model selection, how efficient is AUTOGRL?
- **RQ3 (Effects of Meta-Graph Features):** How do different sets of meta graph-features affect the model selection performance?

6.4.1 Experimental Settings

6.4.1.1 Models and Evaluation

Recall that a model in our *automatic graph representation learning* (GRL) problem for downstream applications like link prediction typically includes a combination of two components. The first component learns the graph representation, and the other component leverages the embeddings learned by the first component for the downstream task of interest. In this work, we evaluate our framework for automatically selecting a *link prediction* model for the new graph without performing any evaluations.

Specifically, for the first component of link prediction model, we use 12 popular GRL methods with distinct hyperparameters, and for the second component for link scoring, we use a simple estimator that uses the cosine similarity between the two node embeddings as the link score. This setup results in a model set $\boldsymbol{\mathcal{M}}$ with 423 unique models. The complete list of GRL methods and their hyperparameter settings used to construct the model set is provided in Table 6.5 in Section 6.7.1.

For evaluation, we create a testbed consisting of benchmark graphs, meta-graph features, and a performance matrix $\mathbf{P}$. We construct the performance matrix $\mathbf{P}$ by evaluating each link prediction model in $\mathcal{M}$ on the datasets in the testbed, in terms of the MAP (Mean Average Precision) score. Then we evaluate AUTOGRL and baselines in the testbed via 5-fold cross validation where the benchmark graphs are split into meta-train and meta-test datasets for each fold, and the performance of meta-learners trained over the meta-train data is evaluated using the meta-test dataset.

Since the goal of model selection problem is to accurately predict the best model for the new graph, we evaluate the top-1 prediction performance of meta-learners in terms of AUC (Area Under the ROC Curve), MAP, and NDCG (Normalized Discounted Cumulative Gain). To apply AUC and MAP, we treat the task as a binary classification problem, where only the top-1 model (*i.e.*, the model with the highest performance for the given graph) is labeled as 1, while all others are labeled as 0. For NDCG, we report NDCG@1, which concerns only the top-1 predicted model's performance. All metrics range from 0 to 1, with larger values indicating higher accuracy.

6.4.1.2 Testbed Setup.

To evaluate meta-learners' performance in different usage scenarios, we construct two testbeds.

- ***Search-within-a-model* testbed** evaluates how well meta-learners perform in finding the best hyperparameter configuration (HC) of a specific method. Thus performance matrix $\mathbf{P}$ contains only the performances obtained with different HCs of a single method.
- ***Search-across-all-models* testbed** evaluates how accurately meta-learners select the best model from the heterogeneous model set composed by pairing each model with its distinct HCs. This is the most general and challenging setup, where performance matrix $\mathbf{P}$ consists of all models' performance on all graphs.

Both testbeds use a graph corpus consisting of 301 graphs from 21 domains, which have fundamentally different structural characteristics. Table 6.6 shows the distribution of graphs per data domain.

6.4.1.3 Baselines

Being the first work for automatic model selection in graph learning, we do not have immediate baselines for comparison. Therefore, we adapt some existing approaches for our problem setting, and also devise baselines based on simple ideas frequently used in practice. Baselines can be organized into three categories.

(a) ***No model selection*** employs the same popular model for link prediction.

- **node2vec** [GL16] is a popular graph representation learning method.
- **GCN** [KW17] denotes graph convolutional network.

(b) ***Simple meta-learners*** select a model that performs generally well, either globally or locally.

- **Global Best (GB)-Avg** selects the model with the largest mean performance across all meta-train graphs.
- **Global Best (GB)-Rank** selects the model that was the best performing most frequently for all meta-train graphs.
- **ISAC** [KMST10] first clusters meta-train datasets using meta-graph features, and at test time, finds the cluster closest to the test graph, and selects the model with the largest average performance over all graphs in that cluster.
- **ARGOSMART (AS)** [NMJ13] finds the meta-train graph closest to the test graph (*i.e.*, 1NN) in terms of meta-graph feature similarity, and selects the model with the best result on the 1NN graph.

GB-Avg and GB-Rank do not use meta-features for model selection.

(c) ***Optimization-based meta-learners*** learn to estimate the model performance by modeling the relation between meta-graph features and model performances.

- **Supervised Surrogates (SS)** [XHS+12] learns a surrogate model (a regressor) that maps meta-graph features to model performances.
- **ALORS** [MS17] factorizes the performance matrix into latent factors on graphs and models, and estimates the performance to be the dot product between the two factors, where a non-linear regressor maps meta-graph features into the latent graph factors.
- **NCF** [HLZ+17] improves upon ALORS by replacing dot product with a more general neural architecture that estimates performance by combining the linearity of MF and non-linearity of DNNs.

In addition, we also include **Random Selection (RS)** as a baseline to see how these methods compare to randomly scoring models. Note that except the simplest meta-learner GB, all of the above baseline meta-learners rely on the proposed meta-graph features to be able to estimate model performances on an unseen test graph.

6.4.1.4 Meta-Graph Features

In experiments, we used density of $\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$, and also the degree assortativity coefficient r. Also, we derived the distribution of degrees, k-core numbers, PageRank scores, wedges, and triangles. For each of these structural property distributions (degree, k-core numbers, and so on), we apply the set Σ of statistical functions (Table 6.8) over it to obtain a vector representation for the per-node/edge structural feature/distribution. Finally, we concatenate all of the meta-graph features together to obtain the final meta-graph feature vector $\mathbf{m}$ for the graph.

6.4.2 Model Selection Accuracy (RQ1)

We evaluate model selection accuracy using both testbed setups.

Table 6.2: AUTOGRL achieves higher model selection accuracy than several baselines in most cases on the *search-within-a-model* testbed. Results are obtained via 5-fold cross validation. The best results are in bold, and the second best results are underlined.

Search-within-a-model testbed	DeepGL [RZA20]			node2vec [GL16]			HONE [RAK18]			GraphSage [HYL17]			role2vec [ARL+18]		
	AUC	**MAP**	**NDCG@1**	**AUC**	**MAP**	**NDCG@1**	**AUC**	**MAP**	**NDCG@1**	**AUC**	**MAP**	**NDCG@1**	**AUC**	**MAP**	**NDCG@1**
Random Selection	0.495	0.036	0.867	0.485	0.303	0.972	0.503	0.127	0.826	0.516	0.195	0.822	0.514	0.030	0.792
Global Best-Avg.	0.691	0.164	0.938	0.502	0.331	0.974	0.779	0.448	0.931	0.783	0.433	0.947	0.846	**0.146**	0.955
Global Best-Rank	0.698	0.192	0.938	0.581	0.377	0.975	0.778	0.440	0.931	0.784	0.407	0.943	0.808	0.119	0.954
ALGOSMART	0.770	0.210	0.935	0.615	0.400	**0.982**	0.786	0.422	**0.939**	0.777	0.409	0.947	0.812	0.124	0.951
ISAC	0.713	0.170	0.940	0.611	0.413	**0.982**	0.777	0.424	0.930	0.785	0.430	0.945	**0.852**	0.144	0.954
ALORS	0.670	0.079	0.925	0.566	0.382	0.981	0.746	0.300	0.924	0.797	0.426	0.945	0.818	0.113	0.950
NCF	0.695	0.144	0.934	0.545	0.358	0.978	0.767	0.392	0.929	0.785	0.420	0.945	0.844	0.126	0.951
Supervised Surrogate	0.610	0.051	0.918	0.586	0.390	0.979	0.752	0.324	0.927	0.766	0.367	0.935	0.809	0.112	0.951
AUTOGRL (Ours)	**0.791**	**0.237**	**0.947**	**0.632**	**0.427**	**0.982**	**0.801**	**0.456**	**0.939**	**0.814**	**0.440**	**0.948**	**0.852**	0.139	**0.961**

Table 6.3: AUTOGRL consistently outperforms existing model selection methods on the *search-across-all-models* testbed that involves 423 models and 301 graphs. Results are obtained via 5-fold cross validation. The best results are in bold, and the second best results are underlined.

	Search-across-all-models testbed		
	AUC	**MAP**	**NDCG@1**
Random Selection	0.490	0.011	0.745
GCN	0.499	0.002	0.755
node2vec	0.505	0.016	0.931
Global Best-Avg.	0.877	0.163	0.932
Global Best-Rank	0.834	0.205	0.933
ALGOSMART	0.905	0.222	0.947
ISAC	0.891	0.215	0.941
ALORS	0.868	0.120	0.921
NCF	0.875	0.132	0.931
Supervised Surrogate	0.861	0.128	0.933
AUTOGRL (Ours)	**0.936**	**0.243**	**0.962**

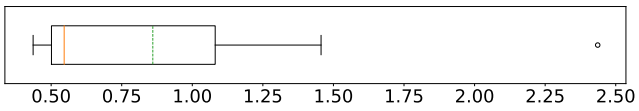
Search-within-a-model testbed. In this setup, the goal is to select the best model from among the HCs of a single method. In Table 6.2, we show how effective AUTOGRL and baselines are in finding the best HC of five chosen methods, averaged over 301 graphs. Results show that AUTOGRL nearly consistently achieves the highest model selection accuracy in terms of three evaluation metrics. Among baselines, ALGOSMART achieves better performance than other baselines. However, there is no consistent winner among baselines. Notably, optimization-based meta-learners, such as ALORS and SS, are mostly outperformed by simpler meta-learners like ISAC and ALGOSMART, which first find similar meta-train graphs and use their observed performance directly for model selection. On the other hand, optimization-based meta-learners try to reconstruct the performance matrix via matrix factorization or regression, which is a much harder task. As an optimization-based technique, AUTOGRL outperforms these baselines by more effectively capturing graph-to-model affinity via learning over meta-graph.

Search-across-all-models testbed This setup aims to identify the best model among all existing methods and their HCs. Thus, most methods that focus on finding an optimal HC for a single method can not be used. Table 6.3 shows the results. Always using popular methods, such as GCN and node2vec, does not perform well. Although these methods perform reasonably well in comparison to other methods in the testbed, they are not often the best model. Note that these models are associated with a specific hyperparameter setting (HS), and thus are treated differently from the same method with a different HS. Again, simple meta-learners, especially ALGOSMART, often outperform optimization-based meta-learners. Our previous discussion on this observation equally applies here. At the same time, this result shows the effectiveness of our meta-graph features, since the performance of ALGOSMART and ISAC heavily relies on the quality of meat-graph features for effective model selection. In this challenging setup, AUTOGRL consistently outperforms baselines in all three categories.

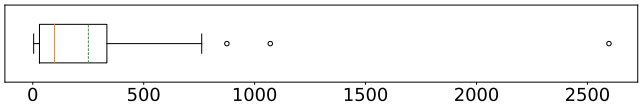
6.4.3 Model Selection Efficiency (RQ2)

To evaluate how efficient AUTOGRL's model selection is, we measure AUTOGRL's runtime (*i.e.*, time to create meta-graph features for new graph at test time, plus the time to make a prediction), and compare it with the time taken for systematic search corresponding to the two testbed settings. Figure 6.4 shows the results in box plots, where orange and dotted green lines denote the median and mean.

First, Figure 6.4a shows that AUTOGRL runs fast. On average over all graphs in the testbed, it takes less than 1 second to run. Next, Figure 6.4b shows the ratio of time taken for searching within a model (*i.e.*, evaluating different hyperparameter configurations (HC) of a single method) to AUTOGRL's runtime. Since it takes too much time to try out all HCs for all GRL methods in our testbed, we selected eight methods and graphs (Table 6.7), and used them as representative cases. On average, search-within-a-model type of evaluation takes 250$\times$ longer time than AUTOGRL. Figure 6.4c shows the ratio of time taken for searching across all models to AUTOGRL's runtime. On average, search-across-all-models evaluation takes 2006$\times$ longer than AUTOGRL. As an aside, these

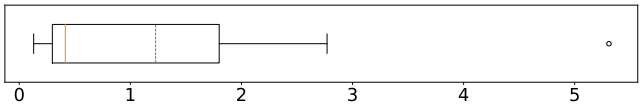


(a) AUTOGRL's runtime (seconds) at test time.



(b) Ratio of time to search within a model to AUTOGRL's runtime.

(c) Ratio of time to search across all models to AUTOGRL's runtime.



(d) Ratio of time to create meta-graph features to time to make a prediction (both by AUTOGRL).

Figure 6.4: AUTOGRL's runtime (*i.e.*, time to generate meta-graph features and make a prediction) at test time and comparison to time taken for systematic search. Orange and dotted green lines denote the median and mean, respectively. AUTOGRL is fast (<1 second), and incurs negligible overhead.

results are on small graphs, and hence, the runtime of naive search would become even larger as graph size gets larger, and also as the testbed gets bigger. Finally, Figure 6.4d shows the ratio of time taken for AUTOGRL to create meta-graph features to the inference time. On average, meta graph-feature generation takes $\sim$20% longer time than inference, while both of them in aggregate take <1 second. Results show that AUTOGRL is fast, and incurs negligible overhead.

6.4.4 Effects of Meta-Graph Features (RQ3)

In this section, we investigate using a different set of meta-graph features. We use $\mathcal{M}$ to denote the default set with $\sim$300 meta-graph features described in Section 6.4.1.4. Then we construct another set $\mathcal{M}_{\text{small}}$ ($\sim$60 features), which is smaller than $\mathcal{M}$, and consists of simple structural features such as degree and k-core. We create $\mathcal{M}_{\text{extended}}$ ($\sim$1000 features) by extending $\mathcal{M}$ with features related to edge-centric graphlet frequency. In particular, we counted all 3 and 4 node graphlets (network motifs) for every edge in the graph. For each of the graphlet frequency distributions (3-stars, 4-cycles, 4-cliques, etc), we apply the functions in Table 6.8 over it, and obtain a vector representation for the edge graphlet frequency distribution. Then we obtain $\mathcal{M}_{\text{extended}}$ by concatenating these meta-graph features with $\mathcal{M}$.

Table 6.4 shows the performance achieved by different methods in search-across-all-

Table 6.4: By effectively capturing the structural characteristics of graphs, our proposed meta-graph features enable effective automatic model selection by AUTOGRL as well as other methods that rely on meta-features. Using different sets of meta-graph features ($\mathcal{M}_{\text{small}}$, $\mathcal{M}$, and $\mathcal{M}_{\text{extended}}$), AUTOGRL consistently outperforms all baselines.

Search-across-all-models testbed	$\mathcal{M}_{\text{small}}$			$\mathcal{M}$			$\mathcal{M}_{\text{extended}}$		
	AUC	**MAP**	**NDCG@1**	**AUC**	**MAP**	**NDCG@1**	**AUC**	**MAP**	**NDCG@1**
Random Selection	0.513	0.022	0.742	0.490	0.011	0.745	0.490	0.011	0.745
GCN	0.499	0.002	0.755	0.499	0.002	0.755	0.499	0.002	0.755
node2vec	0.505	0.016	0.931	0.505	0.016	0.931	0.505	0.016	0.931
Global Best-Avg.	0.877	0.163	0.932	0.877	0.163	0.932	0.877	0.163	0.932
Global Best-Rank	0.834	0.205	0.933	0.834	0.205	0.933	0.834	0.205	0.933
ALGOSMART	0.889	0.206	0.946	0.905	0.222	0.947	0.911	0.224	0.952
ISAC	0.886	0.220	0.944	0.891	0.215	0.941	0.881	0.196	0.942
ALORS	0.869	0.159	0.935	0.868	0.120	0.921	0.851	0.096	0.918
NCF	0.873	0.176	0.932	0.875	0.132	0.931	0.877	0.148	0.935
Supervised Surrogate	0.871	0.132	0.928	0.861	0.128	0.933	0.854	0.087	0.920
AUTOGRL (Ours)	**0.928**	**0.237**	**0.955**	**0.936**	**0.243**	**0.962**	**0.936**	**0.246**	**0.958**

models testbed using those three sets of meta-graph features. Results show that AUTOGRL consistently achieves the best performance as different meta-graph features are used. Even a relatively small set of meta-features $\mathcal{M}_{\text{small}}$ can be effectively used by different methods for model selection. Using a larger set $\boldsymbol{\mathcal{M}}$ of meta-graph features improves the performance of AUTOGRL and a few baselines. On the other hand, a much larger feature set $\mathcal{M}_{\text{extended}}$ does not lead to much improvement for most methods, except for ALGOSMART. However, we hypothesize that additional information that edge graphlets frequency in $\mathcal{M}_{\text{extended}}$ brings may become useful as the testbed gets expanded with more diverse graphs, and also applied to other tasks than link prediction. We release all three sets of meta-graph features to the community.

6.5 Related Work

In this section, we review previous works on model selection (*i.e.*, selecting an algorithm and its hyperparameter settings) in the areas of machine learning and graph learning.

6.5.1 Model Selection in Machine Learning

As increasingly complex machine learning models (*e.g.*, deep neural networks) are widely used, manual model selection, which highly relies on human expertise, is becoming too expensive or even impractical [YZ20]. Thus, a lot of research has focused on automating model selection in ML [HZC21], which can be organized into two categories.

6.5.1.1 Evaluation-Based Model Selection

A majority of model selection methods belong to this category. Representative techniques used by these methods include grid search [LL19], random search [BB12], early stopping-based [GSM+17] and bandit-based [LJD+17] approaches, and Bayesian optimization (BO) [SLA12, WCZ+19, FKH18]. Among them, BO methods are more efficient than grid or random search, requiring fewer evaluations of hyperparameter configurations (HCs), as they determine which HC to try next in a guided manner using prior experience from previous trials. Note that these methods perform model training and/or evaluation multiple times using different HCs, and have much limited efficiency compared to the next group of methods.

6.5.1.2 Evaluation-Free Model Selection

Methods in this category require no model evaluation for model selection at inference time. A simple approach [ABvRV18] identifies the best model by considering the algorithm's rankings observed on prior datasets. Instead of finding the globally best model, ISAC [KMST10] and ARGOSMART [NMJ13] select a model that performed well on similar datasets, where the dataset similarity is taken into account in the meta-feature space by using clustering [KMST10] or k-nearest neighbor search [NMJ13]. A different group of methods perform optimization-based model selection, where the model performance is estimated by modeling the relation between meta-features and model performances. For instance, Supervised Surrogates [XHS+12] learns a (non-linear) surrogate model that maps meta-features to model performance, and ALORS [MS17] models the performance as a dot product between latent factors of models and datasets obtained via matrix factorization. Notably, all of these methods, except the first one, rely on meta-features, while focusing on non-graph datasets. By applying our proposed meta-graph features, we make them applicable to the graph model selection task.

6.5.2 Model Selection in Graph Learning

A majority of graph learning works focus on developing new, effective graph learning algorithms for certain graph learning tasks and applications [XSY+21, ZCZ22]. In comparison, there exist relatively few recent works [TMC+19, GYZB21, YWCP21, BLL21, ZTLL21], which aim to address the graph learning model selection problem. These studies mainly focus on neural architecture search and hyperparameter optimization (HPO) for graph learning models, especially for graph neural networks (GNNs). To achieve more efficient model selection than the naive exhaustive approach (Figure 6.1b), these studies investigated several techniques for efficient HPO, including subgraph sampling [TMC+19], graph coarsening [GYZB21], hierarchical evaluation [YWCP21], hypernets [ZTLL21], and evolutionary algorithms [BLL21]. However, in order to select the best model for a new graph, they still need to perform model training and/or evaluations on the new graph, which is significantly more costly than evaluation-free model selection. Also, these works are mostly limited to finding the best HC of a specific graph learning model, *e.g.*, GCN [KW17], and cannot be used to select a model from a heterogeneous model set $\mathcal{M}$ consisting of various graph learning models (which are typically a com-

bination of graph representation learning methods, downstream task-specific methods such as link predictor, and their hyperparameters). Previous work on network similarity [BKEF12] is somewhat relevant to graph model selection. Yet, it solely focuses on extracting seven node-level features, and does not concern model selection. In comparison, AUTOGRL provides a much richer set of 300+ meta-graph features on node, edge, and graph level. To the best of our knowledge, there exists *no prior work* that can infer the best graph learning model without incurring model training or evaluations. Our proposed AUTOGRL is the first approach for selecting the best model in an evaluation-free manner, from among any heterogeneous set of graph learning models.

6.6 Conclusion

In this chapter, we tackle fast automatic model selection problem for graph representation learning by making the following contributions.

- **Problem Formulation.** We formulate the problem of training and evaluation-free model selection for graph learning,
- **Framework for Automatic Graph Learning.** We propose AUTOGRL, the first approach to automatic graph learning that infers the best model for a new unseen graph in *near real-time*, while *requiring supervision for model training or evaluation*.
- **Specialized Meta-Graph Features.** We design specialized meta-graph features that capture the structural characteristics of graphs.
- **Effectiveness and Efficiency.** Comprehensive experiments show that model selecting by AUTOGRL significantly outperforms always using popular models, as well as other meta-learners adapted for our problem, while incurring negligible overhead.

6.7 Appendix

6.7.1 Model Set

A model in the model set $\mathcal{M}$ refers to a graph representation learning (GRL) method along with its hyperparameters settings, and a link predictor that scores a given link based on the two nodes' embeddings. Table 6.5 shows the complete list of 12 popular GRL methods and their specific hyperparameter settings, which compose 412 unique models in the model set $\mathcal{M}$. Note that the link predictor is omitted from Table 6.5 since we employ the same link predictor based on cosine similarity scoring to all GRL methods.

6.7.2 Graph Domains

The testbed used in this work contains 301 graphs taken from 21 domains, which have widely different structural properties. Table 6.6 show the distribution of testbed graphs across graph domains.

6.7.3 Runtime

Table 6.7 shows results comparing the runtime (in seconds) for naive model selection to the runtime of AUTOGRL. Results show that AUTOGRL is fast, and incurs negligible

Table 6.5: Graph representation learning (GRL) models and their hyperparameter settings, which collectively comprise the model set $\mathcal{M}$ with 423 unique GRL models. For more details of the hyperparameters, please refer to the cited paper.

Methods	Hyperparameter Settings	Count
SGC [WJZ+19]	# (number of) hops $k \in \{1, 2, 3\}$	3
GCN [KW17]	# layers $L \in \{1, 2, 3\}$, # epochs $N \in \{1, 10\}$	6
GraphSAGE [HYL17]	# layers $L \in \{1, 2, 3\}$, # epochs $N \in \{1, 10\}$, aggregation functions $f \in \{\text{mean}, \text{gcn}, \text{lstm}\}$	18
node2vec [GL16]	$p, q \in \{1, 2, 4\}$	9
role2vec [ARL+18]	$\alpha \in \{0.01, 0.1, 0.5, 0.9, 0.99\}$, motif combinations $\mathcal{H} \in \{\{H_1\}, \{H_2, H_3\}, \{H_2, H_3, H_4, H_6, H_8\}, \{H_1, H_2, \ldots, H_8\}\}$, $p, q \in \{0.25, 1, 4\}$	180
GraRep [CLX15]	$k \in \{1, 2\}$	2
DeepWalk [PAS14]	$p = 1, q = 1$	1
HONE [RAK18]	$k \in \{1, 2\}$, $D_{\text{local}} \in \{4, 8, 16\}$, variant $v = \{1, 2, 3, 4, 5\}$	30
node2bits [JHRK19]	walk num $w_n \in \{5, 10, 20\}$, walk len $w_l \in \{5, 10, 20\}$, log base $b \in \{2, 4, 8, 10\}$, feats $f \in \{16\}$	36
DeepGL [RZA20]	$\alpha \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, motif size $\in \{4\}$, eps tolerance $t \in \{0.01, 0.05, 0.1\}$, relational aggr. $\in \{\{m\}, \{p\}, \{s\}, \{v\}, \{m, p\}, \{m, v\}, \{s, m\}, \{s, p\}, \{s, v\}\}$ where m, p, s, v denote mean, product, sum, var	135
LINE [TQW+15]	# hops/order $k \in \{1, 2\}$	2
Spectral Emb. [LWH03]	tolerance $t \in \{0.001\}$	1
Total Count		**423**

Table 6.6: Domains of the graph datasets in the testbed, and the number of graphs for each domain. In sum, our testbed contains 301 graphs drawn from 21 domains, which have fundamentally different structural characteristics.

Graph Data Domain	Number of Graphs
Protein Networks	50
Cheminformatics Network	50
Retweet Network	27
Synthetic-KPGM	25
Biological Network	23
Synthetic-BA	18
Facebook Network	17
Synthetic-CL	15
Web Graph	10
Collaboration Network	10
Social Network	10
Brain Network	9
Synthetic-ER	6
Ecology Network	6
Road Network	6
Email Network	6
Power Networks	6
Recommendation Network	2
Technological Network	2
Infrastructure Network	2
Scientific Computing Network	1
Total Count	**301**

Table 6.7: Results comparing the runtime (in seconds) for naive model selection (*i.e.*, training and evaluating each method using every hyperparameter configuration in the model set $\mathcal{M}$) to the runtime of AUTOGRL (the penultimate row refers to the time to generate meta-graph features, and the last row is the average time taken for model prediction).

	biogrid-plant	**web-pol blogs**	**soc-wiki-Vote**	**eco-mang wet**	**ia-reality**	**tech-routers-rf**	**web-EPA**	**socfb-Caltech**
line	6.22	6.40	5.45	6.37	5.85	5.40	7.28	8.19
node2vec	36.48	52.34	65.28	18.40	504.38	154.54	317.55	184.09
deepwalk	3.84	5.44	7.03	1.84	55.01	16.89	33.09	18.24
HONE	19.24	169.22	203.71	11.20	53.15	552.31	882.35	737.49
node2bits	44.33	55.12	64.85	43.35	113.06	92.22	117.55	106.66
deepGL	72.81	106.95	145.93	71.08	633.44	331.87	880.03	445.89
GraphSage	108.20	301.22	272.97	339.73	1451.87	513.18	1020.30	2586.93
GCN	15.04	22.00	26.30	17.12	57.10	45.64	66.12	94.65
AUTOGRL	0.10	0.14	0.16	0.06 0.39	0.97	0.36	0.78	0.61

computational overhead.

6.7.4 AUTOGRL Algorithm

Algorithm 6.1 provides detailed steps of AUTOGRL, for both offline meta-training (top) and online model selection (bottom).

6.7.5 Experimental settings

We set the embedding size k to 32 for AUTOGRL and other meta-learners that learn embeddings of models and graphs. For AUTOGRL, we created the meta-graph by connecting nodes to their top-10 similar nodes. As an the embedding function $f(\cdot)$ in AUTOGRL, we used HGT [HDWS20] with 2 layers and 4 heads per layer. We used Adam optimizer for training.

6.7.6 Meta-Graph Features

Table 6.8 provides a summary of the global statistical functions Σ to derive a set of meta-graph features from a vector of graph structural features (*e.g.*, node degrees, k-core numbers, etc).

Table 6.8: Summary of the global statistical functions Σ for deriving a set of meta-graph features from a graph invariant (*e.g.*, k-core numbers, node degrees, and so on). Let $\mathbf{x}$ denote an arbitrary graph invariant vector for some graph $G_i = (V_i, E_i)$ and $\pi(\mathbf{x})$ is the sorted vector of $\mathbf{x}$. Note $\mathbf{x}$ can be any representation, *e.g.*, node degree vector (value for each node in G_i) or a degree distribution vector.

Name	**Equation**
Num. unique values	$\mathsf{card}(\mathbf{x})$
Density	$\mathsf{nnz}(\mathbf{x})/\lvert\mathbf{x}\rvert$
Q_1, Q_3	median of the $\lvert\mathbf{x}\rvert/2$ smallest (largest) values
IQR	$Q_3 - Q_1$
Outlier LB $\alpha \in \{1.5, 3\}$	$\sum_i \mathbb{I}(x_i < Q_1 - \alpha IQR)$
Outlier UB $\alpha \in \{1.5, 3\}$	$\sum_i \mathbb{I}(x_i > Q_3 + \alpha IQR)$
Total outliers $\alpha \in \{1.5, 3\}$	$\sum_i \mathbb{I}(x_i < Q_1 - \alpha IQR) + \sum_i \mathbb{I}(x_i > Q_3 + \alpha IQR)$
(α-std) outliers $\alpha \in \{2, 3\}$	$\mu_\mathbf{x} \pm \alpha\sigma_\mathbf{x}$
Spearman (ρ, p-val)	$\mathsf{spearman}(\mathbf{x}, \pi(\mathbf{x}))$
Kendall (τ, p-val)	$\mathsf{kendall}(\mathbf{x}, \pi(\mathbf{x}))$
Pearson (r, p-val)	$\mathsf{pearson}(\mathbf{x}, \pi(\mathbf{x}))$
Min, max	$\min(\mathbf{x})$, $\max(\mathbf{x})$
Range	$\max(\mathbf{x}) - \min(\mathbf{x})$
Median	$\mathsf{med}(\mathbf{x})$
Geometric Mean	$\lvert\mathbf{x}\rvert^{-1} \prod_i x_i$
Harmonic Mean	$\lvert\mathbf{x}\rvert \,/ \sum_i \frac{1}{x_i}$
Mean, Stdev, Variance	$\mu_\mathbf{x}$, $\sigma_\mathbf{x}$, $\sigma_\mathbf{x}^2$
Skewness	$\mathbb{E}(\mathbf{x}-\mu_\mathbf{x})^3/\sigma_\mathbf{x}^3$
Kurtosis	$\mathbb{E}(\mathbf{x}-\mu_\mathbf{x})^4/\sigma_\mathbf{x}^4$
Quartile Dispersion Coeff.	$\frac{Q_3-Q_1}{Q_3+Q_1}$
Median Absolute Deviation	$\mathsf{med}(\lvert\mathbf{x} - \mathsf{med}(\mathbf{x})\rvert)$
Avg. Absolute Deviation	$\frac{1}{\lvert\mathbf{x}\rvert}\mathbf{e}^T\lvert\mathbf{x} - \mu_\mathbf{x}\rvert$
Coeff. of Variation	$\sigma_\mathbf{x}/\mu_\mathbf{x}$
Efficiency ratio	$\sigma_\mathbf{x}^2/\mu_\mathbf{x}^2$
Variance-to-mean ratio	$\sigma_\mathbf{x}^2/\mu_\mathbf{x}$
Signal-to-noise ratio (SNR)	$\mu_\mathbf{x}^2/\sigma_\mathbf{x}^2$
Entropy	$H(\mathbf{x}) = -\sum_i \; x_i \log x_i$
Norm. entropy	$H(\mathbf{x})/\log_2\lvert\mathbf{x}\rvert$
Gini coefficient	$-$
Quartile max gap	$\max(Q_{i+1} - Q_i)$
Centroid max gap	$\max_{ij} \lvert c_i - c_j\rvert$
Histogram prob. dist.	$\mathbf{p}_h = \frac{\mathbf{h}}{\mathbf{h}^T\mathbf{e}}$ (with fixed # of bins)

Algorithm 6.1: AUTOGRL: Offline Meta-Training (Top) and Online Model Selection (Bottom)

Input: Meta-train graph database $\boldsymbol{\mathcal{G}}$, model set $\boldsymbol{\mathcal{M}}$, embedding dimension k
Output: Meta-learner for model selection
/* (Offline) Meta-Learner Training */
Train & evaluate models in $\boldsymbol{\mathcal{M}}$ on graphs in $\boldsymbol{\mathcal{G}}$ to get performance matrix $\mathbf{P}$
Extract meta-graph features $\mathbf{M}$ for each graph G_i in $\boldsymbol{\mathcal{G}}$ (Sec. 6.3.3)
Factorize $\mathbf{P}$ to obtain latent graph factors $\mathbf{U}$ and model factors $\mathbf{V}$, *i.e.*, $\mathbf{P} \approx \mathbf{U}\mathbf{V}^\intercal$
Learn an estimator $\phi(\cdot)$ such that $\phi(\mathbf{m}) = \widehat{\mathbf{U}}_i \approx \mathbf{U}_i$
Create meta-train graph $\mathcal{G}_{\text{train}}$ (Sec. 6.3.4)
while *not converged*
    for $i = 1, \ldots, n$ **do**
        Get embeddings $f(\mathbf{W}[\mathbf{m}; \phi(\mathbf{m})])$ of train graph G_i on $\mathcal{G}_{\text{train}}$
        for $j = 1, \ldots, m$ **do**
            Get embeddings $f(\mathbf{V}_j)$ of each model M_j on $\mathcal{G}_{\text{train}}$
            Estimate $\widehat{p}_{ij} = \langle f(\mathbf{W}[\mathbf{m}; \phi(\mathbf{m})]), f(\mathbf{V}_j) \rangle$ (Eqn. 6.1)
        end
    end
    Compute meta-training loss $L(\mathbf{P}, \widehat{\mathbf{P}})$ (Eqn. 6.3) and optimize parameters
end

Input: new graph G_{test}
Output: selected model M^* for G_{test}
/* (Online) Model Selection (Section 6.3.2) */
Extract graph meta-features $\mathbf{m}_{\text{test}} = \psi(G_{\text{test}})$
Estimate latent factor $\widehat{\mathbf{U}}_{\text{test}} = \phi(\mathbf{m}_{\text{test}})$ for test graph G_{test}
Create meta-test graph $\mathcal{G}_{\text{test}}$ by extending $\mathcal{G}_{\text{train}}$ with new edges between test graph node and existing nodes in $\mathcal{G}_{\text{train}}$ (Sec. 6.3.4)
Get embeddings $f(\mathbf{W}[\mathbf{m}_{\text{test}}; \widehat{\mathbf{U}}_{\text{test}}])$ of test graph on $\mathcal{G}_{\text{test}}$
Get embeddings $f(\mathbf{V}_j)$ of each model M_j on $\mathcal{G}_{\text{test}}$
Return the best model $M^* \leftarrow \arg\max_{M_j \in \boldsymbol{\mathcal{M}}} \left\langle f(\mathbf{W}[\mathbf{m}_{\text{test}}; \widehat{\mathbf{U}}_{\text{test}}]), f(\mathbf{V}_j) \right\rangle$

Part II

Dynamic Graphs and Tensors

Chapter based on work published in ICDE 2021 [PKH+21].

Chapter 7

Knowledge-Guided Dynamic Systems Modeling

Modeling real-world phenomena is a focus of many science and engineering efforts, such as ecological modeling and financial forecasting, to name a few. Building an accurate model for complex and dynamic systems improves understanding of underlying processes and leads to resource efficiency. Towards this goal, knowledge-driven modeling builds a model based on human expertise, yet is often suboptimal. At the opposite extreme, data-driven modeling learns a model directly from data, requiring extensive data and potentially generating overfitting. We focus on an intermediate approach, model revision, in which prior knowledge and data are combined to achieve the best of both worlds. In this chapter, we propose a genetic model revision framework based on tree-adjoining grammar (TAG) guided genetic programming (GP), using the TAG formalism and GP operators in an effective mechanism to incorporate prior knowledge and make data-driven revisions in a way that complies with prior knowledge. Our framework is designed to address the high computational cost of evolutionary modeling of complex systems. Via a case study on the challenging problem of river water quality modeling, we show that the framework efficiently learns an interpretable model, with higher modeling accuracy than existing methods.

7.1 Introduction

Modeling real-world phenomena is the goal of numerous science and engineering endeavors, such as ecological modeling [KMS+10], financial forecasting [LLC09], user modeling [WPB01], disease prediction [PKP+18], popularity estimation [PKD+19, PKD+20], student dropout prediction [JPB20], and drug discovery [BSMN03]. An accurate model of these systems can enable better understanding of underlying mechanisms and more effective use of resources. Real-world systems are typically dynamic and complex, with

multiple observed and latent variables that change over time, and affect each other in complex and often nonlinear ways. As an example, consider the task of forecasting river water quality. Addressing this problem requires an understanding of processes such as plankton dynamics and hydrological mechanisms, and modeling how they influence the system dynamics as a whole.

Existing approaches for modeling dynamical systems can be grouped into three classes. The first is *knowledge-driven modeling*. The structure of knowledge-driven models and their parameters are determined by domain experts, based on their prior knowledge and using observational data to calibrate the model parameters. In knowledge-driven modeling, the state of dynamic systems can be modeled by differential equations. While knowledge-driven models perform reasonably well when the modelled system is simple, they take time to construct, and generally perform less well with increasing system complexity.

The second is *data-driven modeling*: learning a model purely from data, with no need for prior knowledge. Highly accurate models can often be obtained by these methods. Modeling complex systems requires plentiful data, but the high cost of measurement [KAF+17] means this is often unavailable. Sadly, learning a model from limited data often leads to overfitting. Importantly, data-driven modeling might learn models that are not consistent with prior knowledge. Also, some of the popular methods in this class (e.g., neural networks) generate black box models, lacking explanatory power.

The third class combines knowledge- and data-driven modeling to gain the best of both worlds. *Model calibration* is one widely used approach: the initial model structure is specified by domain knowledge, and then model parameters are optimized using data. However, model calibration updates only the model parameters, not the model structure. If this is oversimplified, the accuracy of the optimized model will be compromised, and the calibrated parameter values will be unrealistic. *Model revision* is a more interesting and effective approach: prior knowledge specifies the initial model structure and parameter values, but both are updated iteratively to obtain a better fit to the data. This approach of revising and improving existing models closely resembles traditional scientific discovery process [DLT07]. *Knowledge-guided model revision* further improves plain model revision by letting model revision be guided by prior knowledge and producing a revised model consistent with domain knowledge. Table 7.1 summarizes how different approaches satisfy desirable properties for knowledge-guided modeling of complex dynamic systems.

As a powerful technique for evolving programs, genetic programming (GP) [Koz93] provides an effective framework for model revision. GP has been successfully applied to real-world problems in various fields [VGT15, AWWB09, DNM+12], and has the theoretical advantage that the output is interpretable, unlike blackbox models. Among GP's methods, symbolic regression (SR), which aims to discover a function that fits the training data, is the most relevant to process modeling. Standard SR is a form of data-driven modeling, as it sets no restrictions on the model structure. It thus suffers from a lack of guidance in the optimization process, and may produce models that violate

Table 7.1: Model revision satisfies all properties for interpretable knowledge-guided modeling of complex dynamic systems. Other approaches miss one or more of the properties. "?" means that it depends on the specific method used.

Property \ Approach	Knowledge-Driven Modeling	Data-Driven Modeling	Model Calibration	Model Revision	Knowledge-Guided Model Revision
Learning models consistent with prior knowledge	✓		?		✓
Knowledge-based model specification	✓		✓	✓	✓
Structural model update		?		✓	✓
Automatic tuning of model parameters		✓	✓	✓	✓
Capacity to model complex systems		✓		✓	✓
Interpretable	✓	?	✓	✓	✓

domain knowledge.

A number of newer GP methodologies, such as grammar guided GP (GGGP) [Whi96] and tree-adjoining grammar (TAG) guided GP (TAG3P) [Ngu04], support constraining or biasing the structure of learnt models [Mon95, OR01]. We base our framework on TAG3P, which is a powerful tool for incorporating domain knowledge while exploring the complex search spaces required for modeling real-world processes.

In this chapter, we propose TAG3P-based genetic model revision (GMR), in which the TAG formalism and GP operators provide an effective mechanism to perform data-driven model revisions based on prior knowledge. We show how to represent dynamic processes in TAG, and how to extend the TAG3P framework to incorporate different types of prior knowledge into the optimization process. An important challenge in applying GP to complex systems is the high computational cost of the search and fitness evaluation in GP systems. Our framework achieves efficient and effective optimization by reducing redundancy and enabling evaluation short-circuiting. In our case study, GMR allows us to accurately model water quality in a river ecosystem, a complex dynamic system with extensive geographic coverage, which has previously been much less studied than relatively simple lake ecosystems due to its far higher complexity.

In summary, our contributions are as follows:

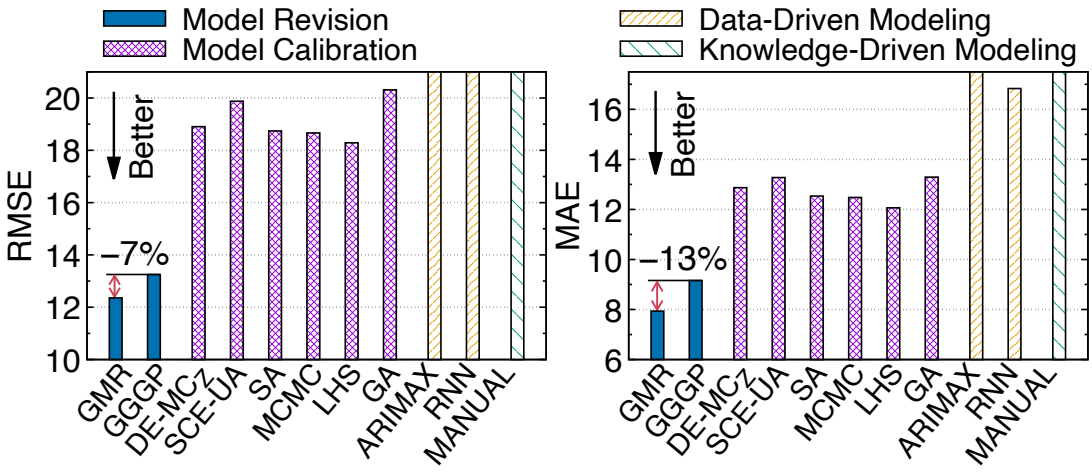


Figure 7.1: GMR achieves the best forecasting accuracy in the river modeling task, obtaining 7% and 13% lower RMSE (left) and MAE (right), respectively, than the second best method, while producing revised models guided by domain knowledge.

- *Framework.* We present a GMR framework for dynamic systems modeling, which improves a knowledge-based model in a data-driven manner, guided by prior knowledge.
- *Knowledge Incorporation.* We design novel mechanisms to represent prior knowledge and perform knowledge-guided optimizations in the GMR framework.
- *River Modeling.* This is the first work to apply model revision to modeling a river system. Previous work on river modeling used model calibration alone.
- *Effectiveness.* Our framework achieves the best forecasting accuracy in river modeling among a variety of methods, while producing models consistent with domain knowledge.
- *Efficiency.* We present techniques to cut down the computational cost of GP systems, achieving $607\times$ speedup.

Reproducibility: Code and data are available at https://www.cs.cmu.edu/~namyongp/gmr.

The rest of the chapter is organized as follows. We describe the river modeling problem in Section 7.2, and present the GMR framework and how to apply it to river modeling in Section 7.3. After presenting experimental results in Section 7.4, we review related works in Section 7.5, and conclude in Section 7.6.

7.2 River Water Quality Modeling

Rivers are precious freshwater resources for households, farming, and industry. Due to intensive use and increasing development, the eutrophication (over-enrichment with nutrients) of rivers has become a serious global problem. Algal blooms are one of the most problematic and widespread consequences that deteriorate river water quality [Mos09]. For improved river management, it is crucial to have an accurate model of the water

quality.

River water quality modeling aims to predict phytoplankton biomass, a proxy for eutrophication. Based on the knowledge of a freshwater ecologist, we designed the following biological processes, modeling the change of phytoplankton biomass over time by capturing the interplay between phytoplankton (B_{Phy}) and zooplankton (B_{Zoo}).

$$
\begin{aligned}
\frac{dB_{Phy}}{dt} &= B_{Phy} \cdot (\mu_{Phy} - \gamma_{Phy}) - B_{Zoo} \cdot \varphi \qquad (7.1)\\
\mu_{Phy} &= C_{UA} \cdot f(V_{lgt}) \cdot g(V_n, V_p, V_{si}) \cdot h(V_{tmp})\\
\gamma_{Phy} &= C_{BRA}\\
\varphi &= C_{MFR} \cdot \lambda_{Phy}\\
\lambda_{Phy} &= (B_{Phy} - C_{Fmin})/(C_{FS} + B_{Phy} - C_{Fmin})\\
f(V_{lgt}) &= (V_{lgt}/C_{BL}) \cdot e^{1-(V_{lgt}/C_{BL})}\\
g(V_n, V_p, V_{si}) &= \min\left(V_n/(C_N + V_n), V_p/(C_P + V_p), V_{si}/(C_{SI} + V_{si})\right)\\
h(V_{tmp}) &= \max(e^{-C_{PT}(V_{tmp} - C_{BTP1})^2}, e^{-C_{PT}(V_{tmp} - C_{BTP2})^2})\\
\frac{dB_{Zoo}}{dt} &= B_{Zoo} \cdot (\mu_{Zoo} - \gamma_{Zoo} - \delta_{Zoo}) \qquad (7.2)\\
\mu_{Zoo} &= C_{UZ} \cdot \lambda_{Phy}\\
\gamma_{Zoo} &= C_{BRZ} + C_{BMT} \cdot \varphi\\
\delta_{Zoo} &= C_{DZ}
\end{aligned}
$$

The phytoplankton dynamics model (dB_{Phy}/dt) incorporates the photosynthetic productivity (μ_{Phy}), metabolic degradation (γ_{Phy}), and grazing pressure of zooplankton (φ). The photosynthetic productivity depends on multiplicative influences from variables such as light intensity (V_{lgt}), nutrient (nitrogen, phosphorus, and silica) concentrations (V_n, V_p, V_{si}), and water temperature (V_{tmp}). These functions build on earlier studies on modeling algal dynamics including [CS98, HJ02]. Further, considering the effect of summer cyanobacteria and winter diatom blooms, we extend the process with two additional parameters reflecting optimal temperatures (C_{BTP1}, C_{BTP2}). The zooplankton dynamics model (dB_{Zoo}/dt), adapted from [HJ02], incorporates the growth (μ_{Zoo}), respiration (γ_{Zoo}), and death (δ_{Zoo}) rates of zooplankton.

The parameters of these biological processes fall into two classes: *constant parameters* (starting with C) have constant values representing physiological rates (e.g., growth or feeding rate), while *variable parameters* (starting with V) correspond to external conditions and forces, changing over time. In evaluating (7.1) and (7.2), variable parameters are imported from the observed data at the evaluation time t. More details on these constant and variable parameters are given in Tables 7.3 and 7.4.

The goal of model revision for our task is summarised as:

Given biological processes (7.1) and (7.2), make relevant changes to the structure and constant parameter values of (7.1) and (7.2) guided by prior knowledge such that the estimated phytoplankton biomass (B_{Phy}) is close to the observed values, and the revised process is consistent with prior knowledge.

River modeling is a challenging task. Although carefully built with domain knowledge, manually-designed processes (7.1) and (7.2) (MANUAL) exhibit poor predictive performance, as shown in Figure 7.1. While the results of model calibration methods show that parameter tuning greatly improves modeling accuracy, it is not enough to be able to update only the model parameters. By improving the process itself via model revision, our method obtains the best result, with 13% and 34% smaller MAE than is obtained with the second best method and the best model calibration result, respectively, while producing revised processes guided by prior knowledge.

7.3 Methods

In this section, we present our genetic model revision (GMR) framework. There are three major challenges in applying model revision to the modeling of dynamical systems.

1) **Representation of dynamic processes.** Given differential equations that model dynamic processes, such as the one underlying river water quality ((7.1) and (7.2)), how can we represent them for successful model revision?
2) **Mechanism for knowledge-guided model revision.** Model revision requires defining specific steps for making revisions to obtain a better model. How can we effectively perform model revision guided by prior knowledge?
3) **Efficient and effective model revision.** Real-world systems are complex, often incurring high computational cost. How can we perform efficient and effective model revision?

In the GMR framework, we address these challenges with the following ideas.

1) **Using tree-adjoining grammar (TAG) for representing dynamic processes** provides a powerful framework to succinctly express dynamic processes and their revision, while facilitating controlled incorporation of prior knowledge.
2) **Making revision via TAG-guided GP and expressing prior knowledge using the TAG formalism** leads to an accurate model consistent with prior knowledge.
3) **Removing redundancy, speeding up operations, and local search** enable fast and effective model revision.

We describe how to represent dynamic processes using TAG in Section 7.3.1, and present the GMR framework in Section 7.3.2. Then we demonstrate how to apply GMR to real-world problems, such as river modeling, in Section 7.3.3, and present techniques to improve efficiency and effectiveness in Section 7.3.4.

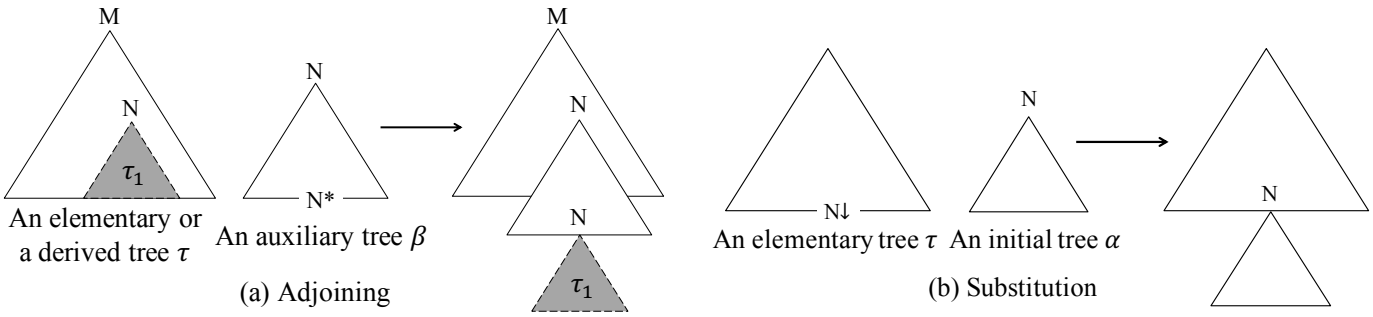


Figure 7.2: Illustrations of tree composition operations used by tree-adjoining grammar (TAG): (a) adjoining and (b) substitution.

7.3.1 Representing Dynamic Processes Using TAG

7.3.1.1 Preliminaries on TAG (Tree-Adjoining Grammar)

A TAG is a tree generating system [JS97], consisting of a quintuple (T, N, I, A, S) where

- T is a finite set of terminal symbols;
- N is a finite set of non-terminal symbols ($N \cap T = \emptyset$);
- $S \in N$ is a non-terminal symbol called the *start symbol*;
- I is a set of finite trees called *initial trees* or α-*trees*;
- A is a set of finite trees called *auxiliary trees* or β-*trees*.

Figures 7.2 and 7.3 provide illustrations of α- and β-trees. The trees in $I \cup A$ (i.e., α- and β-trees) are referred to as *elementary trees*. In an elementary tree, the labels of all interior nodes are non-terminal symbols, while the labels of the nodes on the frontier can be either terminal or non-terminal symbols. The frontier nodes of an elementary tree with non-terminal symbols are marked as $\downarrow$ for substitution, except for one special node in an auxiliary tree, which is called the *foot node* and marked with an asterisk ($*$) by convention. The foot node must have the same non-terminal symbol as that of the corresponding tree's root node (e.g., see Figure 7.3(b)).

Adjoining and *substitution* are the two composition operations TAG uses to construct a *derived* tree (Figure 7.2). Adjoining builds a new tree given an auxiliary tree β and a tree τ (which can be either an elementary or a derived tree). Assume that the root of β is labeled as N, and that the tree τ has an interior node n labeled as N. The steps for adjoining β into τ are as follows (see Figure 7.2(a) for an illustration):

1) The sub-tree τ_1 rooted at node n is disconnected from τ;
2) The tree β is attached at the place where the node n was;
3) τ_1 is attached to the foot node (marked with $*$) of the tree β.

Substitution creates a derived tree from an elementary tree τ and a tree α which is (derived from) an initial tree. As in Figure 7.2(b), substitution selects a non-terminal on the frontier of the tree τ (marked as $\downarrow$) matching the root of α, and replaces it with α.

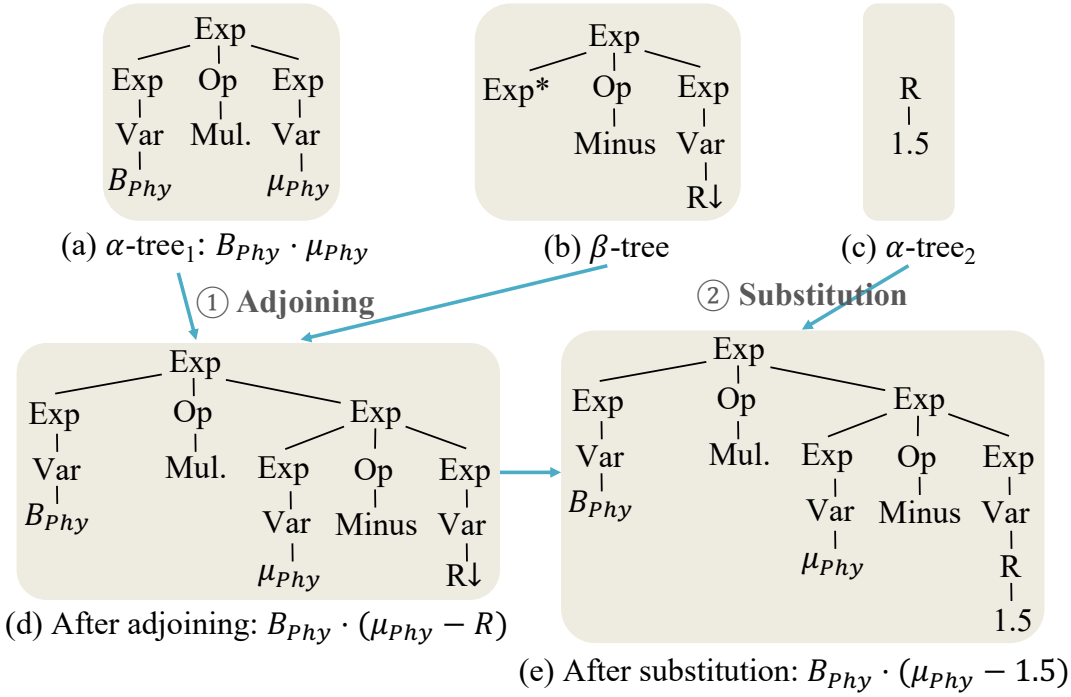


Figure 7.3: (a)–(c): Example α- and β- trees representing a dynamic process and potential revisions. (d), (e): Resulting trees after adjoining and substitution (see text for details).

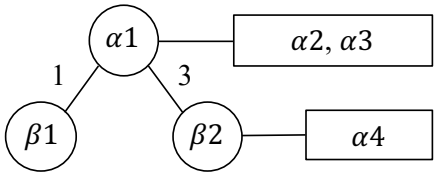


Figure 7.4: TAG derivation tree which encodes a revised differential equation. Two nodes (labeled by $\beta 1$ and $\beta 2$) are β-trees that are adjoined into the specified address (the number on the link) of the root. Rectangles contain α-trees (lexemes), which are substituted into the open nodes in the linked tree.

A tree derived from an initial tree, and lacking frontier non-terminals, is a *completed* tree.

An in-depth description of TAG appears in [JS97, Ngu04].

7.3.1.2 TAG-Based Dynamic Process Representation

Consider this equation, a simplified form of (7.1), to see how TAG can represent dynamic processes and potential revisions:

$$\frac{dB_{Phy}}{dt} = B_{Phy} \cdot \mu_{Phy} \tag{7.3}$$

(7.3) can be represented by an α-tree shown in Figure 7.3(a) where "Mul." denotes multiplication. Figure 7.3(b) shows a β-tree representing one potential extension where an expression (denoted by "Exp") is extended by deducting a random variable (denoted by "R") from it. Then adjoining the β-tree in Figure 7.3(b) into the rightmost "Exp" node of the α-tree in Figure 7.3(a) yields the tree shown in Figure 7.3(d), which corresponds to $B_{Phy} \cdot (\mu_{Phy} - R)$. Another α-tree in Figure 7.3(c) encodes a potential value for variable R. By substituting it into the frontier node R (marked with $\downarrow$) shown in Figure 7.3(d), we obtain a revised process:

$$\frac{dB_{Phy}}{dt} = B_{Phy} \cdot (\mu_{Phy} - 1.5). \tag{7.4}$$

A completed tree (e.g., Figure 7.3(e)) corresponds to a revised process. The history of adjunctions and substitutions is encoded as an object tree called the *derivation tree*. In other words, we encode successive model revisions and the revised process as a derivation tree in TAG. Among several proposed definitions of TAG derivation tree, we use the formulation with restricted substitution [Ngu04]:

1) The root node is labeled with an α-tree (i.e., input process) whose root node is labeled by the start symbol S.
2) All other nodes are labeled with β-trees (adjunction nodes). An adjunction node is associated with an address of the node at which the adjunction took place.
3) An α-tree that is substituted is restricted to have no children, which allows us to regard substitution as an in-node operation, and also simplifies the derivation tree greatly.

With this definition, the TAG derivation tree in GMR (Figure 7.4) is a tree of objects where links between objects indicate adjunction at the specified address, and each node has a list of α-trees (called *lexemes*) to be substituted into the open nodes (called *lexicons*) in the elementary tree labeled by the node.

Note that while the above discussion describes how TAG provides a mechanism for representing and revising dynamic processes, we need a more careful design of α- and β-trees than is shown in Figure 7.3, to be able to make controlled changes. For example, in order to reflect prior knowledge, we may want to adjoin the β-tree in Figure 7.3(b) into only one of the Exp nodes in Figure 7.3(a). However, with the β-tree in Figure 7.3(b), adjoining can happen at any of them.

7.3.2 Knowledge-Guided Genetic Model Revision

In this section, we describe how genetic model revision is performed in our framework, and discuss how we incorporate the prior domain knowledge on (1) plausible processes, (2) plausible revisions, and (3) the model parameters.

7.3.2.1 Framework Overview

Figure 7.5 shows an overview of the genetic model revision (GMR) framework. It builds upon the tree-adjoining grammar guided genetic programming (TAG3P) [NMA03].

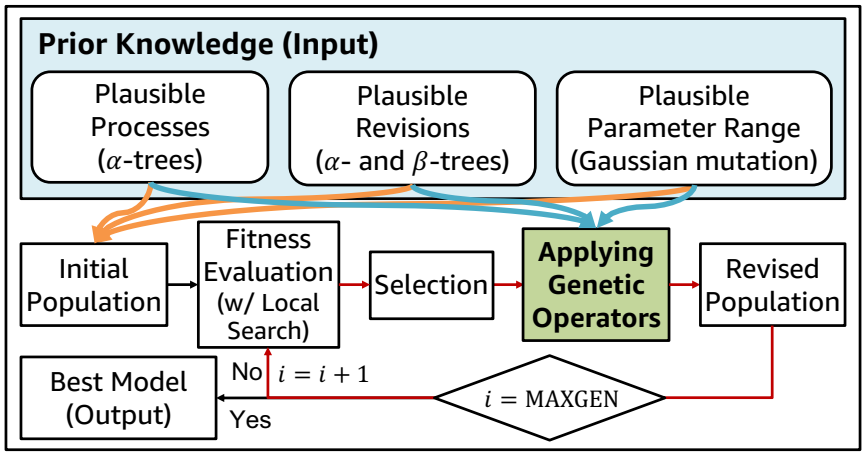
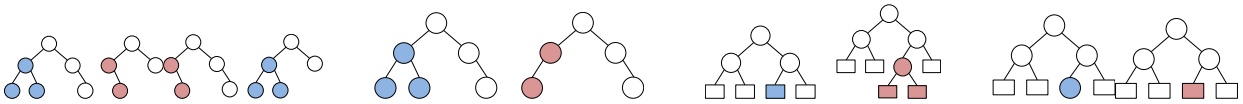


Figure 7.5: An overview of the model revision framework. Red loop denotes one generation. Prior knowledge shown in rounded boxes guides the entire model revision process.



(a) Before crossover (b) After crossover (c) Before subtree mutation (d) After subtree mutation (e) Before point insertion (f) After point insertion (g) Before point deletion (h) After point deletion

Figure 7.6: An illustration of genetic operators in TAG3P.

TAG3P is a population-based optimization algorithm that evolves a population of random (often unfit) initial programs (which are differential equations in our setting) into fitter ones for a given task, over multiple generations. TAG3P differs from standard GP in that it is a grammar-guided GP system where search space exploration is guided by TAG. In Figure 7.5, the loop marked in red corresponds to one generation. At each generation, genetic model revision is performed on the current population by applying genetic operators to produce a revised population of potentially fitter individuals. Note that three types of prior knowledge (shown in rounded boxes) given to the framework govern the entire search process, from population initialization to iterative model revision, until a final model is obtained.

7.3.2.2 Framework Components

Here we detail components of our TAG3P-based framework. A detailed review of the TAG3P system can be found in [NMA03, Ngu04].

Representation for a program (called an *individual*). In our setting, a program denotes differential equations that are to be revised (e.g., biological process in (7.1) and (7.2)). In TAG3P, each program is represented as a derivation tree (Figure 7.4).

α- and β-trees. In GMR, one α-tree is used to encode manually-designed minimal process as in Figure 7.3(a), while other α-trees define model revision via substitution as in Figure 7.3(c). β-trees are defined to represent potential revisions as in Figure 7.3(b). We provide more discussion on how to design these trees in Section 7.3.2.3.

Parameters. Parameters include population size (POPSIZE), maximum number of generations (MAXGEN), minimum size (MINSIZE) and maximum size (MAXSIZE) of individuals, and the probability of genetic operators.

Population Initialization. Individuals are created repeatedly until the population size reaches POPSIZE: TAG3P selects an individual size between MINSIZE and MAXSIZE, chooses an initial derivation tree randomly from α-trees, picks up β-trees and their adjoining addresses at random, and performs adjoining to generate an individual for the first generation.

Fitness Evaluation. An individual (a derivation tree) is transformed into a derived tree, and evaluated as in standard GP. In dynamic systems modeling, this involves evaluating revised differential equations (e.g., Figure 7.3(e)) for each time step, and comparing it with observed values. More accurate individuals are given higher fitness.

Genetic Operators. Genetic operators make revisions to the current population to obtain others. Among several operators in TAG3P, we introduce two representative ones (Figure 7.6 illustrates these operations).

(i) Crossover. Two individuals are chosen by a selection mechanism, and their subtrees are randomly selected and checked whether they are compatible. Subtrees are compatible if each subtree can be adjoined into the node where the other subtree is attached to. If so, the two subtrees are swapped. Otherwise, the previous process is retried unless the retry count has reached some predefined limit.

(ii) Subtree Mutation. A subtree x is randomly selected, and is replaced with a new subtree, which is of similar size to x, and compatible with x (to produce a valid individual).

7.3.2.3 Incorporating Prior Knowledge

By exploring the search space of both the structure and parameters of dynamic processes, complex real-world systems can be modeled more accurately than can be achieved with parameter optimization alone. However, the resulting process might be physically implausible and violate domain knowledge. Our framework learns an accurate model that complies with prior knowledge by incorporating three types of prior knowledge.

Prior Knowledge of Plausible Processes. In an effort to explain real-world phenomena, experts develop models based on domain knowledge and experience. We harness this prior knowledge of dynamic processes, specifically what variables are known to be involved and how they interact with each other. For example, the temporal dynamics of phytoplankton (dB_{Phy}/dt) in (7.1) is expressed as a function of zooplankton biomass (B_{Zoo}) (and other related parameters) since zooplankton grazing pressure is known to

be a major regulator of phytoplankton in a river ecosystem. In our framework, this first type of knowledge, expressed as a differential equation, is encoded as an α-tree as shown in Figure 7.3(a). Note that these input processes act as a significant knowledge transfer at the starting point of model revision. With classic GP systems, by contrast, we need to start from random models.

Prior Knowledge of Plausible Revisions. Real-world dynamic systems are complex, often consisting of multiple intertwined processes (e.g., dB_{Phy}/dt and dB_{Zoo}/dt), each of which can be further decomposed into multiple subprocesses in a nested manner (e.g., μ_{Phy} in (7.1), which represents photosynthetic growth). Domain experts often have an understanding of these subprocesses, such as the functionality of the subprocess, and plausible variables that may play a role for the subprocess. For instance, variables that are known to affect photosynthetic growth include water alkalinity and the amount of dissolved oxygen. Note that this subprocess corresponds to a subtree in the α-tree representing the input process.

Our framework allows specifying variables and operations applicable for revising a specific subprocess. This type of constraint is expressed in both α- and β-trees. In an α-tree, extensible subtrees are placed under a special node, whose name starts with "Ext", denoting a revision that can be made to the corresponding subtree via tree-adjoining. In an equation form, we denote a subprocess $f(\cdot)$ extensible via "Ext" by $\{f(\cdot)\}\odot$Ext. We then generate a list of β-trees for each combination of variables and operators, which have the corresponding "Ext" as the root node, and a foot node of the same type to define allowable operations for the given subtree.

Importantly, we distinguish between the operators that are applied directly to the initial process (called *connectors*), and those that are applied to the subprocesses, which extend the initial process, but do not belong to the initial process (called *extenders*). This is to preserve the initial process by applying a limited set of operations to it, while giving a greater freedom for extenders to make improvements to the initial process.

As an example, consider again this simplified process $dB_{Phy}/dt = \{B_{Phy} \cdot \mu_{Phy}\} \odot$ Ext, extensible via Ext. This can be encoded as an α-tree in Figure 7.7(a). Note that Ext_c (denoting connectors) is used for the root node. We assume that we have two variables B_{Zoo} and R, deduction (Minus) as a connector and multiplication (Mul.) as an extender. Figure 7.7(b) and (c) show two β-trees that can be generated (Ext_c and Ext_E denoting a connector and an extender). Adjoining Figure 7.7(a) and (b) produces $B_{Phy} \cdot \mu_{Phy} - B_{Zoo}$ in Figure 7.7(e); subsequently adjoining Figure 7.7(c) and substituting Figure 7.7(d) into Figure 7.7(e) yields a revised process $B_{Phy} \cdot \mu_{Phy} - B_{Zoo} \cdot 1.5$ in Figure 7.7(f) (extensions removed for brevity). Note that since connectors and extenders use different symbols (Ext_c and Ext_e), connector β-trees cannot adjoin into α-tree nodes with extender symbols, and vice versa.

Prior Knowledge about Model Parameters. From previous research and experience on dynamical systems, domain experts often have the information on the plausible distributions of the model parameters. Even if we obtain a highly accurate model, if

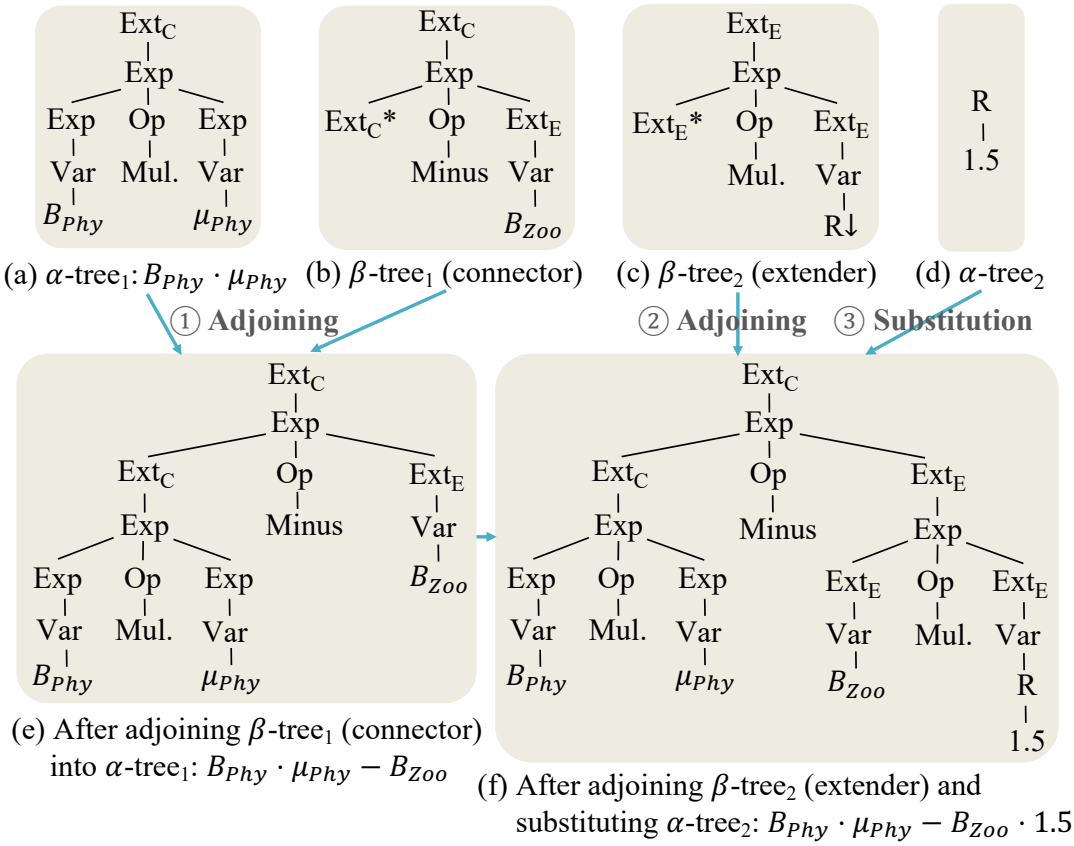


Figure 7.7: (a)–(d): α- and β-trees for (a) the initial process, revision via adjoining with (b) a connector and (c) an extender, and (d) substitution. (e), (f): Resulting trees after revision via a connector and an extender (see text for details).

its parameters are not within a realistic range, that model is not considered a good representation of underlying processes. In GMR, the domain knowledge of model parameters is summarized as the expected value and allowed range of parameter values. For effective search, ranges need to be chosen to cover most practically feasible values. We assume that naturally occurring values follow a truncated Gaussian distribution centered around the expected value.

To optimize model parameters, we apply a genetic operator, *Gaussian mutation*, which locates all constant parameters in an individual, and updates them to new values sampled from their associated Gaussian distribution. In the beginning, parameters are set to the expected value. When Gaussian mutation is applied to a parameter, a new value is generated, and it becomes the new mean of the Gaussian distribution for that parameter. If the sampled value lies outside of the given range, the boundary value is used instead.

For river water quality modeling, we initially set the standard deviation to $1/4$ of the parameter mean, as that covers the range of most observable parameter values. We then ramp down the standard deviation linearly in the final k generations so that it becomes

Table 7.2: Variables, connectors, and extenders used by extensions. R denotes a random variable between 0 and 1.

Extension	Variables	Extension	Variables	Extension	Variables
Ext1	$V_{cd}, V_{ph}, V_{alk}, R$	Ext5	V_{tmp}, R	Ext8	V_{tmp}, R
Ext2	V_{sd}, R	Ext6	V_{tmp}, R	Ext9	V_{tmp}, R
Ext3	$V_{do}, V_{ph}, V_{alk}, R$	Ext7	V_{tmp}, R		
Connectors	$+$ for extensions 1–3, $\times$ for extensions 5–9				
Extenders	$+, -, \times, \div, \log, \exp$ for all extensions				

smaller in later generations.

7.3.3 Applying GMR to Real-World Problems

River Water Quality Modeling. We show how the proposed framework can be applied to the modeling of river water quality introduced in Section 7.2, in which the goal is to make an accurate temporal prediction of the phytoplankton biomass (B_{Phy}). We capture the expert knowledge on the dynamics of phytoplankton (dB_{Phy}/dt) and zooplankton (dB_{Zoo}/dt), and plausible revisions as follows.

$$
\begin{aligned}
\frac{dB_{Phy}}{dt} &= \{B_{Phy} \cdot (\mu_{Phy} - \gamma_{Phy}) - B_{Zoo} \cdot \varphi\} \odot \text{Ext1} \\
\mu_{Phy} &= \{C_{UA} \cdot f(V_{lgt}) \cdot g(V_n, V_p, V_{si}) \cdot h(V_{tmp})\} \odot \text{Ext3} \\
\gamma_{Phy} &= \{C_{BRA}\} \odot \text{Ext5} \\
\varphi &= \{C_{MFR} \cdot \lambda_{Phy}\} \odot \text{Ext6} \\
\lambda_{Phy} &= (B_{Phy} - C_{Fmin})/(C_{FS} + B_{Phy} - C_{Fmin}) \\
f(V_{lgt}) &= (V_{lgt}/C_{BL}) \cdot e^{1-(V_{lgt}/C_{BL})} \\
g(V_n, V_p, V_{si}) &= \min\left(V_n/(C_N + V_n), V_p/(C_P + V_p), V_{si}/(C_{SI} + V_{si})\right) \\
h(V_{tmp}) &= \max(e^{-C_{PT}(V_{tmp} - C_{BTP1})^2}, e^{-C_{PT}(V_{tmp} - C_{BTP2})^2})
\end{aligned}
\tag{7.5}
$$

$$
\begin{aligned}
\frac{dB_{Zoo}}{dt} &= \{B_{Zoo} \cdot (\mu_{Zoo} - \gamma_{Zoo} - \delta_{Zoo})\} \odot \text{Ext2} \\
\mu_{Zoo} &= \{C_{UZ} \cdot \lambda_{Phy}\} \odot \text{Ext7} \\
\gamma_{Zoo} &= \{C_{BRZ}\} \odot \text{Ext8} + C_{BMT} \cdot \varphi \\
\delta_{Zoo} &= \{C_{DZ}\} \odot \text{Ext9}
\end{aligned}
\tag{7.6}
$$

Table 7.2 presents the list of variables, connectors, and extenders applicable to each extension. R denotes a variable that is randomly initialized. Constant parameters (those starting with C) are initialized and updated via Gaussian mutation, based on their mean and exploration range given in Table 7.3.

These extensions are defined based on an extensive river modeling experience of a freshwater ecologist, denoting different types of extensions plausible for specific subprocesses.

Table 7.3: Constant parameters that are updated via Gaussian mutation. Prior knowledge is captured as the mean and exploration bounds (minimum and maximum values).

	Description	Mean	Min	Max	Unit
C_{UA}	Max growth rate of phytoplankton	1.89	0.1	4.0	day^{-1}
C_{UZ}	Max growth rate of zooplankton	0.15	0.0	0.3	day^{-1}
C_{BRA}	Breath rate of phytoplankton	0.021	0.0	0.17	day^{-1}
C_{BRZ}	Breath rate of zooplankton	0.05	0.0	0.2	day^{-1}
C_{MFR}	Maximum feeding rate	0.19	0.01	0.8	day^{-1}
C_{DZ}	Death rate of zooplankton	0.04	0.01	0.1	day^{-1}
C_{FS}	Half-saturation constant of food	5.0	4.0	6.0	$\mu\text{g L}^{-1}$
C_{BTP1}	Blue-green optimal temperature	27.0	20.0	34.0	°C
C_{BTP2}	Diatom optimal temperature	5.0	1.0	20	°C
C_{Fmin}	Minimum food concentration	1.0	0.1	1.9	$\mu\text{g L}^{-1}$
C_{BL}	Best light for phytoplankton	26.78	24.0	30.0	$\text{MJ m}^{-2}\text{ d}^{-1}$
C_N	Half-saturation constant of nitrogen	0.0351	0.02	0.05	mg L^{-1}
C_P	Half-saturation constant of phosphorus	0.00167	0.001	0.02	mg L^{-1}
C_{SI}	Half-saturation constant of silica	0.00467	0.001	0.2	mg L^{-1}
C_{BMT}	Breath multiplier on grazing	0.04	0.01	0.07	N/A
C_{PT}	Temperature coefficient for phytoplankton growth	0.005	0.003	0.2	$°\text{C}^{-2}$
φ	Grazing rate of zooplankton	N/A	N/A	N/A	d^{-1}

For example, electric conductivity (V_{cd}) applies to the dynamics of phytoplankton via Ext1, but not to that of zooplankton.

Revising Multiple Processes. While input processes are to be represented using one α-tree, here we have two differential equations (7.5) and (7.6). Multiple equations can be encoded as a single α-tree by first representing each equation in separate trees, and then combining them into one α-tree under a new, common root node. Then this combined α-tree can be evolved in the same manner as in simpler cases, and decomposed into multiple equations when performing fitness evaluation.

Application to Other Problems. GMR provides general mechanisms to represent and revise process equations guided by prior knowledge, so is readily applicable to diverse problem settings. The only problem-dependent component is representing domain-specific knowledge as discussed in Section 7.3.2.3; this itself is a general technique applicable to various problems.

7.3.4 Improving the Efficiency and Effectiveness

For efficient and effective optimization, we apply three orthogonal speedup techniques, together with local search.

Evaluation Short-Circuiting. Modeling temporal processes involves incremental fitness evaluation over a period of time, in which case intermediate fitness may provide a

Table 7.4: Temporal variable parameters in the river process.

Parameter	Description	Parameter	Description
V_{lgt}	Irradiance (light intensity)	V_{do}	Dissolved oxygen
V_n	Nitrogen concentration	V_{cd}	Electric conductivity
V_p	Phosphorus concentration	V_{ph}	pH
V_{si}	Silica concentration	V_{alk}	Alkalinity
V_{tmp}	Water temperature	V_{sd}	Water transparency

reasonable estimate of the final fitness. Also, GP is known to be robust to noisy evaluation. Based on these observations, fitness evaluation can be short-circuited, using the estimate as a surrogate of the final fitness. Early termination is such an approach where fitness evaluation is stopped when the intermediate fitness gets worse than the previous best fitness obtained from full evaluations. In GMR, we design a generalized evaluation short-circuiting technique (Algorithm 7.1), which allows controlling the eagerness of early termination (via the *threshold* parameter) and use of different extrapolation methods.

Runtime Compilation. A tree representing temporal processes needs to be evaluated multiple times over some time period, and each such evaluation can be done by recursively evaluating subtrees, providing the model parameter values appropriately at each step. Instead, we use runtime compilation, which enables more efficient evaluation than repeated tree parsing: a program encoded in the tree is converted into the corresponding source code, compiled at runtime, and dynamically loaded to be used for fitness evaluation.

Tree Caching. We cache the results of tree evaluation, and reuse them when we need to reevaluate the same trees. By using additional memory to store evaluation results, we avoid redundant computations. Note that the effectiveness of caching depends on the hit rate. GMR improves the hit rate by algebraically simplifying the trees before they are evaluated. We show the benefits of speedup techniques in Section 7.4.6.

Local Search. Local search aims to improve the search effectiveness by making incremental, local revisions to an individual. We apply two local search operators, *insertion* and *deletion*. Insertion randomly chooses an open adjoining address of a TAG derivation tree, and adds a randomly selected compatible auxiliary tree to the chosen location. Deletion removes a random node from the derivation tree. Figure 7.6 illustrates these operations. Specifically, we perform stochastic hill-climbing local search, where a tree resulting from crossover and mutation goes through a series of local search, applying insertion and deletion with equal probability, and adopting the change if it improves the fitness.

7.4 Experiments

In this section, we evaluate our GMR framework via a case study on river modeling. We address the following questions.

Algorithm 7.1: Evaluation Short-Circuiting

Input: *ind* (an individual to be evaluated), *threshold* (a non-negative value that determines when to check for evaluation short-circuiting), *numFitcases* (number of fitness cases), EXTRAPOLATE (a function to extrapolate intermediate fitness).

Output: *fitness* (evaluated fitness of the given individual *ind*).

```
1  Function FITNESSEVALUATION (ind, threshold, numFitcases):
2      bestPrevFull ← ∞
3      fitness ← 0
4      i ← 0
5      while i < numFitcases
6          Update fitness of ind using fitness case i
7          if fitness > bestPrevFull × threshold then
8              estFitness ← EXTRAPOLATE(fitness, i, numFitcases)
9              if estFitness > bestPrevFull then
10                 return estFitness                          ▷ Short Circuiting
11         i ← i + 1
12     if fitness < bestPrevFull then
13         bestPrevFull ← fitness
14     return fitness                                         ▷ Full Evaluation
```

Q1. Prediction Accuracy: How accurately does the GMR framework forecast river water quality?

Q2. Ecological Analysis: How does GMR revise the input process, and does the revision make sense from an ecological standpoint? Which variables are important in the revision?

Q3. Speedup Techniques: How much do the speedup techniques improve efficiency?

7.4.1 Dataset and Modeling Task Description

The Nakdong River catchment in South Korea is one of the largest water-quality monitoring networks supporting long-term ecological research. Our dataset is a collection of measurements for 13 years (1996–2008) at nine stations located in the catchment; (Figure 7.8): six (S1–S6) are sited on the main channel, while three (T1–T3) are on major tributaries. The dataset contains five types of variables: geographical (e.g., catchment area), hydrological (e.g., flow rate), meteorological (e.g., irradiance), physicochemical (e.g., water temperature), and biological (e.g., chlorophyll a). Most were measured daily, except for nutrient concentrations and chlorophyll a, which were measured weekly (at S1) or bi-weekly (at others). For those variables measured with a longer interval, we performed linear interpolation to obtain values between measurements. Note that these measurements provide values for the temporal variables in the river process (i.e., those starting with V in (7.1), (7.2) and Table 7.2). Table 7.4 gives a description of the temporal variables. Given these measurements, our goal is to forecast the algal biomass at the lowest station (S1) due to its geographical importance (around ten million people live

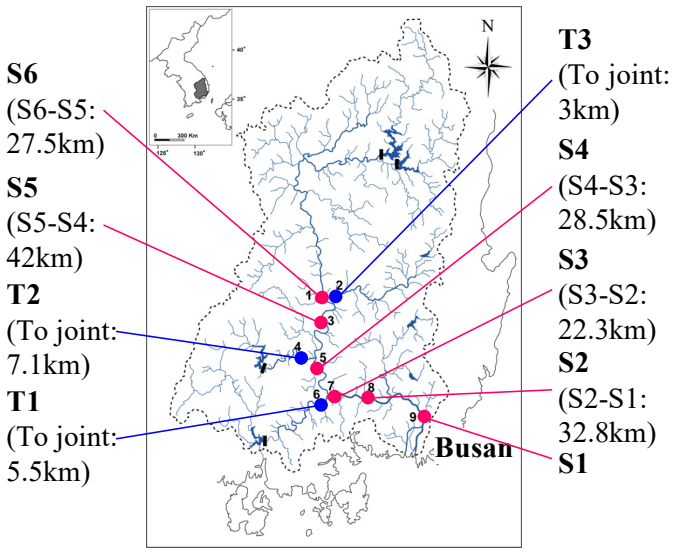


Figure 7.8: The Nakdong River basin in South Korea. Circles denote measuring stations. Red circles (S1–S6) are on the main channel, while blue ones (T1–T3) are on the major tributaries.

near S1). Specifically, we make knowledge-guided revisions to the *biological* process (7.1) and (7.2) to closely estimate observed algal biomass at S1.

In addition to the biological process, river modeling involves modeling the flow of water bodies in the river (called the *hydrological* process). What this hydrological process models includes how water bodies are discharged from stations, how the rainfall is absorbed into the river, etc. As our focus is on improving the biological process, we use a known hydrological process Section 7.7.1 gives details of the hydrological process, and how a river system with multiple stations is implemented.

7.4.2 Comparators

For evaluation, we use representative comparators in the four classes of methods for modeling dynamic systems.

7.4.2.1 Knowledge-Driven Modeling

(a) MANUAL. This is the biological process in (7.1) and (7.2), designed by domain experts.

7.4.2.2 Data-Driven Modeling

(a) RNN (Recurrent Neural Network). We use long short-term memory (LSTM), predicting the phytoplankton biomass at S1 at the next time step from observed variables at the current time. We experiment with two variants: RNN-S1 uses variables observed at station S1 alone; RNN-ALL uses variables observed at all nine stations. (b) ARIMAX is widely used for time series forecasting. As with RNN, we consider two variants differing in variables used (denoted as ARIMAX-S1 and ARIMAX-ALL).

7.4.2.3 Model Calibration

Given the biological process in (7.1) and (7.2), model calibration methods optimize the values of process parameters without revising the form of equations. We use the following widely-used approaches: (a) GA (genetic algorithm) (b) MC (Monte Carlo) (c) LHS (Latin hypercube sampling) (d) MLE (maximum likelihood estimation) (e) MCMC (Markov chain Monte Carlo) (f) SA (simulated annealing) (g) DREAM (differential evolution adaptive metropolis [Vru16]) (h) SCE-UA (shuffled complex evolution [DSG94]) (i) DE-MC$_Z$ (differential evolution Markov chain [VTBC+08]).

7.4.2.4 Model Revision

(a) GGGP (grammar guided GP). We perform model revision using GGGP; that is, GGGP receives the biological process in (7.1) and (7.2) as input, and updates both the model structure and parameter values.

Section 7.7.2 provides experimental settings of all methods.

7.4.3 Performance Evaluation

We use RMSE (root mean square error) and MAE (mean absolute error). Let y_t and $\hat{y}_t$ denote the observed and predicted values at time t, respectively. Given observations for T time steps, we have $\mathbf{y} = (y_1, \ldots, y_T)$ and $\hat{\mathbf{y}} = (\hat{y}_1, \ldots, \hat{y}_T)$. RMSE is a quadratic score which measures the average magnitude of prediction errors, giving a relatively large weight to large errors. RMSE is defined by $\sqrt{\frac{1}{T}\sum_{t=1}^{T}(\hat{y}_t - y_t)^2}$. MAE is a linear scoring rule for measuring the average magnitude of prediction errors, giving equal weights to individual differences. MAE is defined by $\frac{1}{T}\sum_{t=1}^{T}|\hat{y}_t - y_t|$. Both RMSE and MAE range from zero to ∞, and lower values indicate a better prediction.

Fitness Function. RMSE is used as a fitness function.

7.4.4 Q1. Prediction Accuracy

We split the data into two periods, 1996–2005 for training and 2006–2008 for testing, and report the forecasting accuracy in Table 7.5, in terms of the best RMSE and MAE where best models denote those with the smallest test RMSE.

MANUAL performed significantly worse than other approaches, although it is designed with domain knowledge of the biological process and a careful selection of parameter values. Model calibration approaches, such as GA, LHS, and SA, obtained a much better result than MANUAL, indicating the benefits of tuning model parameters. However, model calibration methods were outperformed by model revision methods (with GMR obtaining 32% and 34% smaller RMSE and MAE than the best model calibration results) as they can update only the model parameters, but not the model structure. In both criteria, GMR achieved the best testing performance, with 7% and 13% smaller RMSE and MAE, respectively, than the second best method GGGP.

Table 7.5: GMR achieves the best forecasting accuracy (7% and 13% more accurate than the second best method in terms of RMSE and MAE, respectively), among a variety of methods. Best results are underlined.

Method Class	Method	Training (96–05)		Test (06–08)	
		RMSE	MAE	RMSE	MAE
Knowledge-driven	MANUAL	2.79e+9	2.15e+8	2.23e+6	7.93e+5
Data-driven	RNN-S1	19.605	11.533	23.057	16.833
	RNN-ALL	21.326	13.166	23.009	16.276
	ARIMAX-S1	12.710	<u>5.012</u>	37.770	25.504
	ARIMAX-ALL	<u>12.365</u>	5.775	260.468	71.471
Model calibration	GA	26.329	14.693	20.308	13.291
	MC	26.581	14.426	19.259	12.675
	LHS	26.812	14.536	18.287	12.064
	MLE	26.033	14.408	19.513	13.242
	MCMC	26.514	14.554	18.661	12.480
	SA	26.463	14.585	18.740	12.532
	DREAM	26.825	14.853	19.281	12.581
	SCE-UA	25.995	14.353	19.876	13.275
	DE-MC$_Z$	26.227	14.432	18.904	12.869
Model revision	GGGP	20.741	11.316	13.248	9.158
	GMR	**21.427**	**11.966**	**<u>12.356</u>**	**<u>7.936</u>**

For data driven models, we used two types of input variables. Both variants of RNN and ARIMAX (denoted by S1 and ALL) performed worse than model calibration and model revision methods. While RNN's best test performance was worse than GMR, RNN could achieve much smaller training RMSE (~6.7) than others as training continued. However, it suffered from overfitting and its test RMSE increased to ~44.0. Note that for both RNN and ARIMAX, using additional input variables observed at stations other than S1 did not help improve the performance. In fact, for predicting phytoplankton at S1, ARIMAX-ALL performed worse than ARIMAX-S1. As measuring stations are located over a wide area (see Figure 7.8), using measurements from distant stations simply as additional input features was not helpful for predicting at S1. Also, as is typically the case with ecological data, the dataset is not large enough (2,435 data points for training) for learning complex processes in a purely data-driven manner. On the other hand, by using prior knowledge, GMR can learn an effective model from a small dataset, which is also consistent with domain knowledge.

7.4.5 Q2. Ecological Analysis

From the perspective of ecosystem management, it is critical to understand newly added mechanisms (i.e., extensions). However, many machine learning applications to

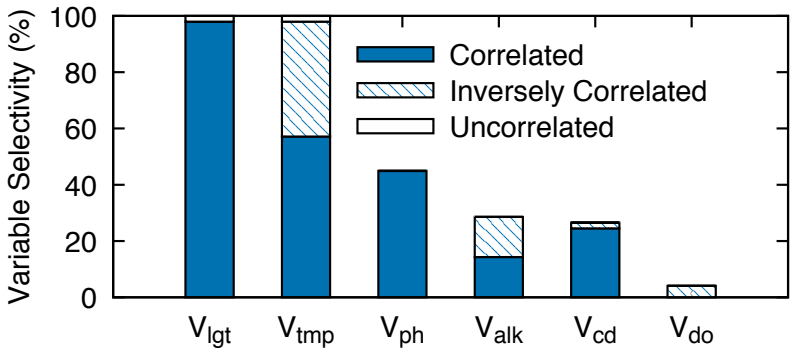


Figure 7.9: Selectivity of variables among the 50 best models.

environmental research employ uninterpretable, black-box models. In this section, we strive to understand to what extent the extensions reinforce meaningful information and variables, and enhance the predictive power of the initial process model.

Case Study. We examined the best models and found that several common variables were added to specific processes. First, the addition of temperature dependence was frequently observed, particularly in zooplankton, which is an algal grazer. (7.7) shows an example where a revision is highlighted.

$$\delta_{Zoo} = \{C_{DZ}\} \times (4V_{tmp} + 253.4) \tag{7.7}$$

This shows that the temperature is related to the metabolic rate of zooplankton and its mortality, and plays a key role on the prediction of algal blooms.

Second, pH was often connected with algal growth process, as in the following example:

$$\frac{dB_{Phy}}{dt} = \{B_{Phy} \cdot (\mu_{Phy} - \gamma_{Phy}) - B_{Zoo} \cdot \varphi\} + \frac{V_{alk}}{V_{ph} - V_{cd} + 848.4} \tag{7.8}$$

In modeling algal process in rivers, pH has not been often used in knowledge-based modeling as it is not known to be a limiting factor. Although pH is closely associated with aquatic carbon complex (which is also an essential component of photosynthesis, along with nitrogen and phosphorus), it has been neglected in modeling due to a plethora of carbon availability. However, it is remarkable to observe the improvement of predictive power when pH is considered as an input variable. In fact, recent machine learning models have illuminated pH as a crucial factor to predict algal blooms. This is one of the major discoveries by GMR in the context of river modeling, which corroborates the importance of pH in environmental research. Also, note that V_{alk} is determined by CO_3^{2-}, which is in turn related to pH, and V_{cd} is a reasonable selection as it is considered a proxy of pollutant concentration in freshwaters.

Relative Importance Analysis. To assess the relative importance of input variables, we

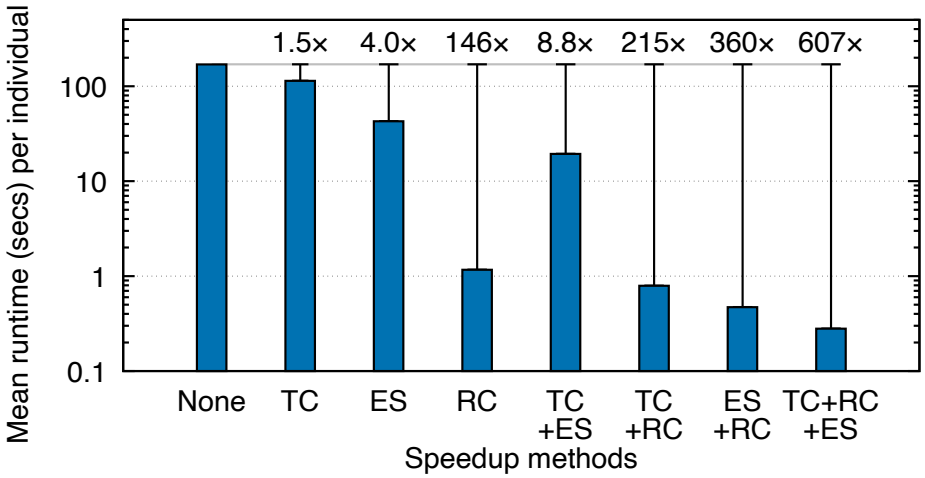


Figure 7.10: Mean runtime (seconds) per individual by speedup techniques. TC: Tree Caching, ES: Evaluation Short-Circuiting, RC: Runtime Compilation. Applying all leads to $607\times$ speedup.

explored the selectivity (%) of variables among the 50 best models, and the correlation of each variable with phytoplankton (B_{Phy}) growth via variable perturbation (Figure 7.9). Among them, light (V_{lgt}) and water temperature (V_{tmp}) were selected most often; this agrees with the fact that they are limiting factors for phytoplankton growth. Note that while V_{lgt} was consistently correlated with B_{Phy}, V_{tmp} was not. As water temperature has multiple optimal points for the best growth of B_{Phy}, V_{tmp} can affect B_{Phy} either positively or negatively depending on the dominance of ambient phytoplankton functional group. The third important factor, V_{ph}, is closely related to photosynthesis and oxygen release. Alkalinity (V_{alk}) and electric conductivity (V_{cd}) were selected similarly often: V_{cd}'s high correlation is plausible as conductivity can be a proxy of nutrient levels in freshwater. Dissolved oxygen (V_{do}) was negatively correlated with B_{Phy}, implying that phytoplankton biomass is lower in higher V_{do} concentrations. Overall, while a few exceptions exist, most selected ones were ecologically plausible and interpretable.

7.4.6 Q3. Analysis of Speedup Techniques

We apply three orthogonal methods for speedup, i.e., tree caching (TC), evaluation short-circuiting (ES), and runtime compilation (RC). Figure 7.10 shows the mean runtime (seconds) per individual obtained with different speedup methods, which indicates that these techniques effectively reduce computational costs, achieving $607\times$ speedup when all methods are applied together, compared to when no speedup techniques were used. We also measured how ES affects the performance when we use different thresholds (0.7, 1.0, and 1.3). Figure 7.11 shows relative values w.r.t. ES with a threshold of 1.0. Results indicate that ES generally cut down evaluation cost without sacrifice in accuracy, and nearly 100% of the best models were fully evaluated. While the overall RMSE increased by 5% as ES got eager (with a threshold of 0.7), the number of evaluated time steps dropped by 19%.

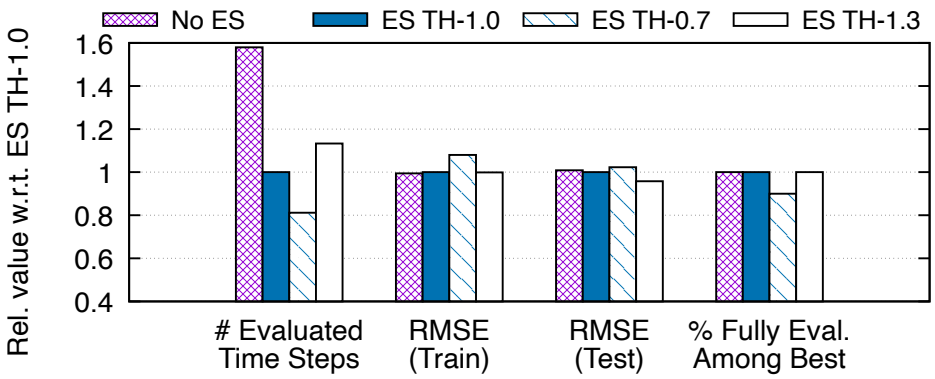


Figure 7.11: Effect of evaluation short-circuiting (ES) as thresholds (TH) are varied. Relative values w.r.t. ES TH-1.0 are reported.

7.5 Related Work

Scientific Discovery from Data. Interest in computational methods for making scientific discoveries from data is growing. Among data-driven approaches, symbolic regression (SR) using GP [SL09] aims to discover scientific laws from data without relying on prior knowledge. While SR searches for the form of equations and their parameters simultaneously as in genetic model revision (GMR), SR may learn models that are inconsistent with domain knowledge due to the lack of guidance in the optimization process.

Combining prior knowledge and data science [KAF+17] is an emerging paradigm with promising results, which can address this challenge. One approach along this line is to guide the learning algorithm, e.g., via theory-guided constrained optimization [KWRK17, KAF+17]. GMR also falls in this category, using TAG formalism for knowledge-guided navigation of the search space. Model calibration [KAF+17] is another such approach: data are used to optimize the parameters of a knowledge-based model. GMR outperforms various model calibration approaches, while learning knowledge-consistent processes.

Modeling River Water Quality. QUAL2E [BB87] is the most well-known model for river ecosystems. Despite its wide use, QUAL2E's accuracy was limited due to their assumption on steady-state flow. Neural network (NN) and genetic algorithm (GA) have recently been used for river modeling. NN-based methods perform data-driven modeling using multilayer perceptron [SBMJ09] and recurrent neural networks [KKNK18]. By performing model calibration, GA-based methods [KMS+10, KPM+14] successfully improved the accuracy of a knowledge-based river model. However, existing evolutionary methods do not have mechanisms to jointly optimize the structure and parameters of a model, in particular, guided by domain knowledge. To fill this gap, we design novel mechanisms to represent prior knowledge and perform knowledge-guided model revision in TAG3P. As the first work on modeling river water quality using knowledge-guided model revision, our work improves upon earlier works that made limited or no

use of prior knowledge.

7.6 Conclusion

Model revision is an effective approach for modeling real-world phenomena where domain expertise and data are used simultaneously to model complex dynamical systems. Our genetic model revision (GMR) framework performs model revision guided by prior knowledge. The case study of river modeling shows its effectiveness. In future work, we will explore new mechanisms to incorporate domain knowledge (new search operators and language biases), and apply GMR to other domains, such as financial forecasting.

Reproducibility: Code and data are available at https://www.cs.cmu.edu/~namyongp/gmr.

Extensibility: How readily can the ideas behind this work be extended to other domains? The degree of effort required depends on the similarity to the present problem. At one extreme, the underlying ideas are applicable to most model identification problems where expert knowledge is available but incomplete. TAG grammars are particularly suited to this, because they readily match the common situation where experts have a simple model that they believe to be generally correct, but potentially to require modification because potentially important processes have been omitted for simplicity. The adjunction operation of TAG3P is particularly suited to recording the way experts think about such problems. Similarly, it will frequently be the case that experts can define feasibility bounds on model parameters. In real world problems, it is frequently the case that evolving models need to be repeatedly evaluated to estimate their fitness. In such cases, the speedup techniques we have described here may well be applicable. In all these circumstances, portions of the code we have made available may be useful for implementation.

At the other extreme, the system can be directly applied to other river systems under the assumption of conservative, non-branching flow, provided that the corresponding data is available. The system also relies specifically on the G++ compiler suite to provide run-time compilation. Similar capabilities are available in other C++ compilers, or run-time compilation can be sacrificed at a substantial cost in speed.

Where the data differs in the water properties collected, revisions in the search limiting grammar will be required, and prior knowledge of the feasible values of any new parameters will need to be specified.

Extensions to river systems with substantial water loss through evaporation or leakage, or to braided streams or delta, require significant re-programming of the flow model, and additional data inputs for those components.

Moving beyond rivers, much of the structure of the system is determined by the need to model variables that change as a fluid flows through a network. It is not hard to find other problems that fit that bill: flow of blood through an organism; flow of feedstocks

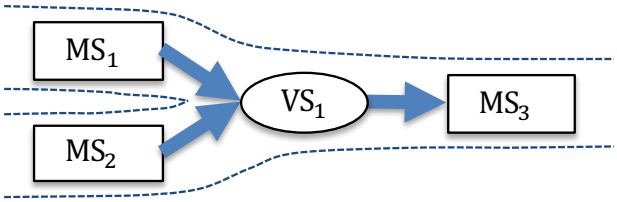


Figure 7.12: An example river system with measuring stations (MS) and a virtual station (VS). VS is placed at a confluence.

through a chemical plant; municipal water or sewage systems. In all these cases, the existing code would form a useful starting point.

7.7 Appendix

7.7.1 Further Details of River Modeling

Modeling A River System. To monitor the ecological status of a river, sample measurement is performed, often on a regular basis, at multiple measuring stations located in geographically important places. Based on this data collection scheme, we model a river system as a directed acyclic graph as shown in Figure 7.12, where a node corresponds to a measuring station and an edge denotes a segment of a river between the two adjacent stations. To model the confluence, we add a virtual station where two or more water bodies meet together as depicted in Figure 7.12 (VS_1). We model our study site shown in Figure 7.8 by adding six stations (S1–S6) at the main channel of the river, three stations (T1–T3) at the major tributaries of the river, and three virtual stations at the confluence of a tributary with the main stream (S6 · T3, S4 · T2, and S3 · T1).

Hydrological Process. Modeling a river system involves modeling two contemporaneous processes, the *biological* and *hydrological* processes. The biological process shown in (7.1) and (7.2) models the evolution of phytoplankton and zooplankton and their interaction. The hydrological process models the flow of water bodies, and provides information of flow at specific time to the biological process. We use the hydrological process first introduced in [KMS+10], which is based on a flow mass balance between stations. Specifically, given that water flows from station A to station B, flow into B consists of three components, (i) inflow from upper station A, (ii) water retained at B (e.g., due to water trapped in side pools or non-laminar flow), and (iii) runoff into B from precipitation:

$$F_{B,t+\Delta} = r_B \cdot F_{B,t} + (1 - r_A) \cdot F_{A,t} + R_{B,t+\Delta} \tag{7.9}$$

where $F_{S,t}$ denotes the flow at station S at time t, r_S is the ratio of the water retained at station S, Δ is the time taken for water bodies from A to arrive at B, and $R_{S,t}$ denotes the amount of inflow into station B at time t that arises from rain fall. Our hydrological process also models how water bodies are merged at a confluence (e.g., VS_1 in Figure 7.12), in which case the attributes (e.g., nutrient level) of each water body are updated based on (7.9) as they are moving to the confluence, and the water bodies are aggregated as a flow-weighted average.

Thus, the hydrological process determines the attributes of a water body at a specific location and time by modeling how water bodies are merged, how the rainfall is absorbed into the river, etc. As the attributes of a water body are used by the biological process, the hydrological process affects the biological process. As we focus on modeling the biological process, we use a static hydrological process in this work.

Biological Process. We evolve a population of individuals (Figure 7.5), where each one is a revised biological process. Each individual receives values for the temporal variables (Table 7.4) from the water body that the hydrological process provides, and updates the phytoplankton (B_{Phy}) and zooplankton (B_{Zoo}) according to its process equation. Tables 7.3 and 7.4 list the variables used in our biological process.

7.7.2 Experimental Settings

Environment. We used an Ubuntu server with 80 Intel Xeon E7-4850 processors at 2.00GHz and 252GB RAM.

GMR, GGGP, and GA. We implemented GMR, GGGP, and GA frameworks in C++. For GMR, we used the following configurations: number of generations (100), population size (200), number of runs (60), minimization objective (RMSE), elite size (2), number of local search steps (5), selection mechanism (tournament selection), tournament size (5), minimum chromosome size (2), maximum chromosome size (50). We applied the following genetic operators (the number in the parentheses denotes the operator probability): crossover (0.3), subtree mutation (0.3), Gaussian mutation (0.3), and replication (0.1). For GGGP and GA, we used the same configurations used for GMR. Since local search incurs additional fitness evaluation in GMR, GGGP and GA used a population of 1200 individuals to use the same number of fitness evaluation for both methods.

RNN. We used a two-layer LSTM implemented in PyTorch, whose hidden size was equal to the number of input features. The output was transformed into an estimated phytoplankton value via dense neural networks with two layers. The input features were standardized. We used Adam optimizer with $\alpha = 0.01, \beta_1 = 0.9, \beta_2 = 0.999$, and weight decay of 0.0005. The model was trained for up to 1000 epochs with MSE loss.

ARIMAX. We used the *pmdarima* library and its Auto-ARIMA functionality with the default parameter settings.

Others. For other model calibration methods, we used the SPOTPY framework [HKCCB15] using RMSE as the objective function. We set method parameters to their default values.

Chapter based on work published in WSDM 2022 [PLM+22].

Chapter 8

Jointly Modeling Event Time and Network Structure for Reasoning over Temporal Knowledge Graphs

How can we perform knowledge reasoning over temporal knowledge graphs (TKGs)? TKGs represent facts about entities and their relations, where each fact is associated with a timestamp. Reasoning over TKGs, i.e., inferring new facts from time-evolving KGs, is crucial for many applications to provide intelligent services. However, despite the prevalence of real-world data that can be represented as TKGs, most methods focus on reasoning over static knowledge graphs, or cannot predict future events. In this chapter, we present a problem formulation that unifies the two major problems that need to be addressed for an effective reasoning over TKGs, namely, modeling the event time and the evolving network structure. Our proposed method EvoKG jointly models both tasks in an effective framework, which captures the ever-changing structural and temporal dynamics in TKGs via recurrent event modeling, and models the interactions between entities based on the temporal neighborhood aggregation framework. Further, EvoKG achieves an accurate modeling of event time, using flexible and efficient mechanisms based on neural density estimation. Experiments show that EvoKG outperforms existing methods in terms of effectiveness (up to *77%* and *116%* more accurate time and link prediction) and efficiency.

8.1 Introduction

How can we perform knowledge reasoning over knowledge graphs (KGs) that continuously evolve over time? KGs [JPC+20] organize and represent facts on various types

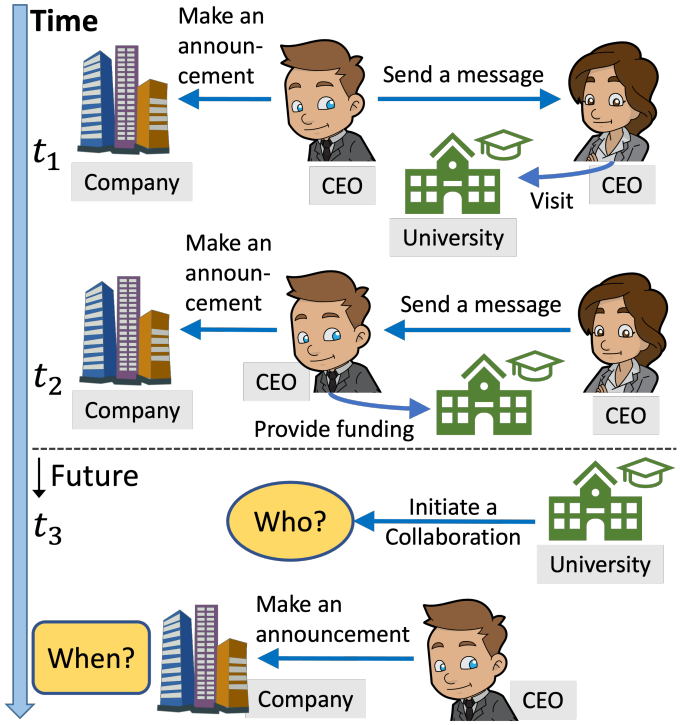


Figure 8.1: An example TKG, where we aim to predict temporal links and event time.

of entities and their relations. By facilitating an effective use of prior knowledge represented as a multi-relational graph, KGs power many important applications, including question answering, recommender systems, search engines, and natural language processing. Knowledge reasoning over KGs [CJX20], the process of inferring new knowledge from existing facts in KGs, lies at the heart of these applications, as KGs are typically incomplete, with many facts missing.

Importantly, real-world events and facts are often associated with time (i.e., occurring at a specific time or valid in limited time), exhibiting complex dynamics among entities and their relations that evolve over time. Such real-world data (e.g., ICEWS [BLO+15] and GDELT [LS13]) can be modeled as temporal knowledge graphs (TKGs), where entities are connected via timestamped edges, and two entities can have multiple interactions at different time steps, as illustrated in Figure 8.1. Despite the prevalence of real-world data that can be represented as TKGs, existing methods [YYH+15, SKB+18, DMSR18, SDNT19] have mainly focused on reasoning over static KGs, and lack the ability to employ rich temporal dynamics available in TKGs.

Recently, a few methods have been developed for reasoning over TKGs. They mainly address two problem setups, i.e., interpolation and extrapolation. Given a TKG ranging from time 0 to time T, methods for the interpolation setup [DRT18, GDN18, LC18] infer missing facts for time t $(0 \leq t \leq T)$; on the other hand, those for the extrapolation setup [TDWS17, TFBZ19, JQJR20] predict new facts for time $t > T$. In this chapter, we

focus on the extrapolation setting, which is more challenging and interesting than the other setting, as forecasting emerging events are of great importance to many applications of TKG reasoning.

In this chapter, we approach the problem of TKG modeling by defining the joint probability distribution of a TKG as a product of conditionals, from which we present a problem formulation that unifies the two problem settings of existing methods, namely, modeling the event time and evolving network structure. While addressing both problems leads to learning rich, complementary information useful for an effective reasoning over TKGs, most methods deal with only either of the two, as summarized in Table 8.1.

Therefore, in this chapter, we develop EvoKG, a method that jointly addresses these two core tasks for reasoning over TKGs. We design an effective framework that can be effectively applied to each task, with only minor adaptations. Our framework performs neighborhood aggregation in a relation- and time-aware manner, and carries out recurrent event modeling in an autoregressive architecture to capture the ever-changing structural and temporal dynamics over time (F1-F3 in Table 8.1). Importantly, EvoKG tackles the challenging task of event time modeling, using flexible and efficient mechanisms based on neural density estimation (T2-1 and T2-2 in Table 8.1), which avoids the limitations of existing methods that the learned distributions are not expressive, and that the log-likelihood and expectation of event time cannot be obtained in closed form, but instead require an approximation. In summary, our contributions are as follows.

- **Problem Formulation** (Section 8.2). We present a problem formulation that unifies the two major tasks for TKG reasoning—modeling the timing of events and evolving network structure.
- **Framework** (Section 8.3). We propose EvoKG, an effective and efficient method for reasoning over TKGs that jointly addresses the two core problems (T1 and T2 in Table 8.1).
- **Effectiveness** (Section 8.4). Experiments show that EvoKG achieves up to *116%* and *77%* better link and event time prediction accuracy, respectively, than existing KG reasoning methods (Figure 8.2).
- **Efficiency** (Section 8.4). EvoKG efficiently processes concurrent events, achieving up to *30*$\times$ and *291*$\times$ speedup in training and inference, respectively, compared to the best existing method.

Reproducibility. The code and data used in this chapter are available at `https://namyongpark.github.io/evokg`.

8.2 Problem Formulation

Notations. A temporal knowledge graph (TKG) G is a multi-relational, directed graph with timestamped edges. We denote a timestamped edge in TKG by a quadruple (s, r, o, t); it represents an event between subject entity s and object entity o, occurring at time t, where edge type (also called relation) r denotes the corresponding event type. In a TKG, we assume no duplicate edges, but there can be multiple edges of the same type

Table 8.1: **EvoKG wins.** EvoKG deals with both tasks (T1-T2) for reasoning over TKGs, while representative baselines fail to address both. EvoKG also possesses desirable features (F1-F3) for modeling TKGs. TD: TA-DistMult [GDN18]. EG: EvolveGCN [PDC+20]. KE: Know-Evolve [TDWS17]. RN: RE-Net [JQJR20].

	TD	EG	KE	RN	EvoKG
T1. Modeling evolving network structure	✓	✓		✓	✓
T2. Modeling event time t			✓		✓
• T2-1. Closed-form likelihood & expectation			✓		✓
• T2-2. Flexible approximation of $p(t)$					✓
F1. Relation-awareness	✓		✓	✓	✓
F2. Neighborhood aggregation		✓		✓	✓
F3. Recurrent event modeling	✓	✓	✓	✓	✓

between two entities, if they have different timestamps. For example, a TKG may have both ('u1', 'emailed', 'u2' '10 am') and ('u1', 'emailed', 'u2' '12 am').

Let (s_n, r_n, o_n, t_n) denote an n-th edge among a set of ordered edges. Given a TKG G with N edges sorted in non-decreasing order of time, we denote it by $G = \{(s_n, r_n, o_n, t_n)\}_{n=1}^{N}$ where $0 \leq t_1 \leq t_2 \leq \ldots \leq t_N$. We use G_t to denote a TKG consisting of events observed at time t, and $G_{<t}$ to refer to a TKG with all events observed before time t. We use e to refer to the event triple (s, r, o). We denote vectors by boldface lowercase letters (e.g., $\boldsymbol{c}$), and matrices by boldface capitals (e.g., $\mathbf{W}$).

Problem: Modeling a TKG. Given a TKG G with a sequence of observed events $\{(s_n, r_n, o_n, t_n)\}_{n=1}^{N}$, our goal is to model the probability distribution $p(G)$. We assume that events at time t depend on events that occurred prior to time t, and events that happen at the same time are independent of each other, given preceding events. Based on these assumptions, the joint distribution of TKG G can be written as:

$$p(G) = \prod_{t} p(G_t|G_{<t}) = \prod_{t} \prod_{(s,r,o,t)\in G_t} p(s, r, o, t|G_{<t}). \tag{8.1}$$

We further decompose the joint conditional probability $p(s, r, o, t|G_{<t})$ in Equation (8.1) as follows.

$$p(s, r, o, t|G_{<t}) = p(t|s, r, o, G_{<t}) \cdot p(s, r, o|G_{<t}) \tag{8.2}$$

Note that by modeling the two terms in Equation (8.2), we model the event time $p(t|s, r, o, G_{<t})$ and the evolving network structure $p(s, r, o|G_{<t})$. Based on this decomposition, we propose to model a TKG by estimating these two probability terms.

Surprisingly, existing methods for TKGs have focused on modeling either of the two terms, but not both at the same time, as summarized in Table 8.1. Methods that solve

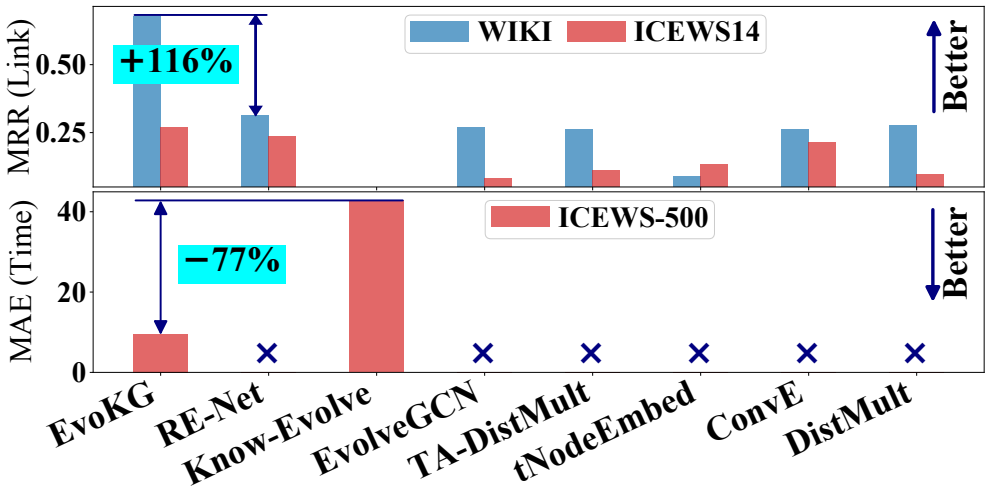


Figure 8.2: **EvoKG wins.** EvoKG achieves the best link prediction (top) and time prediction (bottom) results. × indicates that the corresponding method cannot predict event time.

only one of the tasks fail to utilize rich information that can be learned by addressing the other task: e.g., methods that do not model the event time (e.g., those marked with × in Figure 8.2) cannot predict when events will occur, and those that only model the event time cannot take the likelihood of an event triple (s, r, o) into account when estimating the likelihood of a timestamped event. By unifying these two modeling tasks, we can enable a more accurate reasoning over TKGs.

8.3 Modeling a Temporal Knowledge Graph

We describe how EvoKG models a TKG by addressing the two problems—modeling event time and evolving network structure. The symbols used in this chapter are listed in Table 8.2.

8.3.1 Modeling Event Time

The temporal patterns of events occurring between various types of entities in a TKG depend on the context of their past interactions. To capture intricate temporal dependencies present in real-world TKGs, we treat the event time t as a random variable, and model the occurrence of triple (s, r, o) at time t using temporal point processes (TPPs), which are the dominant paradigm for modeling events that occur at irregular intervals. Given increasing event times $\{\ldots, t_{n-1}, t_n, \ldots\}$, representations in terms of time t_n and the corresponding inter-event time $\tau_n = t_n - t_{n-1}$ are isomorphic, and we use them interchangeably.

Conditional Density Estimation of Event Time. To model the event time, we estimate the conditional probability density $p^*_e(t) = p(t|s, r, o, G_{<t})$ of event time t, given an event of type r between entities s and o, and the history $G_{<t}$ of all past interactions. Note

that the star symbol $*$ as in $p_e^*(t)$ in this chapter denotes the dependency on the history $G_{<t}$.

More concretely, in order to define $p_e^*(t)$, we consider the conditional density of two types of inter-event times τ_{eo} and τ_{min}. Let $p_{\text{EO}}^{*,e}(t) = p(\tau_{\text{eo}}|s, r, o, G_{<t})$ be the conditional density of τ_{eo}, which is the time that has elapsed since entities s and o interacted with each other in their latest event. Also, let $p_{\text{MIN}}^{*,e}(t) = p(\tau_{\text{min}}|s, r, o, G_{<t})$ be the conditional density of τ_{min}, which is defined to be $\min(\tau_{(s)}, \tau_{(o)})$, where $\tau_{(s)}$ and $\tau_{(o)}$ refer to the time that has elapsed since s and o interacted with any other entity in their latest event. In other words, τ_{eo} considers how recent the two entities' interaction was, while τ_{min} considers when the most recent event happened in either entity's history. In experiments where we set $p_e^*(t)$ to be either of these two probabilities, we find $p_{\text{MIN}}^{*,e}(t)$ and $p_{\text{EO}}^{*,e}(t)$ to be most effective for predicting event time (Section 8.4.2) and temporal links (Section 8.4.3), respectively. Note that $p_e^*(t)$ can be more generally defined in terms of both conditional densities to be $p_e^*(t) = \alpha \cdot p_{\text{EO}}^{*,e}(t) + (1-\alpha) \cdot p_{\text{MIN}}^{*,e}(t)$, where $\alpha\,(0 \leq \alpha \leq 1)$ weights each term, and it can also be extended with further conditional densities to model different types of inter-event times.

Importantly, our choice to model event time directly via conditional density estimation differs from existing TPP-based approaches for modeling TKGs [TDWS17, HMW+20], where event times are modeled using the conditional intensity function $\lambda_e^*(t) = \lambda(t|s, r, o, G_{<t})$, which represents the rate of events happening, given the history. In these intensity-based approaches, computing $p_e^*(t)$ requires integrating $\lambda_e^*(t)$, and thus a major challenge lies in selecting a good parametric form for $\lambda_e^*(t)$. Simple intensity functions (e.g., constant and exponential intensity) have a closed-form log-likelihood, but they usually have limited expressiveness (e.g., they have a unimodal distribution); even if they use RNNs to capture rich temporal information, the resulting distribution $p_e^*(t)$ still has limited flexibility. More sophisticated ones using neural networks can better capture complex distributions, but their log-likelihood and expectation cannot be obtained in closed form, requiring Monte Carlo approximation. Mixture distributions, on the other hand, are an expressive model for conditional density estimation, with the potential to approximate any density, and with closed-form likelihood and expectation.

Specifically, we use a mixture of log-normal distributions since inter-event times are positive. Log-normal mixture distributions are defined in terms of mixture weights $\boldsymbol{w}$, means $\boldsymbol{\mu}$, and standard deviations $\boldsymbol{\sigma}$. An important consideration in employing a log-normal mixture is that the timing of an event in a TKG is affected by what has happened before (i.e., $G_{<t}$) and what comprises the event triple $e = (s, r, o)$. In light of this, we obtain the three groups of mixture parameters $\boldsymbol{w}_e^*$, $\boldsymbol{\mu}_e^*$, and $\boldsymbol{\sigma}_e^* \in \mathbb{R}^K$, where the symbols e and $*$ signify these parameters' dependency on the event triple e and the history $G_{<t}$, and K denotes the number of mixture components.

To obtain mixture parameters, we learn entity and relation embeddings such that they reflect their temporal status (which we describe in the next paragraph), as they are influenced by events that occurred over time. Let $\mathbf{t}_s^*$, $\mathbf{t}_o^*$, and $\mathbf{t}_r^*$ denote such temporal embeddings of subject s, object o, and relation r, respectively, after processing events prior

Table 8.2: Table of symbols.

Symbol	Definition
(s, r, o, t)	directed edge from subject s to object o, with edge type (relation) r and timestamp t
e	event triple (s, r, o)
τ	inter-event time (i.e., $\tau_n = t_n - t_{n-1}$)
τ_{eo}	elapsed time since entities s and o last interacted with each other
τ_{min}	elapsed time since entities s and o last interacted with any other entity
$*$	symbol that signifies that an associated symbol (e.g., $p^*(\tau)$ and $\mathbf{t}_i^*$) depends on the past events
$p_e^*(\tau)$	conditional probability density function $p(\tau \mid s, r, o, G_{<t})$
$\boldsymbol{w}, \boldsymbol{\mu}, \boldsymbol{\sigma}$	weights, means, and standard deviations of a log-normal mixture
$\mathbf{t}_i, \mathbf{s}_i, \mathbf{t}_i^*, \mathbf{s}_i^*$	temporal (structural) embeddings of entity i, with $*$ reflecting its state after processing events until time t
$\mathbf{t}_r, \mathbf{s}_r, \mathbf{t}_r^*, \mathbf{s}_r^*$	temporal (structural) embeddings of relation r, with $*$ reflecting its state after processing events until time t
$\mathbf{t}_i^{(\ell,t)}, \mathbf{s}_i^{(\ell,t)}$	temporal (structural) embeddings of entity i learned by ℓ-th GNN layer at time t
$\mathbf{t}_i^{(*,t)}, \mathbf{s}_i^{(*,t)}$	temporal (structural) embeddings of entity i updated after events until time t are processed

to time t. We model the conditional dependence of $p(\tau \mid s, r, o, G_{<t})$ on $e = (s, r, o)$ and $G_{<t}$ by concatenating the embeddings of s, r, and o into a context vector $\boldsymbol{c}_e^* = [\mathbf{t}_s^* \| \mathbf{t}_r^* \| \mathbf{t}_o^*]$, and transforming it into the parameters of the log-normal mixture representing $p(\tau \mid s, r, o, G_{<t})$ using a multilayer perceptron (MLP) as follows:

$$\boldsymbol{w}_e^* = \text{softmax}(\text{MLP}(\boldsymbol{c}_e^*)), \boldsymbol{\mu}_e^* = \text{MLP}(\boldsymbol{c}_e^*), \boldsymbol{\sigma}_e^* = \exp(\text{MLP}(\boldsymbol{c}_e^*)) \tag{8.3}$$

where softmax ensures that mixture weights sum to 1, and $\exp$ makes standard deviations positive. With these parameters, EvoKG defines $p(\tau \mid s, r, o, G_{<t})$ to be

$$\begin{aligned} p(\tau \mid s, r, o, G_{<t}) &= p(\tau \mid \boldsymbol{w}_e^*, \boldsymbol{\mu}_e^*, \boldsymbol{\sigma}_e^*) \\ &= \sum_{k=1}^{K} \frac{(w_e^*)_k}{\tau (\sigma_e^*)_k \sqrt{2\pi}} \exp\left(-\frac{(\log \tau - (\mu_e^*)_k)^2}{2 (\sigma_e^*)_k{}^2} \right), \end{aligned} \tag{8.4}$$

which is a valid probability density function as it is nonnegative and integrates to one for $\tau \in \mathbb{R}_+$.

Time-Evolving Temporal Representations. Informative context for estimating inter-event time can be constructed by summarizing different types of interactions each entity had with others into temporal entity embeddings. Further, how much time elapsed since the latest event gives useful information for learning such temporal representations. To this end, we utilize the neighborhood aggregation framework of relation-aware graph neural networks (GNNs). Specifically, we extend R-GCN [SKB+18] such that the aggregation can take inter-event time $\tau_{i,j}$ between entities i and j into account.

Given concurrent events G_t, we summarize entity i's interaction with others in G_t as follows:

$$\mathbf{t}_i^{(\ell+1,t)} = \sigma\Bigg(\sum_{r\in\mathcal{R}}\sum_{j\in\mathcal{N}_t^{(i,r)}}\frac{1}{\nu_{i,j}}\cdot\mathbf{W}_r^{\ell}\mathbf{t}_j^{(\ell,t)} + \mathbf{W}_0^{\ell}\mathbf{t}_i^{(\ell,t)}\Bigg) \tag{8.5}$$

where $\mathbf{t}_i^{(\ell,t)}$ denotes the temporal embeddings of entity i learned by ℓ-th layer of the extended R-GCN by aggregating events in G_t; $\nu_{i,j}$ is a factor to consider the inter-event time, which we define to be $\nu_{i,j} = \log\tau_{i,j}$; $\mathcal{R}$ is the set of relations; $\mathcal{N}_t^{(i,r)}$ is entity i's concurrent neighbors at time t, connected via an edge of type r; $\mathbf{W}_r^{\ell}$ and $\mathbf{W}_0^{\ell}$ are the weight matrices in the ℓ-th layer for relation r and self-loop, respectively. Then with L layers in total, $\mathbf{t}_i^{L}$ summarizes entity i's temporal interactions in the L-hop neighborhood. The initial temporal embeddings $\mathbf{t}_i^{(0,t)}$ are set to the static representation $\mathbf{t}_i$ that EvoKG learns to capture the temporal characteristics of entities, i.e., $\mathbf{t}_i^{(0,t)} = \mathbf{t}_i$ for any time t.

To model the dynamics of temporal updates, the context for modeling inter-event time should reflect the changes made by new events. Given $\mathbf{t}_i^{(L,t)}$ which summarizes the temporal interaction patterns from concurrent events at time t, EvoKG learns time-evolving dynamics from the evolution of $\mathbf{t}_i^{(L,t)}$ over time, by using recurrent neural networks RNN_{te} for temporal entity representation learning:

$$\mathbf{t}_i^{(*,t)} = \text{RNN}_{\text{te}}\big(\mathbf{t}_i^{(L,t)}, \mathbf{t}_i^{(*,t-1)}\big) \tag{8.6}$$

where $\mathbf{t}_i^{(L,t)}$ is the input to RNNs at each time; $\mathbf{t}_i^{(*,0)}$ is zero-initialized; and $\mathbf{t}_i^{(*,t)}$ is the temporal embedding of entity i updated after events until time t are processed. In this framework, as aggregating incoming and outgoing neighbors captures sending and receiving patterns between entities, EvoKG aggregates neighborhood in both directions to learn embeddings that reflect different interaction patterns, which are then processed by RNN_{te} to be used in the context $\boldsymbol{c}_e^*$.

Next, EvoKG considers the concurrent events G_t^r that have relation r, and takes the average of the temporal embeddings of the entities in G_t^r to provide it as the context $\mathbf{t}_r^t$ to RNN_{tr}, which learns the temporal embedding $\mathbf{t}_r^{(*,t)}$ of relation r at time t:

$$\mathbf{t}_r^{(*,t)} = \text{RNN}_{\text{tr}}\big(\mathbf{t}_r^t, \mathbf{t}_r^{(*,t-1)}\big). \tag{8.7}$$

For brevity, we use the notation $\mathbf{t}_i^* = \mathbf{t}_i^{(*,t)}$ and $\mathbf{t}_r^* = \mathbf{t}_r^{(*,t)}$.

8.3.2 Modeling Evolving Network Structure

As new events occur, TKGs evolve structurally and the dynamics between entities also change over time. For instance, companies that did not work together may start to collaborate at some point to work on the same project, and this change may influence the communication patterns between them and related entities in the TKG. We capture this

intricate structural dynamics by modeling the conditional probability $p(s, r, o|G_{<t})$ of an event triple (s, r, o).

Conditional Density Estimation of Event Triple. To model $p(s, r, o|G_{<t})$, we learn the embeddings of entities and relations (which we discuss in the next paragraph), which capture their time-evolving structural dynamics. For flexibility, we learn these embeddings separately from temporal embeddings discussed in Section 8.3.1. Let $\mathbf{s}_i$ and $\mathbf{s}_r$ denote the static structural embeddings of entity i and relation r, and let $\mathbf{s}_i^*$ and $\mathbf{s}_r^*$ be the structural embeddings of entity i and relation r obtained by processing events until time t. We concatenate static and dynamic embeddings and denote them using $\bar{\mathbf{s}}_i^* = [\mathbf{s}_i^* \| \mathbf{s}_i]$ and $\bar{\mathbf{s}}_r^* = [\mathbf{s}_r^* \| \mathbf{s}_r]$. Then EvoKG summarizes the past events $G_{<t}$, which $p(s, r, o|G_{<t})$ is conditioned on, by the graph-level representation $\bar{\mathbf{g}}^*$, which EvoKG obtains via an element-wise max pooling over the structural embeddings of all entities, i.e.,

$$\bar{\mathbf{g}}^* = \max(\{\bar{\mathbf{s}}_i^* \mid i \in \text{entities}(G_{<t})\}). \tag{8.8}$$

Based on these representations, we decompose $p(s, r, o|G_{<t})$ to be

$$p(s, r, o|G_{<t}) = p(o|s, r, G_{<t}) \cdot p(r|s, G_{<t}) \cdot p(s|G_{<t}) \tag{8.9}$$

and parameterize each term separately, as follows:

$$p(o|s, r, G_{<t}) = \text{softmax}\left(\text{MLP}\left([\bar{\mathbf{s}}_s^* \| \bar{\mathbf{s}}_r^* \| \bar{\mathbf{g}}^*]\right)\right), \tag{8.10}$$

$$p(r|s, G_{<t}) = \text{softmax}\left(\text{MLP}\left([\bar{\mathbf{s}}_s^* \| \bar{\mathbf{g}}^*]\right)\right), \tag{8.11}$$

$$p(s|G_{<t}) = \text{softmax}\left(\text{MLP}\left([\bar{\mathbf{g}}^*]\right)\right). \tag{8.12}$$

Time-Evolving Structural Representations. An effective modeling of $p(s, r, o|G_{<t})$ based on the above parameterization depends on learning informative context that reflects how structural dynamics between entities have changed over time. As with learning temporal embeddings, neighborhood aggregation of GNNs and recurrent event modeling using RNNs provide an effective framework to capture this complex structural evolution. Thus, we adapt the framework used for event time modeling in Section 8.3.1 for learning time-evolving structural embeddings.

Let $\mathbf{s}_i^{(\ell,t)}$ denote the structural embeddings of entity i learned by ℓ-th R-GCN layer by aggregating concurrent events G_t. As before, we set the initial structural embeddings $\mathbf{s}_i^{(0,t)}$ to $\mathbf{s}_i$ for each time t. Given embeddings $\mathbf{s}_i^{(\ell,t)}$ for all entities, $\mathbf{s}_i^{(\ell+1,t)}$ is learned using Equation (8.5), where $\mathbf{t}_i^{(\ell,t)}$ is replaced by $\mathbf{s}_i^{(\ell,t)}$, and $\nu_{i,j}$ is set to the neighborhood size $|\mathcal{N}_t^{(i,r)}|$. The structural relation embedding $\mathbf{s}_r^t$ at time t is constructed using concurrent events in the same way as in the temporal case. Then EvoKG learns the time-evolving structural embeddings $\mathbf{s}_i^{(*,t)}$ and $\mathbf{s}_r^{(*,t)}$ using RNN_{se} and RNN_{sr} as follows.

$$\mathbf{s}_i^{(*,t)} = \text{RNN}_{\text{se}}\left(\mathbf{s}_i^{(L,t)}, \mathbf{s}_i^{(*,t-1)}\right), \;\; \mathbf{s}_r^{(*,t)} = \text{RNN}_{\text{sr}}\left(\mathbf{s}_r^t, \mathbf{s}_r^{(*,t-1)}\right) \tag{8.13}$$

For brevity, we use the notation $\mathbf{s}_i^* = \mathbf{s}_i^{(*,t)}$ and $\mathbf{s}_r^* = \mathbf{s}_r^{(*,t)}$.

8.3.3 Parameter Learning

Loss Function. Let $\mathcal{L}_{\text{iet}}$ and $\mathcal{L}_{\text{triple}}$ denote the negative log-likelihood (NLL) of the inter-event time and an event triple, respectively. Based on our problem formulation and modeling choices, the two NLLs of a quadruple $q=(s,r,o,t)$ (i.e., a timestamped event in a TKG) are obtained as follows.

$$\mathcal{L}_{\text{iet}}(q) = -\log p(t|s,r,o,G_{<t}) = -\log p(\tau|\boldsymbol{w}_e^*, \boldsymbol{\mu}_e^*, \boldsymbol{\sigma}_e^*) \tag{8.14}$$

$$\begin{aligned}\mathcal{L}_{\text{triple}}(q) &= -\log p(s,r,o|G_{<t}) \\ &= -\log p(o|s,r,G_{<t}) - \log p(r|s,G_{<t}) - \log p(s|G_{<t})\end{aligned} \tag{8.15}$$

We optimize EvoKG by minimizing the loss $\mathcal{L}$ containing both NLLs for all events in the training set:

$$\mathcal{L} = \sum_t \sum_{q=(s,r,o,t)\in G_t} \lambda_1 \mathcal{L}_{\text{iet}}(q) + \lambda_2 \mathcal{L}_{\text{triple}}(q) \tag{8.16}$$

where λ_1 and λ_2 control the importance of each loss term.

Learning Algorithm. Since there exist intricate relational and temporal dependencies among events in TKGs, it is not optimal to decompose events into independent sequences for an efficient training, as we lose relational information. At the same time, since a TKG may cover a long period of time, keeping track of the entire history for each entity can incur prohibitively high computation and memory cost, especially when learning graph-contextualized representations for entities and relations. To address these challenges, we organize events by their timestamps and process concurrent events in parallel, while truncating backpropagation every b time steps (Algorithm 8.1). As experimental results show, this enables an accurate and efficient parameter learning, which outperforms the best baseline in terms of both prediction accuracy and efficiency.

8.4 Experiments

In experiments, we answer the following research questions.

- **[RQ1]** How accurately can EvoKG estimate the event time?
- **[RQ2]** How accurately can EvoKG predict temporal links?
- **[RQ3]** How efficient is EvoKG in terms of training and inference?
- **[RQ4]** How do different parameter settings and event time modeling affect EvoKG's performance?

After describing the datasets (Section 8.4.1), we present results for the above research questions (Sections 8.4.2 to 8.4.5). Experimental settings are provided in Section 8.7.1.

8.4.1 Temporal Knowledge Graph Data

We use five real-world TKGs that have been widely used in previous studies: ICEWS18 [BLO+15], ICEWS14 [TDWS17], GDELT [LS13], WIKI [LC18], and YAGO [MBS15]. ICEWS (Integrated Crisis Early Warning System) and GDELT (Global Database of Events, Language,

Algorithm 8.1: Parameter Learning

Input: TKG G with training data, TKG G' with validation data, maximum number of epochs max_epochs, number L of R-GCN layers, patience p, number of time steps b for truncated backpropagation.

$epoch \leftarrow 1$
repeat
  foreach $t \in Timestamps(G)$ **do**
    if $t > 0$ **then**
      Compute the loss $\mathcal{L}_t$ for concurrent events in G_t based on Equation (8.16)
      Optimize model parameters and truncate backpropagation every b time steps
    foreach $i \in Entities\,(G_t)$ **do** /* *executed in parallel* */
      Compute $\mathbf{t}_i^{(L,t)}$ and $\mathbf{t}_i^*$ using eqs. (8.5) and (8.6)
      Compute $\mathbf{s}_i^{(L,t)}$ and $\mathbf{s}_i^*$ using a modified version of eq. (8.5) and eq. (8.13)
    foreach $r \in Rels\,(G_t)$ **do** /* *executed in parallel* */
      Compute $\mathbf{t}_r^*$ and $\mathbf{s}_r^*$ using eqs. (8.7) and (8.13)
  Evaluate the validation performance for events in G'
  $epoch \leftarrow epoch + 1$
until $epoch = max\_epochs$ or no improvement in validation performance for p consecutive times;

Table 8.3: Statistics of real-world TKGs. Time interval denotes the minimum duration between two temporally adjacent events.

Dataset	# Train Edges	# Valid Edges	# Test Edges	# Entities	# Relations	Time Interval
ICEWS18	373,018	45,995	49,545	23,033	256	24 hours
ICEWS14	275,367	48,528	341,409	12,498	260	24 hours
ICEWS-500	184,725	32,292	228,648	500	256	24 hours
GDELT	1,734,399	238,765	305,241	7,691	240	15 minutes
WIKI	539,286	67,538	63,110	12,554	24	1 year
YAGO	161,540	19,523	20,026	10,623	10	1 year

and Tone) are event-based TKGs; WIKI and YAGO are knowledge bases with temporally associated facts. Statistics of these TKGs are presented in Table 8.3. We order these datasets by timestamps, and split each one into training, validation, and test sets, as shown in Table 8.3. We also use ICEWS-500 [TDWS17] for experiments on event time prediction, which is a TKG constructed from ICEWS data, containing a smaller number of nodes than ICEWS18, since some previous studies reported results only on ICEWS-500 without releasing code.

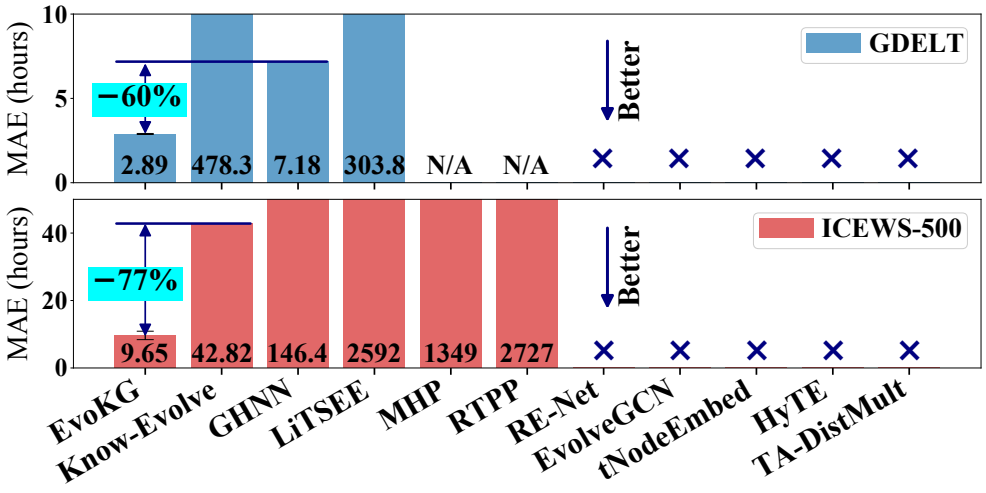


Figure 8.3: **EvoKG is accurate.** EvoKG achieves the best event time prediction results, with up to 77% less MAE than the second best method; all improvements are statistically significant with p-value < 0.05. Note that many methods for TKGs (marked by $\times$) cannot predict event time. N/A denotes results are unavailable.

8.4.2 Event Time Prediction (RQ1)

Task Description. Given an event triple $e = (s, r, o)$ and the history $G_{<t}$, the goal is to predict when the event e will happen. Specifically, the time of an event triple e is estimated to be the expected value of the time that event e occurs, given the history. Thanks to the use of a mixture distribution, in EvoKG, this expectation is obtained in a closed form by

$$\mathbb{E}_{\tau \sim p_e^*(\tau)}(\tau) = \sum\nolimits_k (w_e^*)_k \exp\left((\mu_e^*)_k + {((s_e^*)_k)^2}/{2}\right). \tag{8.17}$$

On the other hand, other approaches, such as GHNN [HMW+20], need to approximate the integral to compute the expected value by using Monte Carlo, as they do not have a close-form solution. We report MAE (mean absolute error), which is the average of the absolute difference between the predicted and true time in hours. Lower MAE indicates higher prediction accuracy.

Baselines. We compare EvoKG against three existing methods for modeling TKGs with the ability to predict event time: Know-Evolve [TDWS17], GHNN [HMW+20], and LiTSEE [XNA+19]. Know-Evolve and GHNN model event time based on temporal point process (TPP) framework. While there exist several other methods for modeling TKGs, they are unable to forecast event time, thus they cannot be used for this evaluation. We also report the result of two other baselines used in [TDWS17], MHP (Multi-dimensional Hawkes Process) and RTPP (Recurrent Temporal Point Process). MHP models dyadic entity interactions as multi-dimensional Hawkes process: an entity pair constitutes an event, and MHP learns when each event occurs, without taking relations (event types) into account. RTPP is a simplified version of RMTPP [DDT+16], which estimates the

conditional intensity function of an event by using a global RNN. Relations are also considered in RTPP.

Results. Figure 8.3 reports the event time prediction accuracy on ICEWS-500 and GDELT. Results of Know-Evolve, RTPP, and MHP are obtained from [TDWS17] (except that we obtained Know-Evolve's result on GDELT using the reference implementation), and those of LiTSEE and GHNN are taken from [HMW+20]. Notably, most TKG methods in Figure 8.3 are marked with $\times$ due to their inability to estimate event time. Also, the results of RTPP and MHP are not available (marked with 'N/A') as their implementation is not publicly available. In EvoKG, we used τ_{min} to model the inter-event time. In experiments, methods are updated using the observed graph snapshot at each time step to make future predictions.

Results show that EvoKG consistently outperforms all existing approaches, with up to 77% less MAE than the second-best method. We conduct one-sample t-tests and verify that all improvements over baselines are statistically significant with p-value < 0.05. Graph-based methods (EvoKG, Know-Evolve, and GHNN), which learn the temporal patterns of events by utilizing information from the neighborhood, perform much better than simpler baselines (RTPP and MHP), which model event time based only on direct interactions between entities. Also, TPP-based Know-Evolve and GHNN outperform LiTSEE, a non-TPP approach which incorporates time information by adding a temporal component into entity embeddings. EvoKG achieves the best event time prediction results by modeling event time using mixture distributions, which are much more flexible and expressive than those used by existing methods.

8.4.3 Temporal Link Prediction (RQ2)

Task Description. Given a test quadruple $q = (s, r, o, t)$ and the history $G_{<t}$, we create a perturbed quadruple $q' = (s, r, o', t)$ by replacing o with every other entity o' in the graph, and compute the score of q'. We then sort all perturbed quadruples in descending order of the score and report the rank of the ground truth quadruple q. We report MRR (mean reciprocal rank), which is the average of the reciprocal of the ground truth q's rank, and Hits@{3,10}, which is the percentage of correct entities in the top 3 and 10 predictions. For both metrics, higher values indicate better link prediction results.

Baselines. We compare EvoKG against the following baselines for both static and temporal KG reasoning. (1) DistMult [YYH+15], R-GCN [SKB+18], ConvE [DMSR18], and RotatE [SDNT19] are methods for static KG reasoning. They are applied to a static, cumulative graph constructed from events in the training data, where edge timestamps are ignored. (2) TA-DistMult [GDN18] and HyTE [DRT18] are methods for temporal KG reasoning in an interpolation setting. (3) dyngraph2vecAE [GCC20], tNodeEmbed [SGR19], EvolveGCN [PDC+20], and GCRN [SDVB18] are methods for reasoning over homogeneous graphs in an extrapolation setting. (4) Know-Evolve [TDWS17], DyRep [TFBZ19], and RE-Net [JQJR20] are methods for temporal KG reasoning in an extrapolation setting. GHNN [HMW+20] is not included as the implementation is not available. Also, results reported in [HMW+20] were obtained after applying its own filtering criteria, where

Table 8.4: **EvoKG wins.** EvoKG outperforms existing methods in terms of temporal link prediction in most cases, achieving up to 116% higher MRR (mean reciprocal rank) on real-world TKGs. Best results are in bold, and second best results are underlined.

	Method	ICEWS14			ICEWS18			WIKI			YAGO			GDELT		
		MRR	H@3	H@10	MRR	H@3	H@10	MRR	H@3	H@10	MRR	H@3	H@10	MRR	H@3	H@10
Static	DistMult	9.72	10.09	22.53	13.86	15.22	31.26	27.96	32.45	39.51	44.05	49.70	59.94	8.61	8.27	17.04
	R-GCN	15.03	16.12	31.47	15.05	16.49	29.00	13.96	15.75	22.05	27.43	31.24	44.75	12.17	12.37	20.63
	ConvE	21.64	23.16	38.37	22.56	25.41	41.67	26.41	30.36	39.41	41.31	47.10	59.67	18.43	19.57	32.25
	RotateE	9.79	9.37	22.24	11.63	12.31	28.03	26.08	31.63	38.51	42.08	46.77	59.39	3.62	2.26	8.37
Temporal	TA-DistMult	11.29	11.60	23.71	15.62	17.09	32.21	26.44	31.36	38.97	44.98	50.64	61.11	10.34	10.44	21.63
	HyTE	7.72	7.94	20.16	7.41	7.33	16.01	25.40	29.16	37.54	14.42	39.73	46.98	6.69	7.57	19.06
	dyngraph2vecAE	6.95	8.17	12.18	1.36	1.54	1.61	2.67	2.75	3.00	0.81	0.74	0.76	4.53	1.87	1.87
	tNodeEmbed	13.36	13.13	24.31	7.21	7.64	15.75	8.86	10.11	16.36	3.82	3.88	8.07	12.97	12.61	21.22
	EvolveGCN	8.32	7.64	18.81	10.31	10.52	23.65	27.19	31.35	38.13	40.50	45.78	55.29	6.54	5.64	15.22
	Know-Evolve	0.05	0.00	0.10	0.11	0.00	0.47	0.03	0.00	0.04	0.02	0.00	0.01	0.11	0.02	0.10
	Know-Evolve+MLP	16.81	18.63	29.20	7.41	7.87	14.76	10.54	13.08	20.21	5.23	5.63	10.23	15.88	15.69	22.28
	DyRep+MLP	17.54	19.87	30.34	7.82	7.73	16.33	10.41	12.06	20.93	4.98	5.54	10.19	16.25	16.45	23.86
	R-GCRN+MLP	21.39	23.60	38.96	23.46	26.62	41.96	28.68	31.44	38.58	43.71	48.53	56.98	18.63	19.80	32.42
	RE-Net	23.91	26.63	42.70	26.81	30.58	45.92	31.55	34.45	42.26	46.37	51.95	61.59	**19.44**	**20.73**	33.81
	EvoKG	**27.18** ±0.001	**30.84** ±0.001	**47.67** ±0.001	**29.28** ±0.002	**33.94** ±0.004	**50.09** ±0.002	**68.03** ±0.031	**79.60** ±0.036	**85.91** ±0.063	**68.59** ±0.003	**81.13** ±0.005	**92.73** ±0.009	19.28 ±0.001	20.55 ±0.001	**34.44** ±0.002

GHNN achieved similar results to RE-Net.

Results. Table 8.4 provides link prediction results on five TKGs. Results of baselines are obtained from [JQJR20]. In EvoKG, we used τ_{eo} to model the inter-event time. In experiments, after making predictions at each time step, methods are updated using the observed graph snapshot. EvoKG outperforms all existing approaches across different datasets, achieving up to 116% higher MRR than the best baseline, except on GDELT, where EvoKG achieves similar performance to the best baseline. It is noteworthy that an improvement over baselines is the most significant on WIKI and YAGO, which contains much more events occurring at relatively regular intervals. By modeling event time, EvoKG can predict such temporal patterns accurately. Among baselines, static methods in the first four rows perform worse than the best temporal baseline, RE-Net, as they do not consider temporal factors. At the same time, some temporal methods, such as dyngraph2vecAE and EvolveGCN, often perform worse than static methods, even though they are designed to take temporal evolution of dynamic networks into account. This indicates that incorporating temporal factors needs to be done carefully to avoid introducing additional noise. Know-Evolve and DyRep are the two existing methods based on temporal point processes. While they can be used for temporal link prediction, they are not effective for predicting links, even after applying an MLP decoder to their embeddings, as they focus on modeling just $p(t|s, r, o, G_{<t})$, and thus do not explicitly learn the evolving network structure by modeling $p(s, r, o|G_{<t})$ as in EvoKG. By modeling event time and network structure simultaneously, EvoKG outperforms various existing methods in predicting temporal links and event times.

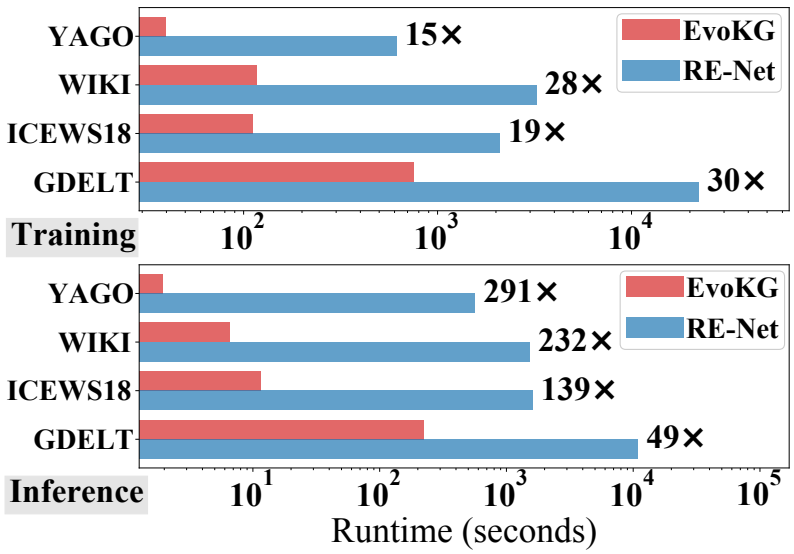


Figure 8.4: **EvoKG is fast.** EvoKG performs training (top) and inference (bottom) up to $30\times$ and $291\times$ faster than RE-Net.

8.4.4 Efficiency (RQ3)

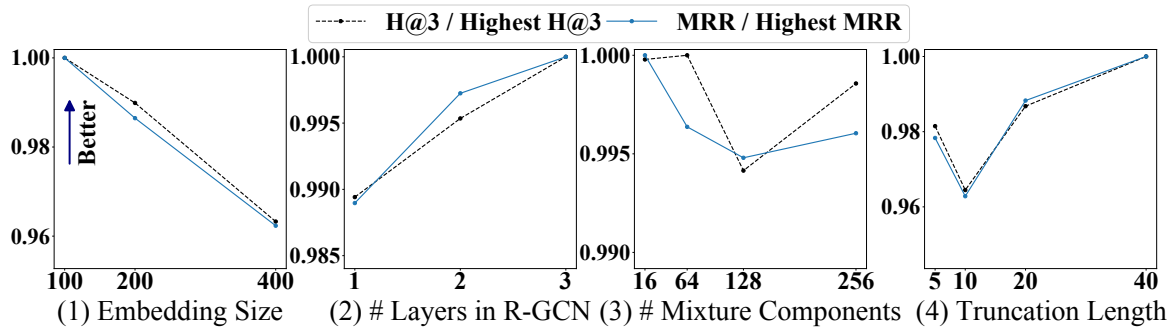
Setup. We compare EvoKG against RE-Net, the best performing baseline method for temporal link prediction, in terms of model training and inference speed. We evaluate the training speed by measuring the time taken to train one epoch, and the inference speed by measuring the time taken to evaluate the entire test data in terms of $p(s, r, o|G_{<t})$.

Results. Figure 8.4 shows the time taken for training (top) and inference (bottom) over four TKGs. The training speed for EvoKG is $23\times$ on average, and up to $30\times$, faster than RE-Net. In making inferences, EvoKG is $177\times$ on average, and up to $291\times$, faster than RE-Net. This is because RE-Net's design for handling events results in a lot of repeated computations for neighborhood aggregation and processing event history. The difference in runtime is even more pronounced in making inferences since RE-Net processes event quadruples individually during inference. On the other hand, EvoKG processes concurrent events simultaneously, effectively reducing redundant operations. As a result, EvoKG performs both tasks much more efficiently than RE-Net.

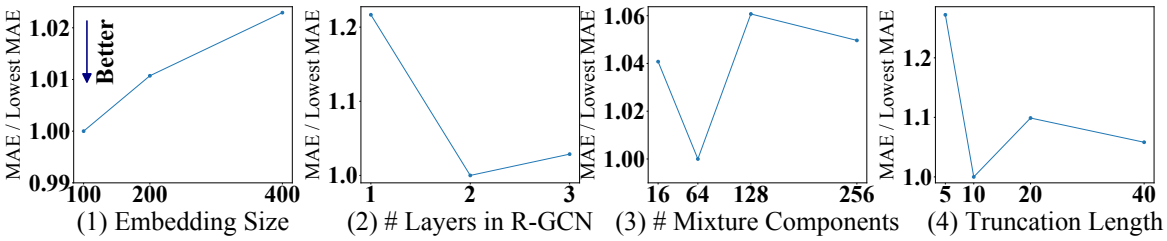
8.4.5 Ablation Study (RQ4)

8.4.5.1 Parameter Sensitivity

We evaluate how the performance of EvoKG changes, as we vary (a) the embedding size, (b) the number of R-GCN layers, (c) the number of mixture components, and (d) truncation length (the number of time steps between backpropagation truncation in RNNs). Figure 8.5 shows the link prediction result on ICEWS18 (top), and event time prediction result on ICEWS-500 (bottom); reported values denote the ratio of the result obtained with the parameter setting on the x-axis to the best result.



(a) Temporal link prediction performance of EvoKG on ICEWS18.



(b) Event time prediction performance of EvoKG on ICEWS-500.

Figure 8.5: Link prediction and event time prediction performance as we vary (1) embedding size, (2) number of R-GCN layers, (3) number of mixture components, and (4) truncation length.

Embedding Size. We set the embedding size (both temporal and structural embeddings) to 100, 200, and 400. As Figures 8.5a and 8.5b show, the best accuracy on the two datasets is achieved by an embedding size of 100, while using a much larger embedding size of 400 hurts the performance, as this leads to overfitting.

Number of R-GCN Layers. EvoKG extends R-GCNs for learning temporal and structural representations. The number of R-GCN layers determines the size of the neighborhood from which a node aggregates information. For predicting both temporal link and event time, using two layers leads to a better result than using a single layer, indicating that an increased neighborhood brings useful information for modeling a TKG. However, using more layers can incur over-smoothing issues, decreasing time prediction accuracy.

Number of Mixture Components. EvoKG uses a mixture distribution to model the event time. The number of mixture components affects the flexibility of the mixture distribution. Figures 8.5a and 8.5b report the performance of EvoKG as the number of mixtures is set to 16, 64, 128, and 256. EvoKG achieves the best link and event time prediction results, with 16 and 64 mixture components, respectively. While using a larger number of mixture components decreases performance, EvoKG still achieves high accuracy, and is not very sensitive to these parameter settings.

Truncation Length. For efficient and scalable training, EvoKG truncates backpropagation

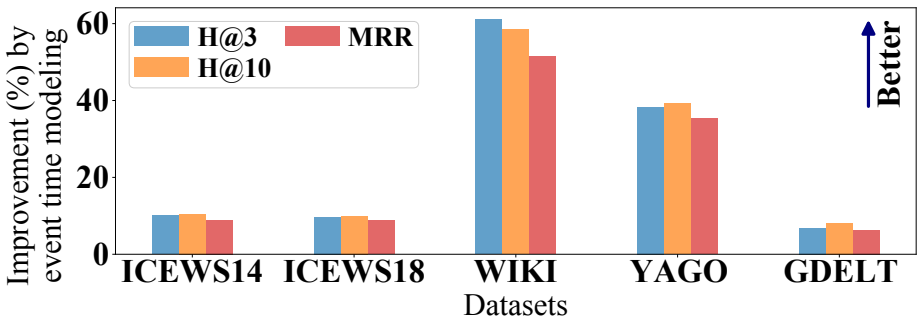


Figure 8.6: Modeling event time improves temporal link prediction accuracy on all TKGs, by up to 61%.

every b time steps (Algorithm 8.1). We set b to 5, 10, 20, and 40, and measure the performance. On the two datasets, the best result is achieved with $b = 40$ (link prediction) and $b = 10$ (event time prediction), and using a smaller time steps tends to decrease the accuracy, as this restricts the model's ability to keep track of the history. Results also show that if the truncation length is longer than appropriate, it may hurt the predictive accuracy.

8.4.5.2 Effects of Event Time Modeling

To evaluate the importance of modeling event time in the overall quality of TKG modeling, we report in Figure 8.6 the improvement made by event time modeling in terms of link prediction accuracy on all TKGs. Specifically, let $\text{Acc}_{1,2}$ and Acc_2 be the link prediction performance obtained with both terms and only the second term in Equation (8.2), respectively. The improvement in Figure 8.6 is defined to be $((\text{Acc}_{1,2} - \text{Acc}_2)/\text{Acc}_2) \times 100$. Results show that modeling event time consistently improves the prediction accuracy on all datasets, by up to 61%.

8.5 Related Work

In this section, we review previous works on reasoning over graphs.

Reasoning over Static Graphs. Inspired by the success of the Skip-gram model [MSC+13] in NLP, several methods [GL16, PAS14, TQW+15] learn node embeddings that maximize the likelihood of preserving neighborhoods of nodes in a network via random walks. More recently, many graph neural networks (GNNs) have been developed for representation learning in homogeneous graphs for semi-supervised and self-supervised settings, including GCN [KW17] and GAT [VCC+18].

To learn the representations of entities and relations in heterogeneous KGs, tensor factorization (TF) [KB09] has been widely used. There exist several types of TF methods [PJLK16, POK19, OPJ+19, JJSK16, GPP20], such as CP and Tucker decomposition, which make different assumptions on the underlying data generating process. Yet, most TF methods are not well suited for temporal data (e.g., they do not take inter arrival

times into account). In recent years, various relational learning techniques have been proposed for heterogeneous KGs, using different scoring functions to evaluate the triples in KGs, including models with distance-based scoring functions (e.g., TransE [BUG+13], RotatE [SDNT19]) and models based on semantic matching (e.g., RESCAL [NTK11], DistMult [YYH+15], NTN [SCMN13], ConvE [DMSR18]). GNNs have also been extended for relation-aware representation learning on KGs, such as R-GCN [SKB+18] and HAN [WJS+19]. Overall, these methods are developed for static graphs and lack the ability to model temporally evolving dynamics.

Reasoning over Dynamic Homogeneous Graphs. To capture temporal dynamics in time-evolving graphs, RNNs have been used to summarize and maintain evolving entity states in many methods [SDVB18, RCF+20, PDC+20, KZL19, SGR19]. Often, GNNs have been combined with RNNs to capture both structural and temporal dependencies [SDVB18, RCF+20, PDC+20]. Another line of work [XRK+20, SWG+20] employed graph attention mechanisms to make the model aware of the temporal order and the time span between entities when computing the attention weights. Some other approaches applied deep autoencoders to dynamic graph snapshots [GKHL18, GCC20], enforced temporal smoothness on entity embeddings [ZGY+16, ZYR+18], and performed temporal random walks [NLR+18]. As these methods are designed for single-relational dynamic graphs, they lack mechanisms to capture the multi-relational nature of TKGs, which we focus on in this work.

Reasoning over Dynamic Heterogeneous Graphs. Static KG embedding methods have been extended to take temporal information into account, including TA-DistMult [GDN18], TTransE [LC18], HyTE [DRT18], and diachronic embedding [GKBP20]. These temporal KG embedding techniques address an interpolation problem where the goal is to infer missing facts at some point in the past, and cannot predict future events. Recently, several methods have been developed to tackle the extrapolation problem setting, where the goal is to predict new facts at future time steps. TensorCast [dARF17] uses exponential smoothing to forecast latent entity representations, obtained with TF. RE-Net [JQJR20] learns dynamic entity embeddings by summarizing concurrent events in an autoregressive architecture; yet, it has no components to model the event time. Know-Evolve [TDWS17], DyRep [TFBZ19], and GHNN [HMW+20] model the occurrences of events over time by using temporal point processes (e.g., Rayleigh and Hawkes processes) that estimate the conditional intensity function. Inspired by [SBG20], EvoKG models the event time by directly estimating its conditional density in a flexible and efficient framework.

In summary, most existing methods for both homogeneous and heterogeneous dynamic graphs model just the second term on the evolving network structure in Equation (8.2), and thus cannot predict when events will occur. On the other hand, a few methods like [TDWS17, TFBZ19] that model the first term on the evolving temporal patterns in Equation (8.2) do not model the other term, which greatly limits their reasoning capacity. In this work, we present a problem formulation that unifies these two major tasks (Section 8.2), and develop an effective framework EvoKG that tackles them simultaneously.

8.6 Conclusion

Temporal knowledge graphs (TKGs) represent facts about entities and their relations, which occurred at a specific time, or are valid for a specific duration of time. Reasoning over TKGs, i.e., inferring new facts from TKGs, is crucial to many applications, including question answering and recommender systems. Towards an effective reasoning over TKGs, this chapter makes the following contributions.

- **Problem Formulation**. We present a problem formulation that unifies the two core problems for TKG reasoning—modeling the timing of events and the evolving network structure.
- **Framework**. We develop EvoKG, an effective framework for modeling TKGs that jointly addresses the two core problems.
- **Effectiveness & Efficiency**. Experiments show that EvoKG outperforms existing methods in terms of effectiveness (link and time prediction accuracy improved by up to *116%*) and efficiency (training speed improved by up to *30*$\times$ over the best baseline).

Reproducibility. The code and data are available at https://namyongpark.github.io/evokg.

8.7 Appendix

8.7.1 Experimental Settings

Data Split. We split datasets into training, validation, and test sets in chronological order, as shown in Table 8.3. For training EvoKG, we applied early stopping, checking the validation performance with a patience of five. Then the model with the best validation performance was used for testing.

Hyperparameters. We used a two-layer R-GCN [SKB+18] with block diagonal decomposition (BDD), which reduces the number of parameters and alleviates overfitting, and set the size of entity and relation embeddings in EvoKG to 200 (except for WIKI, where it was set to 192 to meet the constraint of using R-GCN with BDD). Static entity embeddings were initialized using the Glorot initialization, while the initial dynamic embeddings were zero-initialized. We used a single-layer Elman RNN with tanh nonlinearity, but different RNNs, such as GRU, can easily be used. We trained the model using the AdamW optimizer with a learning rate of 0.001, a weight decay of 0.00001, $\beta_1 = 0.9$, and $\beta_2 = 0.999$, and applied dropout with $p = 0.2$. As training the module for modeling network structure usually takes longer than training the module for modeling event time, we first trained the model with $\lambda_1 = 0$ and $\lambda_2 = 1$, and then trained the entire model with $\lambda_1 = \lambda_2 = 1$ until convergence. We truncated the backpropagation for RNNs every 40 time steps for GDELT, and every 20 time steps for other datasets. We set the number K of mixture components to 128.

For the details of baselines used in this chapter, please refer to [TDWS17] for Know-Evolve, RTPP, and MHP; [HMW+20] for GHNN and LiTSEE; and [JQJR20] for other baselines including RE-Net.

Compute Resources. We ran experiments on a Linux machine with 8 CPUs (Intel(R) Xeon(R) CPU E5-2623 v4 @ 2.60GHz), 30GB RAM, and an NVIDIA Quadro P6000 GPU.

Software. To implement EvoKG and the evaluation pipeline, we used the following software (software version is specified in the parentheses): python (3.8.3), Deep Graph Library (0.53), PyTorch (1.7.1), NumPy (1.18.5), and pandas (1.0.5). We used PyTorch's RNN implementation, and Deep Graph Library's R-GCN implementation.

Chapter based on work published in WWW 2022 [PRK+22].

Chapter 9

Contrastive Graph Clustering for Community Detection and Tracking

Given entities and their interactions in the web data, which may have occurred at different time, how can we effectively find communities of entities and track their evolution in an unsupervised manner? In this chapter, we approach this important task from graph clustering perspective. Recently, state-of-the-art clustering performance in various domains has been achieved by deep clustering methods. Especially, deep graph clustering (DGC) methods have successfully extended deep clustering to graph-structured data by learning node representations and cluster assignments in a joint optimization framework. Despite some differences in modeling choices (*e.g.*, encoder architectures), existing DGC methods are mainly based on autoencoders, minimizing reconstruction loss, and use the same clustering objective with relatively minor adaptations. Also, while many real-world graphs are dynamic in nature, previous studies have designed DGC methods only for static graphs. In this work, we develop CGC, a novel end-to-end framework for graph clustering, which fundamentally differs from existing methods. CGC learns node embeddings and cluster assignments in a contrastive graph learning framework, where positive and negative samples are carefully selected in a multi-level scheme such that they reflect the hierarchical community structures and network homophily. Also, we extend CGC for time-evolving data, where temporal graph clustering is performed in an incremental learning fashion, with the ability to detect change points. Extensive evaluation on static and temporal real-world graphs demonstrates that the proposed CGC consistently outperforms existing methods.

Table 9.1: **CGC wins on features.** Comparison of the proposed CGC with deep learning approaches for graph clustering. [A]: Aware of/Utilizing. CL: Clustering, RP: Representation.

Methods / *Desiderata*	AE [HS06]	GAE [KW16]	DAERNN [GCC20]	DAEGC [WPH+19]	SDCN [BWS+20]	AGCN [PLJH21]	**CGC (Ours)**
Jointly optimizing CL and RP				✓	✓	✓	✓
[A] Input node features	✓	✓		✓	✓	✓	✓
[A] Network homophily		✓	✓	✓	✓	✓	✓
[A] Hierarchical communities							✓
Temporal graph clustering			✓				✓
Learning Objective							
Contrastive learning-based							■
Reconstruction-based	■	■	■	■	■	■	

9.1 Introduction

Given events between two entities, how can we effectively find communities of entities in an unsupervised manner? Also, when the events are associated with time, how can we detect communities and track their evolution? Various web platforms, including social networks, generate data that represent events between entities, occurring at a certain time, *e.g.*, check-in records and user interaction logs. Finding communities from such dyadic temporal events can be formulated as a graph clustering problem, in which the goal is to find node clusters from a graph, where the two entities of an event are nodes, and the event forms a temporal edge between them.

In recent years, state-of-the-art clustering performance has been achieved by deep clustering methods in several application domains [XGF16, GGLY17, YFSH17, YZZ+17, YLY+19, MSFK18, LDZ19]. Following this success, deep graph clustering (DGC) [WPH+19, BWS+20, PHF+20, PLJH21, TGC+14] has been receiving increasing attention recently, which aims to learn cluster-friendly representations using deep neural networks for graph clustering. Early DGC methods [TGC+14, KW16] have taken a two-stage approach, where representation learning and clustering are done in isolation; *e.g.*, node embeddings are learned by graph autoencoders (GAEs) [KW16], to which a clustering method is applied. More accurate clustering results have been obtained by another group of DGC methods [WPH+19, BWS+20, PLJH21] that adopt a joint optimization framework, where a clustering objective is combined with the representation learning objective, and both are optimized simultaneously in an end-to-end manner.

In DGC methods, a major challenge lies in how to effectively utilize node features and graph structure. Graph neural networks provide an effective framework to this end, which propagate and aggregate node features over the graph, thus learning node embeddings that reflect network homophily. Further, to make the most of graph structure and node features, existing methods tried different modeling choices, *e.g.*, in terms of encoder architectures (GAEs, attentional GAEs, GAEs with autoencoders (AEs)) and

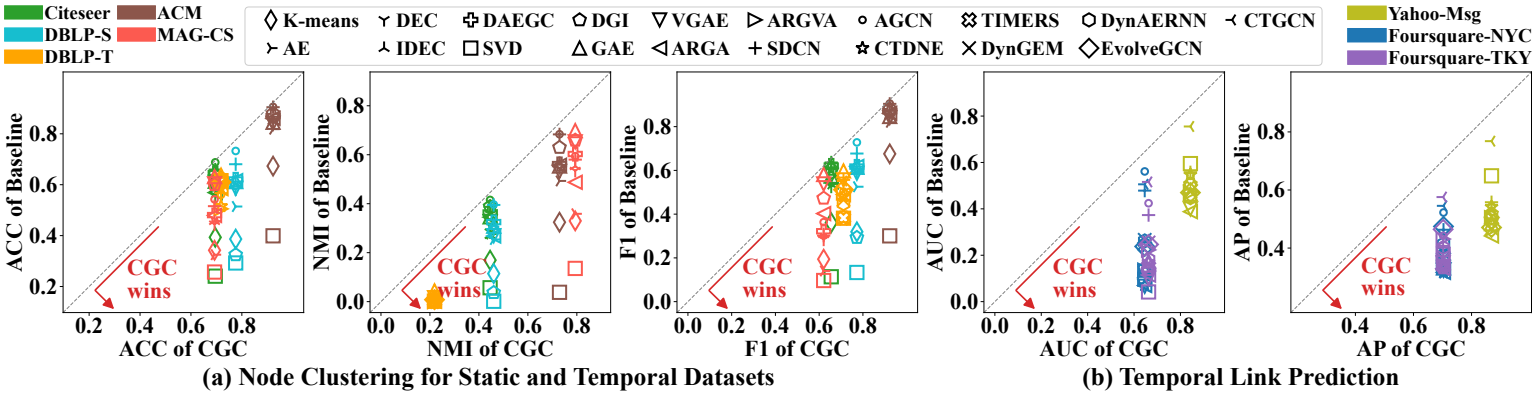


Figure 9.1: **CGC outperforms competition:** All points are below the diagonals for all baselines and graphs. CGC achieves more accurate (a) node clustering on static and temporal data, and (b) link prediction based on the time-evolving cluster membership.

how graph structural features and node attributes are combined. Still, differences among them are relative small: They mainly (1) perform reconstruction loss minimization for unsupervised representation learning (reconstructing the adjacency matrix, node attribute matrix, or both) in an AE-based framework, and (2) employ the clustering objective first proposed in DEC [XGF16], which optimizes cluster assignments by learning from the model's high confidence predictions.

In addition, while many real-world networks are dynamic in nature, no DGC methods are designed for clustering time-evolving graphs to our knowledge. Although we can apply existing methods to cluster temporal graphs (*e.g.*, by ignoring time and applying them to the cumulative graph anew at each time step), practical solutions for temporal graph clustering should be able to model the changing community structures and detect major change points in the network, which cannot be addressed effectively by a straightforward application of existing methods.

In this chapter, we develop CGC, a new graph clustering framework based on contrastive learning, which significantly differs from existing DGC methods as shown in Table 9.1. The main idea of contrastive learning [vdOLV18, CKNH20, KTW+20] is to pull an entity (called an anchor) and its positive sample closer to each other in the embedding space, while pushing the anchor away from its negative sample. When no labels are available, the choice of positive and negative samples plays a crucial role in contrastive learning. In such cases, positive samples are often obtained by taking different views of the data (*e.g.*, via data augmentations such as rotation and color distortion for images [CKNH20]), while negative samples are randomly selected from the entire pool of samples. In CGC, based on our understanding of real-world networks and their characteristics (*e.g.*, homophily and hierarchical community structures), we design a multi-level scheme to choose positive and negative samples such that they reflect the underlying hierarchical communities and their semantics. Also, from information theoretic perspective, our contrastive learning objective is designed to maximize the mutual information between

an entity and the hierarchical communities it belongs to in the latent space. Then guided by this multi-level contrastive objective, cluster memberships and entity embeddings are iteratively optimized in an end-to-end framework.

Furthermore, to find communities from time-evolving data, we extend CGC framework to the temporal graph clustering setting. Upon the arrival of new events, entity representations and cluster memberships are updated to reflect the new information, and at the same time, temporal smoothness assumption is incorporated into the GNN encoder, and also into the contrastive learning objective, which enables CGC to adapt to changing community structures in a controlled manner. We also show how CGC can be applied to detect major changes occurring in the network, and thereby adaptively choose homogeneous historical events to find communities from.

In summary, the key contributions of this work are as follows.

- **Novel Framework.** We propose CGC, a new contrastive graph clustering framework. As discussed above and summarized in Table 9.1, CGC is a significant departure from previous DGC methods.
- **Temporal Graph Clustering.** We extend our CGC framework for temporal data. CGC is the first deep graph clustering method for clustering time-evolving networks.
- **Effectiveness.** We demonstrate the effectiveness of CGC via extensive evaluation of clustering quality on several real-world datasets. Figure 9.1 shows that CGC consistently outperforms various existing methods on both static and temporal datasets.

9.2 Problem Formulation

In this section, we introduce notations and definitions, and present the problem formulation. Table 9.2 lists the symbols used in this work.

9.2.1 Graph Clustering

Let $G = (V, E)$ be a graph with nodes $V = \{1, \ldots, n\}$ and edges $E = \{(u_i, v_i) \mid u_i, v_i \in V\}_{i=1}^{m}$. Let $\mathbf{F} \in \mathbb{R}^{n \times d}$ be an input node feature matrix. Let k denote the number of node clusters. We define cluster membership as follows to represent node-to-cluster assignment.

Definition 1. Cluster Membership:
A cluster membership $\boldsymbol{\phi}_u \in \mathbb{R}^{k}_{\geq 0}$ of node u is a stochastic vector that adds up to one, where the i-th entry is the probability of node u belonging to i-th cluster.

According to Definition 1, a node belongs to at least one cluster, and can belong to multiple clusters. Note that this soft cluster membership includes hard cluster assignments as a special case, in which one node belongs to exactly one cluster. Based on this definition, graph clustering problem is formally defined as follows.

Problem 1. Graph Clustering:
Given a graph $G = (V, E)$ and input node features $\mathbf{F} \in \mathbb{R}^{n \times d}$, learn a cluster membership matrix $\boldsymbol{\Phi} \in \mathbb{R}^{n \times k}_{\geq 0}$ for all n nodes in G.

After graph clustering, we want the nodes to be grouped such that nodes are more similar to those in the same cluster (*e.g.*, in terms of external node labels if available, or connectivity patterns, node features, and structural roles) than nodes in different clusters.

9.2.2 Temporal Graph Clustering

Let $G_\tau = (V, E_\tau)$ be a temporal graph snapshot with nodes $V = \{1, \ldots, n\}$ and temporal edges $E_\tau = \{(u, v, t) \mid u, v \in V,\ t \in \tau\}$, where t is time (*e.g.*, a timestamp at the level of milliseconds), and τ denotes some time span (*e.g.*, one minute, one hour).

Definition 2. Temporal Graph Stream:
A temporal graph stream $\mathcal{G}$ is a sequence of graph snapshots $\mathcal{G} = \{G_{\tau_i}\}_{i=1}^{T}$ where T is the number of graph snapshots thus far in the stream. Graph snapshots $\{G_{\tau_i}\}$ are assumed to be non-overlapping and ordered in increasing order of time.

Problem 2. Temporal Graph Clustering:
Given a temporal graph stream $\mathcal{G} = \{G_{\tau_i}\}_{i=1}^{T}$ and input node features $\mathbf{F} \in \mathbb{R}^{n \times d}$, learn a cluster membership matrix $\mathbf{\Phi}_i \in \mathbb{R}_{\geq 0}^{n \times k}$ for each time span τ_i.

9.3 Preliminaries

Mutual Information (MI) and Contrastive Learning. The MI between two random variables (RVs) measures the amount of information obtained about one RV by observing the other RV. Formally, the MI between two RVs X and Y, denoted $I(X;Y)$, is defined as

$$I(X;Y) = \mathbb{E}_{p(x,y)}\left[\log\left(p(x,y)/p(x)p(y)\right)\right] \tag{9.1}$$

where $p(x, y)$ is the joint density of X and Y, and $p(x)$ and $p(y)$ denote the marginal densities of X and Y, respectively. Several recent studies [VFH+19, vdOLV18, CKNH20, BBR+18, HFL+19] have seen successful results in representation learning by maximizing the MI between a learned representation and different aspects of the data.

Since it is difficult to directly estimate MI [POvdO+19], MI maximization is normally done by deriving a lower bound on MI and maximizing it instead. Intuitively, several lower bounds on MI are based on the idea that RVs X and Y have a high MI if samples drawn from their joint density $p(x, y)$ and those drawn from the product of marginals $p(x)p(y)$ can be distinguished accurately. InfoNCE [vdOLV18] is one such lower bound of MI in the form of a noise contrastive estimator [GH10]:

$$I(X;Y) \geq \mathbb{E}\left[\frac{1}{K}\sum_{i=1}^{K}\log\frac{\exp(f(x_i, y_i))}{\frac{1}{K}\sum_{j=1}^{K}\exp(f(x_i, y_j))}\right] \triangleq I_{\text{NCE}}(X;Y) \tag{9.2}$$

where the expectation is over K independent samples $\{x_i, y_i\}_{i=1}^{K}$ from the joint density $p(x, y)$. Given a set of K independent samples, the critic function $f(\cdot)$ aims to

predict for each x_i which one of the K samples x_i was drawn together with, *i.e.*, by assigning a large score to the positive pair (x_i, y_i), and small scores to other negative pairs $\{(x_i, y_j)\}_{j \neq i}^{K}$.

Graph Neural Networks (GNNs). GNNs are a class of deep learning architectures for graphs that produce node embeddings by repeatedly aggregating local node neighborhoods. In general, a GNN encoder $\mathcal{E}$ maps a graph G and input node features $\mathbf{F} \in \mathbb{R}^{n \times d}$ into node embeddings $\mathbf{H} \in \mathbb{R}^{n \times d'}$, that is, $\mathcal{E}(G, \mathbf{F}) = \mathbf{H}$.

9.4 Proposed Framework

In this section, we present the CGC framework. We describe how CGC performs graph clustering in a multi-level contrastive learning framework (Section 9.4.1), and discuss how we extend CGC for temporal graph clustering and address its challenges (Section 9.4.2).

9.4.1 CGC: Contrastive Graph Clustering

The proposed framework CGC performs contrastive graph clustering by carrying out the following two steps in an alternating fashion: (1) refining cluster memberships based on the current node embeddings, and (2) optimizing node embeddings such that nodes from the same cluster are closer to each other, while those from different clusters are pushed further away from each other.

9.4.1.1 Multi-Level Contrastive Learning Objective

In CGC, contrastive learning happens in the second step above, where positive samples of a node are assumed to have been generated by the same cluster as the node of interest, whereas negative samples are assumed to belong to different clusters. While no cluster membership labels are available, there exist several signals at different levels of the input data that we can utilize to effectively construct positive and negative samples for contrastive graph clustering, namely, input node features and the characteristics of real-world networks, such as network homophily and hierarchical community structure.

Signal: Input Node Features. Entities in the same community tend to have similar attributes. Thus informative node features can be used to distinguish nodes in the same class from those in different classes. Node features are especially helpful for sparse graphs, since they can complement the scarce relational information.

Therefore, for node u, we take its input features $\mathbf{f}_u$ as its positive sample, and randomly select another node v to take its input features $\mathbf{f}_v$ as a negative sample; these positive and negative samples are then contrasted with node embedding $\mathbf{h}_u$. Let $\mathcal{S}_u^F = \{\mathbf{f}_u'^i\}_{i=0}^{r}$ be the set of one positive ($i = 0$) and r negative ($1 \leq i \leq r$) samples (*i.e.*, input features) for node u, where $'$ indicates that sampling was involved. Since input features and latent embeddings can have different dimensionality, we define a node feature-based contrastive loss $\mathcal{L}_F$ using a bilinear critic function (see Section 9.3 for more details of the

Table 9.2: Table of symbols.

Symbol	Definition
u, v	node indices
n	number of nodes
k	number of clusters
t	timestamp of an edge, $t \geq 0$
τ	time span
$G = (V, E)$	static graph with nodes V and edges E
$\boldsymbol{\phi}_u \in \mathbb{R}^k_{\geq 0}$	cluster membership vector of node u for graph G
$G_\tau = (V, E_\tau)$	temporal graph snapshot with nodes V and temporal edges for time span τ
$\mathcal{G} = \{G_{\tau_i}\}$	temporal graph stream
$\boldsymbol{\Phi}_i \in \mathbb{R}^{n \times k}_{\geq 0}$	cluster membership matrix for time span τ_i
$\mathbf{F} \in \mathbb{R}^{n \times d}$	input node feature matrix
$\mathbf{H} \in \mathbb{R}^{n \times d'}$	node embedding matrix
$\mathcal{N}(u)$ ($\mathcal{N}_\Delta(u)$)	neighbors of node u (participating in triangles with u)
$\mathcal{K} = \{k_\ell\}_{\ell=1}^L$	number of clusters for the contrastive learning

critic function) parameterized by $\mathbf{W}_F \in \mathbb{R}^{d' \times d}$:

$$\mathcal{L}_F = \sum_{u=1}^{n} -\log \frac{\exp((\mathbf{h}_u^\intercal \mathbf{W}_F \mathbf{f}_u'^0)/\tau)}{\sum_{v=0}^{r} \exp((\mathbf{h}_u^\intercal \mathbf{W}_F \mathbf{f}_u'^v)/\tau)} \tag{9.3}$$

where $\tau > 0$ is a temperature hyper-parameter.

Signal: Network Homophily. In real-world graphs, similar nodes are more likely to attach to each other than dissimilar ones, and accordingly, a node is more likely to belong to the same cluster as its neighbors than randomly chosen nodes. In particular, many real-world networks demonstrate the phenomenon of higher-order label homogeneity, *i.e.*, the tendency of nodes participating in higher-order structures (*e.g.*, triangles) to share the same label, which is a stronger signal than being connected by an edge alone. Thus, we use edges and triangles in constructing positive samples. Further, CGC encodes nodes using GNNs, whose neighborhood aggregation scheme also enforces an inductive bias for network homophily that neighboring nodes have similar representations.

Let $\mathcal{N}(u)$ denote the neighbors of node u. Let $\mathcal{N}_\Delta(u)$ be node u's neighbors that participate in the same triangle as node u; thus, $\mathcal{N}_\Delta(u) \subseteq \mathcal{N}(u)$. A positive sample for node u is then chosen from among $\mathcal{N}(u)$, with a probability of $\delta/|\mathcal{N}_\Delta(u)|$ for the neighbor in $\mathcal{N}_\Delta(u)$, and a probability of $(1-\delta)/|\mathcal{N}(u) \setminus \mathcal{N}_\Delta(u)|$ for its other neighbors, where $\delta \geq 0$ determines the weight for nodes in $\mathcal{N}_\Delta(u)$. Then the positive sample's embeddings are taken from $\mathbf{H} = \mathcal{E}(G, \mathbf{F})$.

To construct negative samples, we design a network corruption function $\mathcal{C}(G, \mathbf{F})$, which constructs a negative network from the original graph G and input node features $\mathbf{F}$.

Specifically, we define $\mathcal{C}(\cdot)$ to return corrupted node features $\widetilde{\mathbf{F}}$, via row-wise shuffling of $\mathbf{F}$, while preserving the graph G, *i.e.*, $\mathcal{C}(G, \mathbf{F}) = (G, \widetilde{\mathbf{F}})$, which can be considered as randomly relocating nodes over the graph while maintaining the graph structure. Then negative node embeddings $\widetilde{\mathbf{H}} \in \mathbb{R}^{n \times d'}$ are obtained by applying the GNN encoder to G and $\widetilde{\mathbf{F}}$, and r negative samples and their embeddings are randomly chosen.

Let $\mathcal{S}_u^H = \{\mathbf{h}_u'^i\}_{i=0}^r$ be the set containing the embeddings of one positive ($i = 0$) and r negative ($1 \leq i \leq r$) samples for node u. In CGC, a homophily-based contrastive loss $\mathcal{L}_H$ is defined as:

$$\mathcal{L}_H = \sum_{u=1}^{n} -\log \frac{\exp(\mathbf{h}_u \cdot \mathbf{h}_u'^0/\tau)}{\sum_{v=0}^{r} \exp(\mathbf{h}_u \cdot \mathbf{h}_u'^v/\tau)} \tag{9.4}$$

where we use an inner product critic function with a temperature hyper-parameter $\tau > 0$, and $'$ denoting that sampling was involved.

Signal: Hierarchical Community Structure. The above loss terms contrast an entity with other individual entities and their input features, thereby learning community structure at a relatively low level. Here, we consider communities at a higher level than before by directly contrasting entities with communities.

CGC represents communities as a cluster centroid vector $\mathbf{c} \in \mathbb{R}^{d'}$ in the same latent space as entities, so that the distance between an entity and cluster centroids reflects the entity's degree of participation in different communities. To effectively optimize an entity embedding by contrasting it with communities, cluster centroids need to have been embedded such that they reflect the underlying community structures and the semantics of input node features. While the model's initial embeddings of entities and clusters may not capture such community and semantic structures well, the above two objectives and the use of GNN encoders in CGC effectively guide the optimization process towards identifying meaningful cluster centroids, especially in the early stage of model training.

Importantly, real-world networks have been shown to exhibit hierarchical community structures. To model this phenomenon, we design CGC to group nodes into a varying number of clusters. For example, when we aim to group nodes into three clusters, we may also group the same set of nodes into ten and thirty clusters; then all clustering results taken together reveal hierarchical community structures in different levels of granularities.

Let $\mathcal{K} = \{k_\ell\}_{\ell=1}^L$ be the set of the number of clusters, and $\mathbf{C}_\ell \in \mathbb{R}^{k_\ell \times d'}$ be the cluster centroid matrix for each ℓ. Given the current node embeddings $\mathbf{H}$ and cluster centroids $\{\mathbf{C}_\ell\}_{\ell=1}^L$, positive samples for node u are chosen to be the L cluster centroids that node u most strongly belongs to, while its negative samples are randomly selected from among the other $k_\ell - 1$ cluster centroids for each ℓ. Let $\mathcal{S}_{u,\ell}^C = \{\mathbf{c}_{u,\ell}'^i\}_{i=0}^{r_\ell}$ be the set with the embeddings of one positive ($i = 0$) and r_ℓ negative ($1 \leq i \leq r_\ell$) samples (*i.e.*, centroids) for node u chosen among k_ℓ centroids. Using an inner product critic, CGC defines a

Algorithm 9.1: ContrastiveGraphClustering

Input: graph G, input node features $\mathbf{F} \in \mathbb{R}^{n \times d}$, clustering algorithm Π, number of clusters $\mathcal{K} = \{k_\ell\}_{\ell=1}^{L}$

Output: cluster membership matrix $\mathbf{\Phi} \in \mathbb{R}_{\geq 0}^{n \times k_1}$, node embedding matrix $\mathbf{H} \in \mathbb{R}^{n \times d'}$, cluster centroid matrix $\mathbf{C} \in \mathbb{R}^{k_1 \times d'}$

while *not max epoch and not converged*
  $\mathbf{H} = \mathcal{E}(G, \mathbf{F})$ `/* Equation (9.7) */`
  for $\ell = 1$ ***to*** L **do**
    $\mathbf{C}_\ell, \mathbf{\Phi}_\ell = \Pi(\mathbf{H}, k_\ell)$ `/* refine clusters and cluster memberships */`
  Calculate loss $\mathcal{L}$ using $\mathbf{H}, \mathbf{F}, \{\mathbf{C}_\ell\}$ `/* Equations (9.3) to (9.6) */`
  Backpropagate and optimize model parameters
$\mathbf{H} = \mathcal{E}(G, \mathbf{F})$
$\mathbf{C}, \mathbf{\Phi} = \Pi(\mathbf{H}, k_1)$
return $\mathbf{\Phi}, \mathbf{H}, \mathbf{C}$

hierarchical community-based contrastive loss $\mathcal{L}_C$ to be:

$$\mathcal{L}_C = \sum_{u=1}^{n} - \left(\frac{1}{L} \sum_{\ell=1}^{L} \log \frac{\exp(\mathbf{h}_u \cdot \mathbf{c}'^{0}_{u,\ell}/\tau)}{\sum_{v=0}^{r_\ell} \exp(\mathbf{h}_u \cdot \mathbf{c}'^{v}_{u,\ell}/\tau)} \right). \tag{9.5}$$

Multi-Level Contrastive Learning Objective. The above loss terms capture signals on the community structure at multiple levels, *i.e.*, individual node attributes ($\mathcal{L}_F$), neighboring nodes ($\mathcal{L}_H$), and hierarchically structured communities ($\mathcal{L}_C$). CGC jointly optimizes

$$\mathcal{L} = \lambda_F \mathcal{L}_F + \lambda_H \mathcal{L}_H + \lambda_C \mathcal{L}_C \tag{9.6}$$

where λ_F, λ_H, and λ_C are weights for the loss terms. Via multi-level noise contrastive estimation, CGC maximizes the MI between nodes and the communities they belong to in the learned latent space.

9.4.1.2 Encoder Architecture

As our node encoder $\mathcal{E}$, we use a GNN with a mean aggregator,

$$\mathbf{h}_v^l = \text{ReLU}(\mathbf{W}_G \cdot \text{MEAN}(\{\mathbf{h}_v^{l-1}\} \cup \{\mathbf{h}_u^{l-1} \mid \forall u \in \mathcal{N}(v)\})) \tag{9.7}$$

where node v's embedding $\mathbf{h}_v^l$ from the l-th layer of $\mathcal{E}$ is obtained by averaging the embeddings of node v and its neighbors from the $(l-1)$-th layer, followed by a linear transformation and the ReLU non-linearity; $\mathbf{h}_v^0$ is initialized to be the input node features $\mathbf{f}_v$.

9.4.1.3 Algorithm

Algorithm 9.1 shows how (1) cluster memberships and (2) node embeddings are alternately optimized in CGC. (1) Given the current node embeddings $\mathbf{H}$ produced by $\mathcal{E}$ (line 2), a clustering algorithm Π (*e.g.*, k-means) refines cluster centroids $\{\mathbf{C}_\ell\}$ and memberships $\{\mathbf{\Phi}_\ell\}$ (lines 3-4). (2) Based on the updated cluster centroids and memberships, CGC computes the loss and optimizes model parameters (lines 5-6). In $\{k_\ell\}$, we assume that k_1 is the number of clusters that we ultimately want to identify in the network.

9.4.2 CGC for Temporal Graph Clustering

As a new graph snapshot G_{τ_i} arrives in a temporal graph stream $\mathcal{G} = \{G_{\tau_1}, \ldots, G_{\tau_{i-1}}\}$, node embeddings $\mathbf{H}_{i-1}$ and cluster memberships $\mathbf{\Phi}_{i-1}$ that CGC learned from the snapshots until $(i-1)$-th time span are incrementally updated to reflect the new information in G_{τ_i}. Specifically, given a sequence of graph snapshots, CGC merges them into a temporal graph and performs contrastive graph clustering, taking the temporal information into account. We use the notation $G_{i:j}$ to denote a temporal graph that merges the snapshots $\{G_{\tau_i}, \ldots, G_{\tau_j}\}$, *i.e.*, $G_{i:j} = (V, E_{i:j})$ where $E_{i:j} = \bigcup_{o=i}^{j} E_{\tau_o}$. Below we describe how we extend CGC for temporal graph clustering.

9.4.2.1 Temporal Contrastive Learning Objective

As entities interact with each other, their characteristics may change over time, and such temporal changes normally occur smoothly. Thus, edges of a node observed across a range of time spans provide similar and related temporal views of the node in terms of its connectivity pattern. Accordingly, given node u for time span j, we take its embedding $\mathbf{h}_{u,j-1}$ obtained in the previous, $(j-1)$-th time span as its positive sample. To obtain negative samples, we use the same network corruption function used in Section 9.4.1.1, obtaining corrupted node features $\widetilde{\mathbf{F}}$, and take node u's embedding from the corrupted node embeddings $\mathcal{E}(G_{i:j-1}, \widetilde{\mathbf{F}})$ as the negative sample; multiple negative samples can be obtained by using multiple sets of corrupted node features. Let $\mathcal{S}^T_{u,j} = \{\mathbf{h}'^i_{u,j-1}\}_{i=0}^r$ be the set with the embeddings of one positive ($i = 0$) and r negative ($1 \leq i \leq r$) samples of node u for the j-th time span, again $'$ denoting the involvement of sampling. CGC defines a time-based contrastive loss $\mathcal{L}_T$ for time span j to be:

$$\mathcal{L}_T = \sum_{u=1}^{n} -\log \frac{\exp(\mathbf{h}_{u,j} \cdot \mathbf{h}'^0_{u,j-1}/\tau)}{\sum_{v=0}^{r} \exp(\mathbf{h}_{u,j} \cdot \mathbf{h}'^v_{u,j-1}/\tau)} \tag{9.8}$$

Note that Equation (9.8) is combined with the objectives discussed in Section 9.4.1.1 with a weight of λ_T, augmenting the loss $\mathcal{L}$ to be

$$\mathcal{L} = \lambda_F \mathcal{L}_F + \lambda_H \mathcal{L}_H + \lambda_C \mathcal{L}_C + \lambda_T \mathcal{L}_T. \tag{9.9}$$

9.4.2.2 Encoder Architecture

We extend the GNN encoder such that when it aggregates the neighborhood of a node, more weight is given to the neighbors that interacted with the node more recently. To

this end, we adjust the weight of a neighbor based on the elapsed time since its latest interaction. Let $t_{(u,v)}$ denote the timestamp of an edge between nodes u and v, and let $t_v^{\max} = \max_{u \in \mathcal{N}(v)}\{t_{(u,v)}\}$, *i.e.*, the most recent timestamp when node v interacted with its neighbors. With ψ denoting a time decay factor between 0 and 1, we apply time decay to the embedding $\mathbf{h}_u$ of neighbor u as follows:

$$\mathsf{td}(\mathbf{h}_u) = \psi^{t_v^{\max} - t_{(u,v)}}\, \mathbf{h}_u. \tag{9.10}$$

Then for time-aware neighborhood aggregation, $\mathbf{h}_u$ in Equation (9.7) is replaced with its time decayed version $\mathsf{td}(\mathbf{h}_u)$.

9.4.2.3 Graph Stream Segmentation

Given a new graph snapshot, CGC merges it with the previous ones, and refines cluster memberships on the resulting temporal graph. This process is based on the assumption that new events are similar to earlier ones. However, the new snapshot may differ greatly from the previous ones, when significant changes have occurred in the network. Detecting such changes is important, as it lets CGC find clusters from snapshots with similar patterns, and such events also correspond to important milestones or anomalies in the network.

Let $\mathcal{G}_{\text{seg}} = \{G_{\tau_i}, \ldots, G_{\tau_j}\}$ be the current graph stream segment for some i and j $(i < j)$. Given a new snapshot $G_{\tau_{j+1}}$, we expand the current segment $\mathcal{G}_{\text{seg}}$ with $G_{\tau_{j+1}}$ if $G_{\tau_{j+1}}$ is similar to $\mathcal{G}_{\text{seg}}$; if not, we start a new graph stream segment consisting only of $G_{\tau_{j+1}}$. This is basically a binary decision problem on whether to segment the graph stream or not. Our idea to solve this problem is to compare the embeddings of the nodes appearing in both $\mathcal{G}_{\text{seg}}$ and $G_{\tau_{j+1}}$. Note that the GNN encoder in this step was trained with the graphs in the current segment, and no further training has been performed on the new snapshot. Since embeddings from GNNs reflect the characteristics of nodes that CGC learned from the existing segment, the embeddings of the nodes in the new graph $G_{\tau_{j+1}}$ will be similar to their embeddings in the existing segment $\mathcal{G}_{\text{seg}}$ if $G_{\tau_{j+1}}$ is similar to $\mathcal{G}_{\text{seg}}$. By the same token, a major change in the new snapshot will lead to a large difference between the embeddings of a node in $G_{\tau_{j+1}}$ and $\mathcal{G}_{\text{seg}}$. Let V^* be the nodes appearing in both $\mathcal{G}_{\text{seg}}$ and $G_{\tau_{j+1}}$. Let $\mathbf{H}_{V^*}^{\text{seg}}$, $\mathbf{H}_{V^*}^{j+1} \in \mathbb{R}^{|V^*| \times d'}$ be the two sets of embeddings of the nodes in V^*, computed for $\mathcal{G}_{\text{seg}}$ and $G_{\tau_{j+1}}$, respectively, as discussed above. Using a distance metric $d(\cdot, \cdot)$ (*e.g.*, cosine distance), we define the distance $\mathsf{Dist}(\cdot, \cdot)$ between $\mathbf{H}_{V^*}^{\text{seg}}$ and $\mathbf{H}_{V^*}^{j+1}$ to be

$$\mathsf{Dist}(\mathbf{H}_{V^*}^{\text{seg}}, \mathbf{H}_{V^*}^{t+1}) = \text{MEAN}\{d((\mathbf{H}_{V^*}^{\text{seg}})_i, (\mathbf{H}_{V^*}^{t+1})_i) \mid i \in V^*\} \tag{9.11}$$

and segment the stream if the distance is beyond a threshold (Algorithm 9.3).

9.4.2.4 Putting Things Together

CGC tracks changing cluster memberships in an incremental end-to-end framework (Algorithm 9.2). As a new graph snapshot arrives, CGC adaptively determines a sequence of graph snapshots to find clusters from, using Algorithm 9.3 (line 3), and updates clustering results and node embeddings, using Algorithm 9.1 (line 4).

Algorithm 9.2: CGC Framework

Input: graph stream $\mathcal{G}$, input node feature matrix $\mathbf{F} \in \mathbb{R}^{n \times d'}$

Output: $\{$ cluster memberships $\mathbf{\Phi}_i \in \mathbb{R}^{n \times k}$, node embeddings $\mathbf{H}_i \in \mathbb{R}^{n \times d'}$, graph stream segment $\mathcal{G}_i^{\text{seg}} \}$ for each time span i

$\mathcal{G}_0^{\text{seg}} = \{\}$

foreach $G_{\tau_i} \in \mathcal{G}$ **do**

  $\mathcal{G}_i^{\text{seg}} = \text{GraphStreamSegmentation}(G_{\tau_i}, \mathcal{G}_{i-1}^{\text{seg}}, \mathbf{F})$ `/* Algorithm 9.3 */`

  $\mathbf{\Phi}_i, \mathbf{H}_i, \mathbf{C}_i = \text{ContrastiveGraphClustering}(\mathcal{G}_i^{\text{seg}}, \mathbf{F})$ `/* Algorithm 9.1 */`

return $\{\mathbf{\Phi}_i, \mathbf{H}_i, \mathcal{G}_i^{\text{seg}}\}_i$

9.5 Experiments

The experiments are designed to answer the following questions:

- **RQ1 (Node Clustering):** Given static and temporal graphs, how accurately can the proposed CGC cluster nodes? (Section 9.5.3)
- **RQ2 (Temporal Link Prediction):** How informative is the learned cluster membership in predicting temporal links? (Section 9.5.4)
- **RQ3 (Ablation Study):** How do different variants of the proposed framework affect the clustering performance? (Section 9.5.5)

Further results are in Appendix, *e.g.*, mining case studies (Section 9.8.1).

9.5.1 Datasets

9.5.1.1 Static Datasets.

Table 9.7 presents the statistics of static datasets. These datasets have labels and input features for all nodes.

ACM is a paper network from the ACM digital library [Libne], where two papers are linked by an edge if they are written by the same author. Papers in this dataset are published in KDD, SIGMOD, SIGCOMM, and MobiCom, and belong to one of the following three classes: database, wireless communication, and data mining. Node features are the bag-of-words of the paper keywords.

DBLP-S is an author network from the DBLP computer science bibliography [DBLne], where an edge connects two authors (*i.e.*, nodes) if they have a coauthor relationship. Authors are divided into the following four areas, according to the conferences of their publications: database, data mining, machine learning, and information retrieval. Node features are the bag-of-words of their keywords.

Citeseer is a citation network from the CiteSeer digital library [Citne], where an edge represents a citation between two documents. Documents are assigned to one of the six areas: agents, AI, database, information retrieval, machine language, and human-computer interaction. Node features are the bag-of-words of the documents.

MAG-CS is a network of authors in CS from the Microsoft Academic Graph. An edge connects two authors (*i.e.*, nodes) if they co-authored a paper. Node features are keywords of the author's papers, and node labels denote most active field of study of each author.

9.5.1.2 Temporal Datasets.

Table 9.6 presents the statistics of temporal datasets. These datasets do not contain input node features, and dynamic node labels are available only for DBLP-T.

DBLP-T is an author network from DBLP [DBLne], where edges denote coauthorship from 2004 to 2018. Node labels represent the authors' research areas (computer networks and machine learning), and may change over time as authors switch their research focus.

Yahoo-Msg is a communication network among Yahoo! Messenger users [Prone], where two users are linked by an edge if a user sent a message to another user.

Foursquare-NYC and **Foursquare-TKY** are user check-in records, collected by Foursquare [Foune] between April 2012 and February 2013 from New York City and Tokyo, respectively. An edge links a user and a venue if a user checked in to the venue.

9.5.2 Baselines

Static Baselines. K-means [HW79] is a classic clustering method applied to the raw input features. AE [HS06] produces node embeddings by using autoencoders. DEC [XGF16] is a deep clustering method that optimizes node embeddings and performs clustering simultaneously. IDEC [GGLY17] extends DEC by adding a reconstruction loss.

A group of methods also take graph structures into account for node representation learning and graph clustering. SVD [GR71] applies singular value decomposition to the adjacency matrix. GAE [KW16] and VGAE [KW16] employ a graph autoencoder and a variational variant. ARGA [PHF+20] and ARGVA [PHF+20] are an adversarially regularized graph autoencoder and its variational version. DGI [VFH+19] learns node embeddings by maximizing their MI with the graph. DAEGC [WPH+19], SDCN [BWS+20], and AGCN [PLJH21] are deep graph clustering methods that jointly optimize node embeddings and graph clustering.

Temporal Baselines. CTDNE [NLR+18] learns node embeddings based on temporal random walks. TIMERS [ZCP+18] is an incremental SVD method that employs error-bounded SVD restart on dynamic networks. DynGEM [GKHL18] leverages AEs to incrementally generate node embeddings at time t by using the graph snapshot at time $t-1$. DynAERNN [GCC20] uses historical adjacency matrices to reconstruct the current one by using an encoder-decoder architecture with RNNs. EvolveGCN [PDC+20] models how the parameters of GCNs [KW17] evolve over time. CTGCN [LXY+20] is a k-core based temporal GCN.

For methods that produce only node embeddings (*e.g.*, AE, SVD, GAE, CTDNE), we

Table 9.3: CGC achieves the best node clustering results on static graphs. Best results are in bold, and second best results are underlined.

Method	DBLP-S				ACM			
	ACC	NMI	ARI	F1	ACC	NMI	ARI	F1
K-means [HW79]	38.7±0.7	11.5±0.4	7.0±0.4	31.9±0.3	67.3±0.7	32.4±0.5	30.6±0.7	67.6±0.7
AE [HS06]	51.4±0.4	25.4±0.2	12.2±0.4	52.5±0.4	81.8±0.1	49.3±0.2	54.6±0.2	82.0±0.1
DEC [XGF16]	58.2±0.6	29.5±0.3	23.9±0.4	59.4±0.5	84.3±0.8	54.5±1.5	60.6±1.9	84.5±0.7
IDEC [GGLY17]	60.3±0.6	31.2±0.5	25.4±0.6	61.3±0.6	85.1±0.5	56.6±1.2	62.2±1.5	85.1±0.5
SVD [GR71]	29.3±0.4	0.1±0.0	0.0±0.1	13.3±2.2	39.9±5.8	3.8±4.3	3.1±4.2	30.1±8.2
DGI [VFH+19]	32.5±2.4	3.7±1.8	1.7±0.9	29.3±3.3	88.0±1.1	63.0±1.9	67.7±2.5	88.0±1.0
GAE [KW16]	61.2±1.2	30.8±0.9	22.0±1.4	61.4±2.2	84.5±1.4	55.4±1.9	59.5±3.1	84.7±1.3
VGAE [KW16]	58.6±0.1	26.9±0.1	17.9±0.1	58.7±0.1	84.1±0.2	53.2±0.5	57.7±0.7	84.2±0.2
ARGA [PHF+20]	61.6±1.0	26.8±1.0	22.7±0.3	61.8±0.9	86.1±1.2	55.7±1.4	62.9±2.1	86.1±1.2
DAEGC [WPH+19]	62.1±0.5	32.5±0.5	21.0±0.5	61.8±0.7	86.9±2.8	56.2±4.2	59.4±3.9	87.1±2.8
SDCN [BWS+20]	68.1±1.8	39.5±1.3	39.2±2.0	67.7±1.5	90.5±0.2	68.3±0.3	73.9±0.4	90.4±0.2
AGCN [PLJH21]	73.3±0.4	39.7±0.4	42.5±0.3	72.8±0.6	90.6±0.2	68.4±0.5	74.2±0.4	90.6±0.2
CGC (Ours)	**77.6±0.5**	**46.1±0.6**	**49.7±1.1**	**77.2±0.4**	**92.3±0.3**	**72.9±0.7**	**78.4±0.6**	**92.3±0.3**

Method	Citeseer				MAG-CS			
	ACC	NMI	ARI	F1	ACC	NMI	ARI	F1
K-means [HW79]	39.3±3.2	16.9±3.2	13.4±3.0	36.1±3.5	34.2±2.2	33.0±1.5	4.5±1.3	19.4±0.4
AE [HS06]	57.1±0.1	27.6±0.1	29.3±0.1	53.8±0.1	32.5±1.9	35.9±2.3	12.9±1.5	14.0±1.1
DEC [XGF16]	55.9±0.2	28.3±0.3	28.1±0.4	52.6±0.2	44.4±3.4	53.5±2.8	33.6±4.0	28.4±3.1
IDEC [GGLY17]	60.5±1.4	27.2±2.4	25.7±2.7	61.6±1.4	45.7±1.8	55.3±2.6	33.5±3.4	30.8±2.3
SVD [GR71]	24.1±1.2	5.7±1.5	0.1±0.3	11.4±1.7	25.7±4.4	13.6±7.3	1.3±2.2	9.7±4.6
DGI [VFH+19]	64.1±1.3	38.8±1.2	38.1±1.9	60.4±0.9	60.0±0.6	65.9±0.4	50.3±0.9	47.3±0.4
GAE [KW16]	61.4±0.8	34.6±0.7	33.6±1.2	57.4±0.8	63.2±2.6	69.9±0.6	52.8±1.5	58.1±4.1
VGAE [KW16]	61.0±0.4	32.7±0.3	33.1±0.5	57.7±0.5	60.4±2.9	65.3±1.4	50.0±2.1	53.8±4.0
ARGA [PHF+20]	56.9±0.7	34.5±0.8	33.4±1.5	54.8±0.8	47.9±6.0	48.7±3.0	23.6±9.0	40.3±5.0
DAEGC [WPH+19]	64.5±1.4	36.4±0.9	37.8±1.2	62.2±1.3	48.1±3.8	60.3±0.8	47.4±4.2	32.2±3.2
SDCN [BWS+20]	66.0±0.3	38.7±0.3	40.2±0.4	63.6±0.2	51.6±5.5	58.0±1.9	46.9±8.1	30.2±4.3
AGCN [PLJH21]	68.8±0.2	41.5±0.3	43.8±0.3	62.4±0.2	54.2±5.2	59.4±2.1	49.2±6.5	36.3±4.4
CGC (Ours)	**69.6±0.6**	**44.6±0.6**	**46.0±0.6**	**65.5±0.7**	**69.3±4.0**	**79.3±1.2**	**64.4±3.7**	**62.1±4.5**

apply k-means to the node embeddings to obtain cluster memberships. As the temporal link prediction task in Section 9.5.4 involves dot product scores, we apply Gaussian mixture models to node embeddings to obtain soft cluster memberships. Section 9.8.3 presents experimental settings of baselines and CGC.

Table 9.4: CGC achieves the highest node clustering accuracy on the temporal DBLP-T graph. Best results are in bold, and second best results are underlined.

Method	DBLP-T			
	ACC	NMI	ARI	F1
SVD [GR71]	61.60±0.01	0.16±0.02	-0.06±0.01	38.13±0.02
SVD-latest	61.62±0.02	0.16±0.02	-0.04±0.02	38.17±0.04
DGI [VFH+19]	61.64±0.02	0.06±0.01	0.08±0.01	38.77±0.07
DGI-latest	61.66±0.02	0.06±0.02	0.03±0.02	38.44±0.06
GAE [KW16]	63.76±0.18	4.40±0.16	7.28±0.25	59.75±0.20
GAE-latest	60.17±0.04	0.72±0.02	2.47±0.05	52.36±0.11
VGAE [KW16]	60.06±0.18	1.63±0.06	3.44±0.11	55.66±0.11
VGAE-latest	60.67±0.03	0.77±0.02	2.61±0.03	51.90±0.06
ARGA [PHF+20]	58.46±0.25	0.16±0.04	0.86±0.16	48.95±0.27
ARGA-latest	60.54±0.13	0.19±0.05	0.81±0.15	45.37±0.30
SDCN [BWS+20]	56.70±0.60	2.18±0.72	2.88±0.51	55.66±0.87
SDCN-latest	51.51±0.26	0.13±0.03	0.11±0.04	50.79±0.30
AGCN [PLJH21]	56.04±0.86	0.88±0.38	1.11±0.40	50.34±1.13
AGCN-latest	54.52±0.91	0.09±0.03	0.14±0.12	48.67±0.85
CTDNE [NLR+18]	51.58±0.07	1.98±0.06	-0.99±0.03	48.19±0.27
CTDNE-latest	50.57±0.10	0.02±0.01	0.01±0.01	49.85±0.10
TIMERS [ZCP+18]	61.70±0.00	0.09±0.01	0.02±0.00	38.21±0.01
DynGEM [GKHL18]	60.73±0.12	0.27±0.04	1.26±0.12	46.52±0.22
DynAERNN [GCC20]	62.34±0.09	0.69±0.08	1.66±0.13	44.83±0.22
EvolveGCN [PDC+20]	61.02±0.00	0.79±0.00	2.64±0.00	51.16±0.02
CTGCN [LXY+20]	59.07±0.47	1.06±0.12	2.88±0.27	55.14±0.23
CGC (Ours)	**71.82±0.99**	**21.87±1.85**	**27.28±2.93**	**71.12±0.86**

9.5.3 Node Clustering Quality (RQ1)

We evaluate the clustering quality using static and temporal graphs with node labels (Citeseer, DBLP-S, ACM, MAG-CS, and DBLP-T). Given cluster assignments, the best match between clusters and node labels is obtained by the Munkres algorithm [Kuh55], and clustering performance is measured using four metrics, which range from 0 to 1 (higher values are better): ACC (Accuracy), NMI (Normalized Mutual Information), ARI (Adjusted Rand Index), and F1 score.

9.5.3.1 Static Datasets

Table 9.3 shows the results on static graphs. The proposed method CGC consistently outperforms existing methods on all datasets in four metrics. Our novel multi-level contrastive graph learning objectives enable CGC to accurately identify node clusters by effectively leveraging the characteristics of real-world networks. We summarize our observations on the results below.

(1) Deep clustering methods (DEC, IDEC) outperform AE, which performs dimensionality reduction of the input features without clustering objectives. (2) Comparing AE against GAE and ARGA, we can see that utilizing graph structures improves the clustering quality; in some cases, the performance of GAE and ARGA is even better than DEC and IDEC, although they do not have clustering objectives. (3) Deep graph clustering methods (DAEGC, SDCN, AGCN) further improve upon deep clustering methods and those that learn from input features or the graph structure without clustering objectives, which shows the benefit of combining deep clustering with graph structural information. (4) A comparison with DGI is also noteworthy, as DGI learns node embeddings via MI maximization over a graph. Despite some similarity, DGI cannot effectively identify community structures, as it maximizes the MI between nodes and the entire graph, without regard to communities therein.

9.5.3.2 Temporal Datasets

Results on the temporal graph DBLP-T are in Table 9.4, which reports the average of the clustering performance over multiple temporal snapshots. Since static baselines have no notion of graph stream segmentation, it is up to the user to decide which data to provide as input. We evaluate static baselines in two widely used settings, representative of the way existing temporal graph clustering methods operate: The default setting is to use all observed snapshots at each time step, and the other setting is to use only the latest graph snapshot (marked with "-latest" suffix).

CGC outperforms all baselines, achieving up to 13% and 397% higher ACC and NMI, respectively, than the best performing baseline. Notably, nearly all baselines do not perform well, obtaining close to zero NMI and ARI, which demonstrates the difficult of finding clusters over time-evolving networks. Especially, no input features are available for DBLP-T, which poses an additional challenge to methods that heavily rely on them. For static baselines, using all snapshots often led to similar or better results in comparison to using the last snapshot. Results also show that temporal baselines fail

to identify changing community structure. While they are designed to keep track of time-evolving node embeddings, their representation learning mechanism does not take clustering objective into account, which makes them less effective for community detection. Figure 9.6a in Section 9.8.2 shows how ACC and NMI of CGC and four select baselines change over time. While baselines' performance shows an upward trend, their improvement is not significant. On the other hand, CGC's performance improves remarkably over time, successfully identifying changing communities.

9.5.4 Temporal Link Prediction Accuracy (RQ2)

The task is to predict the graph $G_{t+1} = (V, E_{t+1})$ at time $t+1$, where E_{t+1} are the temporal positive (*i.e.*, observed) edges. We uniformly randomly sample the same amount of temporal negative edges E^-_{t+1} such that $E^-_{t+1} = \{(u, v) \mid u, v \sim \text{Uniform}(1, \dots, n) \wedge (u, v) \notin E_{t+1}\}$. Given an edge $(u, v) \in E_{t+1} \cup E^-_{t+1}$ at time $t+1$ to predict, we estimate the likelihood of such an edge existing as $A^{(t+1)}_{uv} = \boldsymbol{\phi}^T_u \boldsymbol{\phi}_v$, where $\boldsymbol{\phi}_u$ and $\boldsymbol{\phi}_v$ are cluster memberships for nodes u and v. We can use link prediction task for evaluating clustering quality, since nodes in the same cluster are more likely to form a link between them than nodes belonging to different clusters. Also, since temporal link prediction is based on the time-evolving membership vector $\boldsymbol{\phi}$, it summarizes how accurately the learned cluster memberships capture temporally-evolving community structure. Table 9.5 reports the link prediction accuracy in terms of the area under the receiver operating characteristic curve (AUC) and the average precision (AP). Both metrics range from 0 to 1, and higher values are better. As the number of test edges (*i.e.*, $E_t \cup E^-_t$) changes over time, we average the performance weighted by the size of $E_t \cup E^-_t$ over multiple snapshots. Results show that CGC consistently outperforms baselines on all datasets, achieving up to 29% higher temporal link prediction performance. The best results among baselines were mainly obtained by CTGCN, which is a temporal method that models the network evolution. Among static baselines, AGCN mostly outperforms other statc methods, and even most dynamic baselines, except CTGCN. This can be explained by the fact that these dynamic baselines are trained using cluster agnostic objectives, which again shows that incorporating the clustering objective can be helpful for detecting communities. As in Section 9.5.3.2, we report results obtained in the two settings (*i.e.*, all vs. latest) for static baselines. There is no clear winner between them. Figure 9.6b shows how the performance of CGC and four baselines changes over time.

9.5.5 Ablation Study (RQ3)

We investigate how contrastive learning objectives affects CGC. Figure 9.2 shows node clustering results where CGC was trained with different combinations of contrastive objectives; F, H, and C denote the loss terms on node features (λ_F), network homophily (λ_H), and hierarchical communities (λ_C) in Equation (9.6), respectively, and only the specified terms were included with a weight of 1. We report relative scores, *i.e.*, scores divided by the best score for each metric. Results show that the proposed contrastive objectives are complementary, *i.e.*, jointly optimizing these objectives improves the performance, *e.g.*, F to F+H on ACM and H to H+C on DBLP-S. Especially, the best

Table 9.5: CGC consistently outperforms baselines, achieving up to 29% higher temporal link prediction performance than the best baseline. Best results are in bold, and second best results are underlined.

Method	Foursquare-NYC		Foursquare-TKY		Yahoo-Msg	
	ROC AUC	Avg. Prec.	ROC AUC	Avg. Prec.	ROC AUC	Avg. Prec.
SVD [GR71]	9.68±0.3	33.28±0.2	4.18±0.0	37.84±0.0	59.51±0.5	64.88±0.4
SVD-latest	17.67±0.6	37.93±0.6	7.20±0.2	35.08±0.1	49.26±0.2	53.21±0.2
DGI [VFH+19]	14.37±0.7	33.02±0.1	13.79±1.0	33.17±0.3	50.60±0.5	51.83±0.3
DGI-latest	18.55±1.2	34.16±0.3	20.01±0.7	34.69±0.3	41.92±0.2	45.00±0.2
GAE [KW16]	13.55±1.1	33.16±0.3	17.44±0.5	35.01±0.3	46.40±0.5	48.41±0.2
GAE-latest	19.80±0.4	35.13±0.3	21.67±0.7	37.44±0.5	42.45±0.5	44.87±0.2
VGAE [KW16]	6.63±0.1	32.34±0.1	10.06±0.3	34.90±0.4	39.97±0.0	47.99±0.1
VGAE-latest	12.02±0.2	33.18±0.0	12.91±0.2	34.93±0.2	44.21±0.1	49.61±0.0
ARGA [PHF+20]	6.96±0.0	31.63±0.0	11.45±0.1	33.00±0.3	38.79±0.1	44.17±0.1
ARGA-latest	11.89±1.1	32.38±0.2	13.17±0.2	32.61±0.0	39.84±0.1	43.78±0.0
ARGVA [PHF+20]	13.56±0.4	34.95±0.2	22.30±0.4	43.11±0.3	46.99±0.1	50.44±0.1
ARGVA-latest	26.01±0.7	39.11±0.3	32.01±0.5	45.14±0.1	50.54±0.1	51.25±0.1
SDCN [BWS+20]	47.86±0.7	46.31±0.6	37.32±0.8	40.73±0.6	55.76±1.5	55.78±1.3
SDCN-latest	25.24±0.3	36.47±0.3	19.01±1.3	35.05±0.9	54.51±0.6	55.35±0.5
AGCN [PLJH21]	56.13±1.0	52.24±1.5	42.43±2.7	44.24±2.2	54.23±2.2	54.43±1.5
AGCN-latest	41.24±3.2	49.01±2.5	41.44±5.8	51.27±4.0	51.81±1.1	52.87±0.4
CTDNE [NLR+18]	7.06±0.0	31.55±0.0	16.97±0.3	33.59±0.1	54.73±0.1	54.16±0.1
CTDNE-latest	7.27±0.0	32.28±0.0	7.36±0.1	31.98±0.0	50.11±0.0	52.70±0.1
TIMERS [ZCP+18]	23.84±0.2	37.02±0.1	15.09±0.1	33.72±0.0	48.87±0.1	49.65±0.1
DynGEM [GKHL18]	26.65±0.8	36.61±0.3	25.52±2.8	36.24±0.9	47.46±0.5	46.69±0.4
DynAERNN [GCC20]	26.17±2.1	41.39±1.6	18.23±1.1	40.15±0.7	44.81±2.0	50.44±2.1
EvolveGCN [PDC+20]	23.79±1.0	47.45±0.1	24.67±0.6	46.45±0.2	47.00±0.9	47.08±0.4
CTGCN [LXY+20]	50.58±2.4	54.54±1.5	51.61±4.5	57.56±2.8	75.51±0.9	76.82±0.7
CGC (Ours)	**64.60±0.6**	**70.34±0.5**	**66.26±0.8**	**70.22±0.6**	**84.30±0.1**	**86.88±0.1**

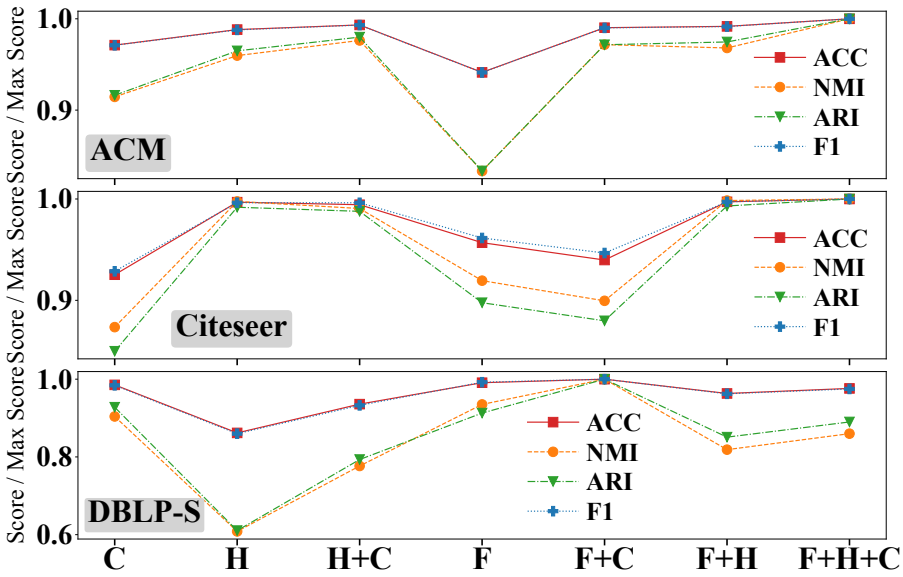


Figure 9.2: Node clustering performance of CGC, obtained with different contrastive objectives. F: Node features. H: Network homophily. C: Hierarchical Communities.

result on ACM and DBLP-S are obtained when all objectives are used together (F+H+C). However, DBLP-S shows a different pattern, where the best result was obtained with F+C. Notably, in DBLP-S, the objective on network homophily was not useful whether it is used alone (H) or with others (F vs. F+H). In DBLP-S, 36% of the nodes are isolated, making it hard to learn from graph structure. Still, joint optimization improved the results (*e.g.*, H vs. H+C).

9.6 Related Work

In this section, we review previous works on deep clustering, graph clustering, as well as temporal graph clustering.

Deep Clustering (DC). PARTY [PXF+16] is a two-stage DC method that uses autoencoders (AEs) with sparsity prior. Single-stage DC methods aim to achieve an effective clustering in an end-to-end framework. To achieve more effective clustering in an end-to-end framework, several single-stage methods have been developed. DEC [XGF16] is a single-stage AE-based method that jointly learns latent embeddings and cluster assignments by minimizing the KL divergence between the model's soft assignment and an auxiliary target distribution. IDEC [GGLY17] further improves DEC by integrating DEC's clustering loss and AE's reconstruction loss. DCN [YFSH17] adopts the K-means objective to help AEs learn K-means-friendly representations. In [JZT+17], variational AEs are used to model the data generative procedure for DC. Recently, adversarial fairness has also been incorporated for deep fair clustering [LZL20]. However, all these methods focus only on data samples, and do not consider relational information between them, which can provide valuable guidance for clustering.

Graph Clustering. Several approaches have been developed or adapted for graph clustering and community detection, including modularity-based methods [GN02], METIS [KK98], spectral methods [BS93], methods based on SVD [GR71], connected components [PPMK16, PPMK20], tensor factorization [GPP20, POK19, POK17, OPSK18] and MDL (Minimum Description Length) [ATMF12, SFPY07]. However, these methods all miss one or more of the desiderata of Table 9.1, as they mostly focus on utilizing the graph structure alone, with no support for input node features or the time evolution of graphs, and without learning node representations, which can be useful for downstream applications. Our comparison with SVD [GR71], one of the representative methods for community detection, shows the benefits of satisfying the desiderata in Table 9.1.

In this work, we focus on another group of methods for graph clustering, namely, deep graph clustering (DGC). Methods for DGC can be grouped into two categories: (1) two-stage methods that perform clustering after learning representations, and (2) single-stage methods that jointly perform clustering and representation learning (RL). Unsupervised graph RL methods are used for two-stage deep graph clustering (DGC). In [TGC+14], for instance, AEs are used to learn non-linear node embeddings, and then K-means is applied to get clustering assignments. GNN-based encoders are adopted in more recent methods. GAE [KW16] and VGAE [KW16] learn node embeddings using a graph autoencoder and a variational variant. ARGA [PHF+20] and ARGVA [PHF+20] employ an adversarially regularized graph autoencoder and its variational version. A few recent studies [VFH+19, YCWS20, WLHS21, SLZ20] investigated self-supervised learning techniques for graph RL, *e.g.*, DGI [VFH+19] optimizes GCN encoder by contrasting node embeddings with the embedding of the graph.

DMoN [TPPM20] is a single-stage method that performs clustering via spectral modularity maximization. DAEGC [WPH+19] simultaneously optimizes embedding learning and graph clustering by combining the clustering loss of DEC with the graph reconstruction loss of graph attentional AEs. SDCN [BWS+20] improves DAEGC by integrating a GCN encoder and AEs via a delivery operator. AGCN [PLJH21] further improves upon SDCN by developing two attention-based fusion modules, which aggregate features from GCNs and AEs, and multi-scale features from different layers. Despite some differences (*e.g.*, encoder architectures), existing DGC methods are mainly based on AEs, involve reconstruction loss minimization, and use the same clustering objective [XGF16] with small adjustments. The proposed CGC performs deep graph clustering in a novel contrastive graph learning framework with multi-level contrastive objectives.

Temporal Graph Clustering (TGC). Existing methods mainly perform TGC based on the graph structure and its temporal change, without considering node features and their semantics in the clustering objective. Existing TGC methods can be grouped into two classes: snapshot clustering [CSZ+07, BS06, GDC10] and consensus clustering [LF12, RB08, RB11, AG11, CM18]. Given graph snapshots, each snapshot is clustered separately in snapshot clustering, thereby ignoring inter-snapshot information. Consensus clustering instead finds a single partitioning for the entire graph snapshots. Consensus and snapshot clustering correspond to two fixed choices (*i.e.*, the entire snap-

shots vs. the last one), which is not always optimal. CGC instead adaptively determines a subset of snapshots to find clusters from.

For two-stage deep TGC, unsupervised dynamic graph representation learning methods can also be employed, which learn dynamic embeddings using temporal random walk [NLR+18], incremental SVD [ZCP+18], AEs [GKHL18, GCC20], and RNNs combined with GCNs [PDC+20, LXY+20, PLM+22]. Yet no single-stage DGC methods have been designed for TGC. This work presents the first such method for temporal network analysis.

9.7 Conclusion

This work presented CGC, a new deep graph clustering framework for community detection and tracking in the web data.

- **Novel Framework.** CGC jointly learns node embeddings and cluster memberships in a novel contrastive graph learning framework. CGC effectively finds clusters by using information along multiple dimensions, *e.g.*, node features, hierarchical communities.
- **Temporal Graph Clustering.** CGC is designed to find clusters from time-evolving graphs, improving upon existing deep graph clustering methods, which are designed for static graphs.
- **Effectiveness.** We show the effectiveness of CGC via extensive evaluation on several static and temporal real-world graphs.

9.8 Appendix

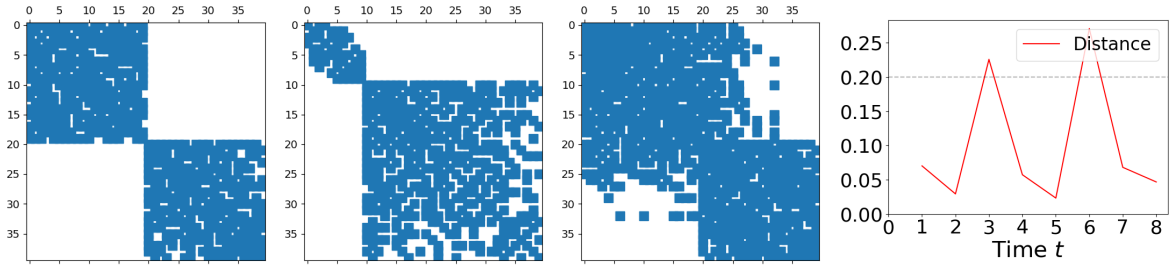
9.8.1 Mining Case Studies

9.8.1.1 Case Studies on Synthetic Graphs

In this section, we show how effectively CGC performs community detection and tracking, using synthetic graphs that consist of a small number of groups; each group corresponds to a tightly knit community, which experiences significant changes over time.

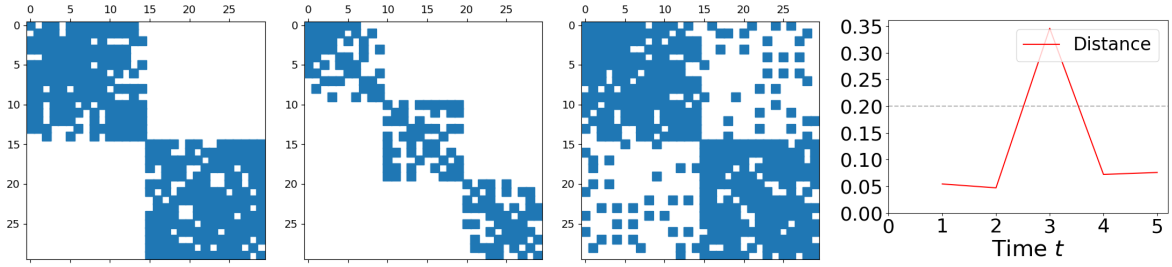
Case 1: Two Groups With Traveling Members (Figure 9.3). We have groups 1 and 2 for time 0-2. At time 3, half of the nodes in group 1 move to group 2, stay there until time 5, and then at time 6, move back to group 1, where they originally belonged. Thus there are two change points (CPs), *i.e.*, time 3 and 6 (Figure 9.3d). Figure 9.3a shows the segment prior to the first CP. Figures 9.3b and 9.3c show the segment at the first CP when the graph stream was properly segmented or not; Figure 9.3c does not clearly show the change in the size of two groups. By performing segmentation in the presence of a significant change, CGC captures a clearer community structure.

Case 2: Two Groups Reorganizing Into Three (Figure 9.4). This network initially consists of two communities, which are regrouped into three communities due to a major reorganization at time 3. Figure 9.4a shows the two communities captured by CGC



(a) Segment before the change point (time 3) (b) Segment after segmentation at the change point (c) Segment with no segmentation at change point (d) Distance between G_t and the existing segment over time

Figure 9.3: Two groups with traveling members (Case 1). Segmentation reveals a clearer community structure across time.



(a) Segment before the change point (time 3) (b) Segment after segmentation at the change point (c) Segment with no segmentation at change point (d) Distance between G_t and the existing segment over time

Figure 9.4: Two groups reorganizing into three (Case 2). CGC identifies reorganizing communities and detects the change point.

before the CP at time 3. Figure 9.4b shows that CGC successfully detects the CP (Figure 9.4d), and discovers restructured communities. Again, when the CP is ignored, it gets harder to see a clear structure of three communities from the resulting graph stream segment (Figure 9.4c).

9.8.1.2 Case Studies on Real-World Graphs

To see how the cluster membership found by CGC evolves over time, we cluster nodes based on the transition pattern (TP) of their membership vectors (Figure 9.5 (top)). Specifically, we concatenate the cluster membership vectors of each node obtained at different time steps, apply t-SNE to embed nodes in a two-dimensional space, and perform k-means clustering on the resulting two-dimensional node embeddings to obtain TP clusters. Then for each TP, we consider how cluster distribution changed over time (Figure 9.5 (bottom)). For each time step, we take the average of the membership vectors of the nodes belonging to a specific TP, and display the cluster distribution at each time as a column; clusters are associated with distinct colors, and the cluster distribution

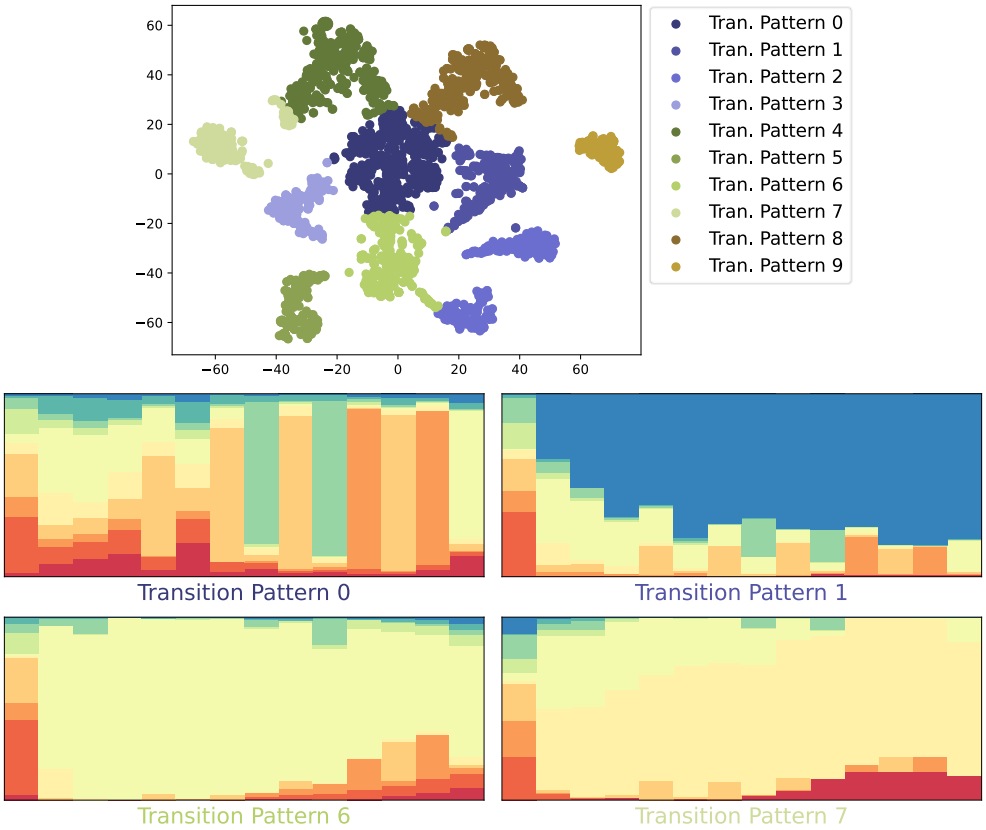


Figure 9.5: Node clusters (top) based on their transition patterns (bottom) in the Yahoo-Msg dataset.

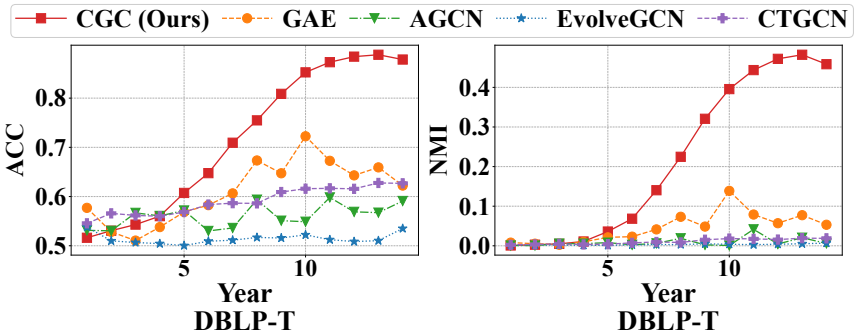
in the averaged membership vector at different time is shown by the proportion of the corresponding colors.

Yahoo-Msg (Figure 9.5). Nodes are clustered into 10 TPs. Among them, TP 0 shows a different pattern than others, where a major cluster changes frequently over time (e.g., switches between orange and green). In the scatter plot above, TP 0 is the cluster at the center, located close to a few surrounding clusters. Over time, the cluster assignments of nearby clusters have had a varying impact on how the nodes in TP 0 are clustered. Also, note that a segmentation occurred at the second time step, as can be seen in the TP plots. The color distribution of the first column in the four TPs greatly differs from those of the second and subsequent columns. Via segmentation, CGC discovers a clearer community structure.

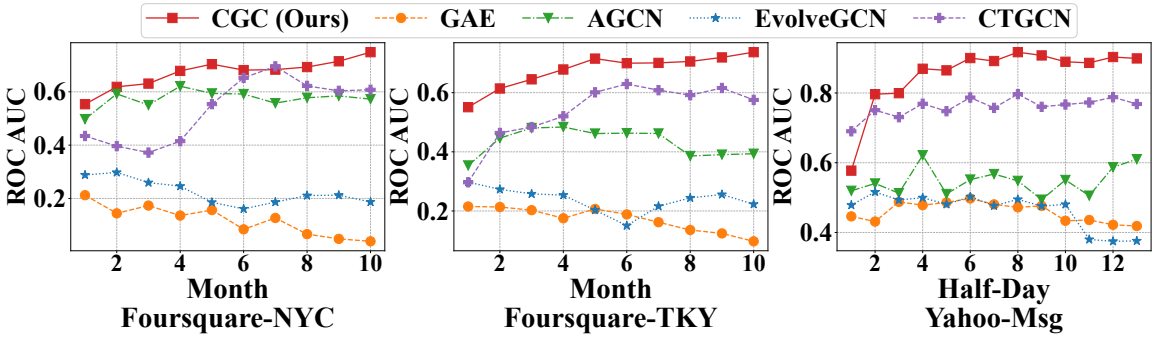
9.8.2 Clustering Performance over Time

Figure 9.6 shows how the performance of CGC and four select baselines changes over time. For static baselines, we report the results obtained by clustering all observed graph snapshots at each time step.

Node Clustering (Figure 9.6a). While all methods do not perform well for the first few



(a) Temporal Node Clustering



(b) Temporal Link Prediction

Figure 9.6: CGC achieves the best clustering performance nearly consistently on temporal graphs over the entire time period.

Table 9.6: Summary of temporal real-world datasets. N/A denotes that the corresponding datasets do not have node labels.

Dataset	Edge Type (node i, node j, time t)	# Nodes	# Edges
Yahoo-Msg	(user, user, time-second)	82,309 (82,309 users)	786,911
Foursquare-NYC	(user, venue, time-second)	39,416 (1,083 users, 38,333 venues)	454,856
Foursquare-TKY	(user, venue, time-second)	64,151 (2,293 users, 61,858 venues)	1,147,406
DBLP-T	(author, author, time-year)	6,942 (6,942 authors)	168,124

Dataset	Time Range (Inclusive)	# Time Steps	Snapshot Interval	# Dynamic Node Classes
Yahoo-Msg	0-6 (days)	14	12 hours	N/A
Foursquare-NYC	0-318 (days)	11	30 days	N/A
Foursquare-TKY	0-318 (days)	11	30 days	N/A
DBLP-T	0-13 (years)	14	1 year	2

Table 9.7: Summary of static real-world datasets used in experiments. In all datasets, nodes have labels and input features.

Dataset	Edge Type (node i, node j)	# Nodes	# Edges	# Node Classes	Feature Dimension
ACM	(paper, paper)	3,025	26,256	3	1,870
DBLP-S	(author, author)	4,057	7,056	4	334
Citeseer	(document, document)	3,327	9,104	6	3,703
MAG-CS	(author, author)	18,333	163,788	15	6,805

time steps, CGC's performance continuously improves over time, reaching an ACC of $\sim$0.89 and an NMI of $\sim$0.48 in the end. Although baselines' performance also improves with time, their improvement is much smaller than that of CGC, failing to effectively track the evolution of communities in the network.

Link Prediction (Figure 9.6b). CGC significantly outperforms baselines throughout most of the time span. Dynamic methods are not effective at capturing community structure, while deep clustering baselines like AGCN fail to track the evolution of clusters.

9.8.3 Experimental Settings

Experiments for Static Data. For ACM, DBLP-S, and Citeseer, we cite the results of all baselines (except SVD, DGI and AGCN) from [BWS+20]. For AGCN, we take its result from [PLJH21]. Settings of these baselines are given in [BWS+20, PLJH21]. We directly evaluate SVD and DGI on these datasets. On MAG-CS, we evaluate all baselines using the settings in [BWS+20, PLJH21]. For methods we evaluate, we report results averaged over 5 runs. We set node embedding size to 200 for SVD [sl21] and DGI [Lib21]. We use

a single-layer GCN in DGI as in the open source code [Lib21]. For CGC, we set node embedding size to 200, and used Adam optimizer with a weight decay of 0.0001. We set the learning rate to 0.0005 (Citeseer), 0.001 (ACM, MAG-CS), and 0.005 (DBLP-S). We used a single layer GNN in CGC. We set the temperature τ to 0.65 and δ to 0.7. Let r_F, r_H, and r_C^ℓ be the number of negative samples per positive sample for the contrastive loss $\mathcal{L}_F$, $\mathcal{L}_H$, and $\mathcal{L}_C$, where ℓ in r_C^ℓ refers to the ℓ-th level clusters. We set r_F to 180 (MAG-CS), 50 (DBLP-S), and 30 (ACM, Citeseer); r_H to 60 (MAG-CS) and 10 (others); r_C^ℓ to 60 (MAG-CS) and 30 (others) for each ℓ. Let k denote the number of clusters to find (*i.e.*, # node classes). We set $\mathcal{K} = \{k, 5k, 25k\}$. For DBLP-S, we set $\lambda_F = 4$, $\lambda_H = 0$, $\lambda_C = 1$. For ACM, Citeseer, and MAG-CS, we set $\lambda_F = 1$, $\lambda_H = 1$, $\lambda_C = 1$.

Experiments for Temporal Data. Since the temporal graphs used in experiments have no input node features $\mathbf{F}$, we used learnable node embeddings as the input node features, which were initialized by applying SVD to the row normalized adjacency matrix.

For both node clustering (Table 9.4) and link prediction (Table 9.5) evaluation, baselines used mostly the same settings. We set the size of initial node features and latent node embeddings to 128 and 32, respectively, and used the Adam optimizer with a learning rate of 0.001. Since the datasets used for temporal link prediction (Yahoo-Msg, Foursquare-NYC, Foursquare-TKY) do not have ground truth clusters, we set the size of cluster membership to 64 for all baselines and CGC. Tables 9.4 and 9.5 report results averaged over five runs.

For SVD and DGI, we used the same setting used for static graphs. For GAE, VGAE, ARGA, and ARGVA, we used the implementation of the PyTorch Geometric [Geo21] with two-layer GCN encoders. For SDCN and AGCN, we used the default settings used in [BWS+20, PLJH21], while setting the size of node embeddings to 32. For CTDNE, we used the default settings of the open source implementation [Sin21]. We set θ in TIMERS to 0.17. In DynGEM, we set α to 10^{-5}, β to 10, and both ν_1 and ν_2 to 10^{-4}. For DynAERNN, we set β to 5, the look back parameter to 3, and both ν_1 and ν_2 to 10^{-6}. In EvolveGCN, we used a two-layer GCRN; specifically, we used EvolveGCN-H, which incorporates node embeddings in RNNs. For CTGCN, we used the CTGCN-C version with the settings used in [LXY+20]. In CGC, we set $\lambda_H = 1$, $\lambda_C = \lambda_T = 0.2$, $\lambda_F = 0$; $\psi = 0.99$, $\theta = 0.3$. Let r_T be the number of negatives per positive sample for the loss $\mathcal{L}_T$. For all temporal datasets, we set $r_F = r_H = r_T = 10$. We set r_C^ℓ to 60 (link prediction datasets) and 30 (DBLP-T) for each ℓ. We set $\mathcal{K} = \{5k, 25k\}$ for DBLP-T, and $\mathcal{K} = \{k, 5k, 25k\}$ for all others. For CGC, we set the learning rate to 0.005, and the node embedding size to 32.

9.8.4 Graph Stream Segmentation

Algorithm 9.3 shows how CGC adaptively decides whether to segment the graph stream or not. A description of Algorithm 9.3 is given in Section 9.4.2.3.

Algorithm 9.3: GraphStreamSegmentation

Input: graph stream segment $\mathcal{G}_{\text{seg}}$, new graph $G_{\tau_{j+1}}$ for time span $j+1$, input node features $\mathbf{F} \in \mathbb{R}^{n \times d}$, segmentation threshold θ

Output: graph stream segment $\mathcal{G}_{\text{seg}}$

if $\mathcal{G}_{seg} \neq \varnothing$ **then**
  $G_{\text{seg}} = \mathsf{Merge}(\mathcal{G}_{\text{seg}})$
  $V^* = \mathsf{Nodes}(G_{\text{seg}}) \cap \mathsf{Nodes}(G_{\tau_{j+1}})$
  $\mathbf{H}^{\text{seg}} = \mathsf{GNN}(G_{\text{seg}}, \mathbf{F})$
  $\mathbf{H}^{j+1} = \mathsf{GNN}(G_{\tau_{j+1}}, \mathbf{F})$
if $\mathcal{G}_{seg} = \varnothing$ ***or*** *Dist*$(\mathbf{H}^{seg}_{V^*}, \mathbf{H}^{j+1}_{V^*}) > \theta$ **then**
  $\mathcal{G}_{\text{seg}} = \{G_{\tau_{j+1}}\}$ `/* Start a new graph stream segment. */`
else
  $\mathcal{G}_{\text{seg}} = \mathcal{G}_{\text{seg}} \cup \{G_{\tau_{j+1}}\}$ `/* Add` $G_{\tau_{j+1}}$ `to the current segment. */`
return $\mathcal{G}_{\text{seg}}$

Part III

Conclusions and Future Directions

Chapter 10

Conclusions

Graphs and tensors provide a powerful framework to model real-world entities and their relationships, and their evolution over time. *Mining and learning with graph and tensors* plays a pivotal role in understanding how real-world networks form and evolve, and in utilizing them for various downstream tasks. To this end, this thesis addresses important mining and learning tasks for both static and dynamic graphs and tensors by developing a suite of effective and efficient tools for

(1) analyzing and modeling large dynamic real-world networks (*e.g.*, deep graph clustering, temporal evolution modeling, computational scientific discovery, node importance estimation, and tensor factorization), and
(2) designing algorithms that leverage graphs and tensors for knowledge inference and reasoning applicable for several application domains (*e.g.*, reasoning over knowledge graphs, and explainable recommendation).

Below, we summarize the contributions of this thesis, organized into two parts on static and dynamic graphs and tensors.

10.1 Summary of Contributions

10.1.1 Part I: Static Graphs and Tensors

In Part I, we develop effective and scalable algorithms for mining and modeling static graphs and tensors, namely, node importance estimation, recommendation justification, distributed tensor decomposition, and automatic selection of graph learning models.

- **Estimating Node Importance in Knowledge Graphs (Chapters 2 and 3).** We explore graph-regularized semi-supervised machine learning algorithms for node importance estimation (NIE) in KGs from heterogeneous information reflecting node importance, which completely differ from previous non-trainable NIE methods. Specifically, we develop two algorithms, GENI and MULTIIMPORT, novel semi-supervised GNN-based methods that infer node importance by learning to combine heterogeneous

information from multiple sources (*e.g.*, input signals on node popularity, graph structure, and edge types). The proposed algorithms estimate node importance *up to 24% more accurately* than the best existing method.

- **Principled and Scalable Recommendation Justification (Chapter 4).** We develop J-RECS, a principled graph-based post-hoc framework to infer personalized recommendation justification. J-RECS is guided by a set of principles characterizing desirable justifications, and does not require manually labeled data. Justifications generated by J-RECS match user preference up to *21% more accurately* than the best baseline. Also, J-RECS is scalable, *running in time linear* in the size of input data.
- **Fast and Scalable Distributed Boolean Tensor Factorization (Chapter 5).** We develop two distributed algorithms, DBTF-CP and DBTF-TK, for distributed Boolean CP and Tucker factorizations, respectively, which are carefully designed to achieve high speed and scalability. These algorithms are the first distributed algorithms for large-scale Boolean tensor factorization. DBTF-CP decomposes up to 16^3–$32^3\times$ *larger tensors* than existing methods in *82–180× less time*. DBTF-TK decomposes up to 8^3–$16^3\times$ *larger tensors* than existing methods in *86–129× less time*.
- **Fast and Automatic Model Selection for Graph Representation Learning (Chapter 6).** We develop the *first meta-learning approach for automatic graph representation learning*, called AUTOGRL, which automatically infers the best model for the new graph *without requiring model training or evaluations*, by capitalizing on the prior performances of a large body of existing methods. Using AUTOGRL for model selection achieves up to *15× higher mean average precision (MAP)* than consistently applying a popular method like node2vec, as well as obtaining *10% higher MAP* than the best existing meta-learner, while *incurring negligible overhead (<1 sec)* at inference time.

10.1.2 Part II: Dynamic Graphs and Tensors

In Part II, we develop effective and efficient algorithms for the mining, inference, and learning with dynamic graphs and tensors, namely, finding mathematical expressions that model dynamic real-world processes, reasoning over temporal knowledge graphs, and detecting communities in graph data and tracking their evolution.

- **Knowledge-Guided Dynamic Systems Modeling (Chapter 7).** Towards an accurate and interpretable modeling of dynamic systems, we propose *knowledge-guided model revision*, which optimizes both the parameters and the structure of the model (*e.g.*, differential equations describing real-world phenomena) guided by both prior knowledge and data, such that the revised model accurately estimates the underlying process, while being consistent with domain knowledge. In experiments on forecasting the river water quality, the proposed GMR framework achieves the best forecasting accuracy, with *up to 34% lower error* than the best parameter fitting approach, while achieving $607\times$ *speedup* by employing the proposed orthogonal speedup techniques.
- **Reasoning over Temporal Knowledge Graphs (TKGs) (Chapter 8).** We identify the two major tasks that need to be addressed for an effective reasoning over TKGs,

i.e., modeling the event time and evolving network structure, and develop EvoKG, an effective framework that jointly models both tasks by capturing structural and temporal dynamics in TKGs via recurrent event modeling, temporal neighborhood aggregation, and modeling event time using neural density estimation. In comparison to the best existing method, EvoKG achieves up to *116%* and *77% better link and event time prediction accuracy*, while being up to *30×* and *291× times faster* in performing model training and inference, respectively.

- **Contrastive Graph Clustering for Community Detection and Tracking (Chapter 9).** We develop CGC, a novel end-to-end framework for graph clustering, which fundamentally differs from existing deep graph clustering (DGC) methods. CGC learns node embeddings and cluster assignments in a contrastive graph learning framework, where positive and negative samples are carefully selected in a multi-level scheme such that they reflect the hierarchical community structures and network homophily. Also, we extend CGC for time-evolving data, where temporal graph clustering is performed in an incremental learning fashion. In extensive evaluation of clustering quality on both static and temporal real-world datasets, CGC consistently outperforms various existing methods, achieving *up to 27% higher node clustering accuracy* and *29% higher temporal link prediction accuracy* than the previous best DGC method.

Chapter 11

Future Directions

The explosive growth of online platforms and Internet of Things presents rich and ever-increasing information to understand how real-world networks form and evolve, and to develop novel applications and intelligent services operating on graphs and tensors. Building on the contributions of this thesis, there are three major directions going forward to make *mining and learning with graphs and tensors* more effective and broadly useful:

(1) designing approaches for detecting more general and complex anomalies occurring in real-world networks (*e.g.*, anomalous motifs, and anomalies from multimodal data sources, such as graphs, text, and images)
(2) developing novel learning algorithms and frameworks for modeling the dynamics of real-world networks in light of co-evolving signals (*e.g.*, graph structural changes, temporal patterns, and trajectories of movement), and
(3) developing mechanisms for knowledge reasoning and inference over graphs, applicable for time-evolving graphs, and for providing explainability capabilities.

By tackling these interrelated problems, we can develop new insights and holistic solutions to maximize real-world impact. We discuss the research opportunities and challenges for each of these directions below.

11.1 Complex Anomaly Detection

Identifying patterns and building models for real-world networks enable us to detect anomalies in the network, such as abusive behaviors, fraudulent users, and compromised accounts. So far, existing methods for detecting anomalies in networks have mainly focused on relatively simple cases. For instance, many of them look for anomalous edges or dense subgraphs in plain static graphs. Accordingly, they miss more interesting and sophisticated anomalies, which might be hidden until we consider more complex and diverse aspects simultaneously, *e.g.*, graph substructures different from edges or cliques, and deviation from the norm in terms of both temporal and structural aspects, not just

from the structural point of view. Our goal is to develop holistic methods for complex anomaly detection, including the followings.

Anomalous Motif Detection. A motif refers to a specific graph substructure, such as a triangle or a star. One proposed direction is to detect anomalous occurrences of motifs from a large time-evolving network. This requires answering several challenging questions, including how to quantify the anomalousness of different types of motifs, and how to efficiently detect abnormal occurrences of motifs in near real-time.

Multimodal Anomaly Detection. In contrast to existing methods that can deal with just a single data source, we aim to detect anomalies in multimodal data drawn from heterogeneous sources (*e.g.*, graphs, text, and images). One of the major challenges lies in modeling the characteristics of real-world multimodal data, which may violate the usual, textbook assumptions, *e.g.*, on the independent and identically distributed data. To address this challenge, we plan to investigate the characteristics of multimodal anomalies in real-world networks, what assumptions we need to make to capture them accurately, and based on those assumptions, how to score the anomalousness of multimodal anomalies effectively and efficiently.

11.2 Modeling Dynamic Networks

Building upon our work on dynamic graphs and tensors, we will further improve our understanding of dynamic networks by developing effective computational tools for automatic scientific discovery and temporal evolution modeling.

Computational Scientific Discovery. Given longitudinal measurements on multi-aspect networks, such as sensor measurement data, how can we automatically extract scientific knowledge that can describe the observed phenomena? In Chapter 7, we presented one answer to this question by developing GMR, an evolutionary framework for knowledge-guided model revision. Our experience of applying GMR for ecological modeling shows that different science and engineering domains may present distinct challenges and constraints that need to be addressed for an effective modeling of the domain. Thus, we plan to devise versatile and expressive mechanisms to represent and improve knowledge in different domains (*e.g.*, designing new language biases and search operators to influence the search, as well as to represent new types of knowledge and constraints). Further, we plan to expand the toolbox for scientific discovery with different machine learning methods, such as deep learning models; research topics include incorporating physical constraints and prior knowledge into the architecture and learning objective of the model, which involves addressing problems such as how to parameterize theoretical physical laws, and how to design specialized architectures for important science domains.

Temporal Evolution Modeling. One of the crucial tasks for temporal evolution modeling is to detect communities in the network and track their evolution. While many community detection methods exist, they are mostly designed for static networks, and a few dynamic methods mainly focus on structural changes. By considering a wider range of signals altogether (e.g., temporal patterns like periodicity and seasonality, spatial

distributions, graph structural changes, along with node and edge attributes), more interesting types of communities can be discovered. To achieve this goal requires novel methods that can jointly model heterogeneous signals in a self-supervised manner. Our research shows that contrastive learning presents a promising framework to this end, where our understanding of real-world networks and their characteristics (e.g., network homophily and hierarchical community structures) are used to find communities in light of multiple signals. Building upon this success, we will develop self-supervised techniques that will enable more interesting and novel community discovery.

Identifying important nodes in a graph and how their importance changes over time is another important task for modeling dynamic networks, which can be used to provide more accurate and temporally relevant recommendation and search services, among many other applications. To this end, we will develop a framework that infers dynamically changing node importance by learning to fuse multiple types of signals that capture certain aspects of node importance, and model their evolution over time.

11.3 Knowledge Reasoning

Multi-aspect networks, such as knowledge graphs (KGs), are a rich source of information for intelligent services and applications. I plan to develop techniques that employ KGs for knowledge reasoning and explainability.

Reasoning over Temporal Knowledge Graphs. In addition to modeling various relations among entities, temporal knowledge graphs (TKGs) provide rich information to capture the temporal patterns of real-world events. While modeling TKGs has been receiving increasing attention recently, knowledge reasoning over TKGs largely remains limited to a single-hop reasoning, *i.e.*, predicting a missing link that directly connects two nodes in a TKG. A more effective reasoning over TKGs would need to infer new knowledge by considering multiple connected links, *i.e.*, multi-hop reasoning; this is a challenging problem, especially for TKGs, as the temporal evolution and temporal constraints of TKGs need to be taken into account for multi-hop reasoning.

Explainability. Capabilities to reason over complex networks can provide explainability for a wide variety of tasks. Towards this goal, we plan to develop (1) graph machine learning algorithms with explainable and interpretable mechanisms, and design (2) effective techniques that utilize KGs to provide explanations for different machine learning tasks, including recommendation, entity classification, and anomaly detection.

Bibliography

[ABK16] W. Austin, G. Ballard, and T. G. Kolda. Parallel tensor compression for large-scale scientific data. In *IPDPS*, 2016. 98

[ABvRV18] Salisu Mamman Abdulrahman, Pavel Brazdil, Jan N. van Rijn, and Joaquin Vanschoren. Speeding up algorithm selection using average ranking and active testing by introducing runtime. *Mach. Learn.*, 107(1):79–108, 2018. 152

[AG11] Thomas Aynaud and Jean-Loup Guillaume. Multi-step community detection and hierarchical time segmentation in evolving networks. In *Proceedings of the 5th SNA-KDD workshop*, volume 11, 2011. 226

[ARL+18] Nesreen K. Ahmed, Ryan A. Rossi, John Boaz Lee, Xiangnan Kong, Theodore L. Willke, Rong Zhou, and Hoda Eldardiry. Learning role-based graph embeddings. *CoRR*, abs/1802.02896, 2018. 148, 154

[ATA18] S. Acer, T. Torun, and C. Aykanat. Improving medium-grain partitioning for scalable sparse tensor decomposition. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2018. 98

[ATMF12] Leman Akoglu, Hanghang Tong, Brendan Meeder, and Christos Faloutsos. PICS: parameter-free identification of cohesive subgroups in large attributed graphs. In *SDM*, pages 439–450. SIAM / Omnipress, 2012. 226

[AWWB09] Michael Affenzeller, Stephan M. Winkler, Stefan Wagner, and Andreas Beham. *Genetic Algorithms and Genetic Programming - Modern Concepts and Practical Applications*. CRC Press, 2009. 162

[BB87] Linfield C Brown and Thomas O Barnwell. *The enhanced stream water quality models QUAL2E and QUAL2E-UNCAS: Documentation and user manual*. US Environmental Protection Agency. Office of Research and Development, Environmental Research Laboratory., 1987. 183

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, 2012. 152

[BBR+18] Mohamed Ishmael Belghazi, Aristide Baratin, Sai Rajeswar, Sherjil Ozair, Yoshua Bengio, R. Devon Hjelm, and Aaron C. Courville. Mutual information neural estimation. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 530–539. PMLR, 2018. 211

[BC17] Or Biran and Courtenay Cotton. Explanation and justification in machine learning: A survey. In *IJCAI-17 workshop on explainable AI (XAI)*, volume 8, 2017. 4, 64

[BEP+08] Kurt D. Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*, pages 1247–1250, 2008. 3, 17, 20, 28, 44, 53

[BGV13] Radim Belohlávek, Cynthia Vera Glodeanu, and Vilém Vychodil. Optimal factorization of three-way binary data using triadic concepts. *Order*, 30(2):437–454, 2013. 88, 97, 120

[BKEF12] Michele Berlingerio, Danai Koutra, Tina Eliassi-Rad, and Christos Faloutsos. Netsimile: A scalable approach to size-independent network similarity. *CoRR*, abs/1209.2684, 2012. 153

[BLL21] Chenyang Bu, Yi Lu, and Fei Liu. Automatic graph learning with evolutionary algorithms: An experimental study. In *PRICAI (1)*, volume 13031 of *Lecture Notes in Computer Science*, pages 513–526. Springer, 2021. 152

[BLO+15] Elizabeth Boschee, Jennifer Lautenschlager, Sean O'Brien, Steve Shellman, James Starz, and Michael Ward. Icews coded event data. *Harvard Dataverse*, 12, 2015. 188, 196

[BM05] Mustafa Bilgic and Raymond J Mooney. Explaining recommendations: Satisfaction vs. promotion. In *Beyond Personalization Workshop, IUI*, volume 5, 2005. 64, 82

[BS93] Stephen T. Barnard and Horst D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In *PPSC*, pages 711–718. SIAM, 1993. 226

[BS06] Tanya Y. Berger-Wolf and Jared Saia. A framework for analysis of dynamic social networks. In *KDD*, pages 523–528. ACM, 2006. 226

[BSMN03] OM Becker, S Shacham, Y Marantz, and S Noiman. Modeling the 3d structure of gpcrs: advances and application to drug discovery. *Current opinion in drug discovery & development*, 6(3), 2003. 161

[BTK+14] Alex Beutel, Partha Pratim Talukdar, Abhimanu Kumar, Christos Faloutsos, Evangelos E. Papalexakis, and Eric P. Xing. Flexifact: Scalable flexible factorization of coupled tensors on hadoop. In *SDM*, pages 109–117, 2014. 97

[BUG+13] Antoine Bordes, Nicolas Usunier, Alberto García-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *NIPS*, pages 2787–2795, 2013. 28, 37, 53, 204

[BWS+20] Deyu Bo, Xiao Wang, Chuan Shi, Meiqi Zhu, Emiao Lu, and Peng Cui. Structural deep clustering network. In *WWW*, pages 1400–1410. ACM / IW3C2, 2020. 13, 208, 219, 220, 221, 224, 226, 231, 232

[BWY13] Denilson Barbosa, Haixun Wang, and Cong Yu. Shallow information extraction for the knowledge web. In *ICDE*, pages 1264–1267, 2013. 3, 17

[BZSL14] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. In *ICLR*, 2014. 35

[CBRB09] Loïc Cerf, Jérémy Besson, Céline Robardet, and Jean-Franccois Boulicaut. Closed patterns meet *n*-ary relations. *TKDD*, 3(1), 2009. 97

[CCJ+17] Venkatesan T. Chakaravarthy, Jee W. Choi, Douglas J. Joseph, Xing Liu, Prakash Murali, Yogish Sabharwal, and Dheeraj Sreedhar. On optimizing distributed tucker decomposition for dense tensors. *CoRR*, abs/1707.05594, 2017. 98

[CCL+21] Lei Cai, Zhengzhang Chen, Chen Luo, Jiaping Gui, Jingchao Ni, Ding Li, and Haifeng Chen. Structural temporal graph neural networks for anomaly detection in dynamic graphs. In *CIKM*, pages 3747–3756. ACM, 2021. 7, 136

[CCX+19] Xu Chen, Hanxiong Chen, Hongteng Xu, Yongfeng Zhang, Yixin Cao, Zheng Qin, and Hongyuan Zha. Personalized fashion recommendation with visual explanations based on multimodal attention network: Towards visually explainable recommendation. In *SIGIR*, pages 765–774, 2019. 82

[Citne] CiteSeer. https://citeseerx.ist.psu.edu, 2021 [Online]. Accessed: 2021-10-01. 218

[CJX20] Xiaojun Chen, Shengbin Jia, and Yang Xiang. A review: Knowledge reasoning over knowledge graph. *Expert Syst. Appl.*, 141, 2020. 188

[CKNH20] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey E. Hinton. A simple framework for contrastive learning of visual representations. In *ICML*, volume 119 of *Proceedings of Machine Learning Research*, pages 1597–1607. PMLR, 2020. 209, 211

[CLX15] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In *CIKM*, pages 891–900. ACM, 2015. 154

[CM18] Joseph Crawford and Tijana Milenković. Cluenet: Clustering a temporal network based on topological similarity rather than denseness. *PLOS ONE*, 13(5):1–25, 05 2018. 226

[CMX18] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *ICLR*, 2018. 35

[CQL+07] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. In *ICML*, pages 129–136, 2007. 49, 141

[CS98] Kyung-Je Cho and Jae-Ki Shin. Growth and nutrient kinetics of some algal species iso-lated from the naktong river. *Algae*, 13(2), 1998. 165

[CSZ+07] Yun Chi, Xiaodan Song, Dengyong Zhou, Koji Hino, and Belle L. Tseng. Evolutionary spectral clustering by incorporating temporal smoothness. In *KDD*, pages 153–162. ACM, 2007. 226

[CV14] Joon Hee Choi and S Vishwanathan. Dfacto: Distributed factorization of tensors. In *NIPS*, 2014. 98

[CZH+17] Jingyuan Chen, Hanwang Zhang, Xiangnan He, Liqiang Nie, Wei Liu, and Tat-Seng Chua. Attentive collaborative filtering: Multimedia recommendation with item- and component-level attention. In *SIGIR*, pages 335–344, 2017. 82

[dARF17] Miguel Ramos de Araujo, Pedro Manuel Pinto Ribeiro, and Christos Faloutsos. TensorCast: Forecasting with context using coupled tensors (best paper award). In *ICDM*, pages 71–80. IEEE Computer Society, 2017. 204

[DBHS10] Xin Dong, Laure Berti-Équille, Yifan Hu, and Divesh Srivastava. Global detection of complex copying relationships between sources. *Proc. VLDB Endow.*, 3(1):1358–1369, 2010. 59

[DBLne] DBLP. https://dblp.org, 2021 [Online]. Accessed: 2021-10-01. 218, 219

[DBS09] Xin Luna Dong, Laure Berti-Équille, and Divesh Srivastava. Integrating conflicting data: The role of source dependence. *Proc. VLDB Endow.*, 2(1), 2009. 59

[DBV16] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, pages 3837–3845, 2016. 19, 35

[DDT+16] Nan Du, Hanjun Dai, Rakshit Trivedi, Utkarsh Upadhyay, Manuel Gomez-Rodriguez, and Le Song. Recurrent marked temporal point processes: Embedding event history to vector. In *KDD*, pages 1555–1564. ACM, 2016. 198

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004. 98

[DHW+17] Li Dong, Shaohan Huang, Furu Wei, Mirella Lapata, Ming Zhou, and Ke Xu. Learning to generate product reviews from attributes. In *EACL*, 2017. 64

[DLT07] Sašo Džeroski, Pat Langley, and Ljupčo Todorovski. Computational discovery of scientific knowledge. pages 1–14. Springer, 2007. 162

[DLZ18] Tim Donkers, Benedikt Loepp, and Jürgen Ziegler. Explaining recommendations by means of user reviews. In *IUI Workshops*, volume 2068 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018. 82

[DMSR18] Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2d knowledge graph embeddings. In *AAAI*, pages 1811–1818. AAAI Press, 2018. 188, 199, 204

[DNM+12] Ngoc Phong Dao, Xuan Hoai Nguyen, Robert Ian (Bob) McKay, Constantin Siriteanu, Quang Uy Nguyen, and Namyong Park. Evolving the best known approximation to the Q function. In *GECCO*, pages 807–814, 2012. 162

[DQW+14] Qiming Diao, Minghui Qiu, Chao-Yuan Wu, Alexander J. Smola, Jing Jiang, and Chong Wang. Jointly modeling aspects, ratings and sentiments for movie recommendation (JMARS). In *KDD*, pages 193–202, 2014. 82

[DRT18] Shib Sankar Dasgupta, Swayambhu Nath Ray, and Partha P. Talukdar. HyTE: Hyperplane-based temporally aware knowledge graph embedding. In *EMNLP*, pages 2001–2011. Association for Computational Linguistics, 2018. 188, 199, 204

[DSG94] Qingyun Duan, Soroosh Sorooshian, and Vijai K Gupta. Optimal use of the sce-ua global optimization method for calibrating watershed models. *Journal of hydrology*, 158(3-4):265–284, 1994. 179

[DWZX15] Li Dong, Furu Wei, Ming Zhou, and Ke Xu. Question answering over freebase with multi-column convolutional neural networks. In *ACL*, pages 260–269, 2015. 3, 17

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996. 50

[EM13a] Dóra Erdős and Pauli Miettinen. Discovering facts with boolean tensor tucker decomposition. In *CIKM*, pages 1569–1572, 2013. 88, 97

[EM13b] Dóra Erdős and Pauli Miettinen. Walk 'n' merge: A scalable algorithm for boolean tensor factorization. In *ICDM*, pages 1037–1042, 2013. 88, 96, 97, 120, 121, 126

[FKH18] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: robust and efficient hyperparameter optimization at scale. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 1436–1445. PMLR, 2018. 152

[FML+19] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Yihong Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. In *WWW*, 2019. 64, 136

[FMT04] Christos Faloutsos, Kevin S. McCurley, and Andrew Tomkins. Fast discovery of connection subgraphs. In *KDD*, pages 118–127, 2004. 83

[Foune] Foursquare. https://foursquare.com, 2021 [Online]. Accessed: 2021-10-01. 219

[GCC20] Palash Goyal, Sujit Rokka Chhetri, and Arquimedes Canedo. dyngraph2vec: Capturing network dynamics using dynamic graph representation learning. *Knowl. Based Syst.*, 187, 2020. 13, 199, 204, 208, 219, 221, 224, 227

[GDC10] Derek Greene, Dónal Doyle, and Padraig Cunningham. Tracking the evolution of communities in dynamic social networks. In *ASONAM*, pages 176–183. IEEE Computer Society, 2010. 226

[GDN18] Alberto García-Durán, Sebastijan Dumancic, and Mathias Niepert. Learning sequence encoders for temporal knowledge graph completion. In *EMNLP*, pages 4816–4821. Association for Computational Linguistics, 2018. 188, 190, 199, 204

[Geo21] PyTorch Geometric. Pyg. https://github.com/pyg-team/pytorch_geometric, 2021. Accessed: 2021-10-20. 232

[GGLY17] Xifeng Guo, Long Gao, Xinwang Liu, and Jianping Yin. Improved deep embedded clustering with local structure preservation. In *IJCAI*, pages 1753–1759. ijcai.org, 2017. 13, 208, 219, 220, 225

[GH10] Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *AISTATS*, volume 9 of *JMLR Proceedings*, pages 297–304. JMLR.org, 2010. 211

[GKBP20] Rishab Goel, Seyed Mehran Kazemi, Marcus Brubaker, and Pascal Poupart. Diachronic embedding for temporal knowledge graph completion. In *AAAI*, pages 3988–3995. AAAI Press, 2020. 204

[GKHL18] Palash Goyal, Nitin Kamra, Xinran He, and Yan Liu. DynGEM: Deep embedding method for dynamic graphs. *CoRR*, abs/1805.11273, 2018. 204, 219, 221, 224, 227

[GL16] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *KDD*, pages 855–864, 2016. 30, 39, 146, 148, 154, 203

[GN02] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002. 226

[GPP20] Ekta Gujral, Ravdeep Pasricha, and Evangelos E. Papalexakis. Beyond rank-1: Discovering rich community structure in multi-aspect graphs. In *WWW*, pages 452–462. ACM / IW3C2, 2020. 203, 226

[GR71] Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. In *Linear algebra*, pages 134–151. Springer, 1971. 219, 220, 221, 224, 226

[GR06] Oleg Grodzevich and Oleksandr Romanko. Normalization and other topics in multi-objective optimization. 2006. 71

[GSM+17] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *KDD*, pages 1487–1495. ACM, 2017. 152

[GSR+17] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *ICML*, pages 1263–1272, 2017. 22, 44, 60

[GTW+15] Rong Gu, Yun Tang, Zhaokang Wang, Shuai Wang, Xusen Yin, Chunfeng Yuan, and Yihua Huang. Efficient large scale distributed matrix computation with spark. In *IEEE BigData*, pages 2327–2336, 2015. 99

[GYZB21] Mengying Guo, Tao Yi, Yuqing Zhu, and Yungang Bao. Jitune: Just-in-time hyperparameter tuning for network embedding algorithms. *CoRR*, abs/2101.06427, 2021. 152

[had] Apache hadoop. http://hadoop.apache.org/. 98

[Hav02] Taher H. Haveliwala. Topic-sensitive pagerank. In *WWW*, pages 517–526, 2002. 3, 19, 21, 29, 34, 35, 42, 56, 59, 69, 77, 82, 83

[HBL15] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *CoRR*, abs/1506.05163, 2015. 35

[HDWS20] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. Heterogeneous graph transformer. In *WWW*, pages 2704–2710. ACM / IW3C2, 2020. 145, 155

[HFL+19] R. Devon Hjelm, Alex Fedorov, Samuel Lavoie-Marchildon, Karan Grewal, Philip Bachman, Adam Trischler, and Yoshua Bengio. Learning deep representations by mutual information estimation and maximization. In *ICLR*. OpenReview.net, 2019. 211

[HJ02] Pei Hongping and Ma Jianyi. Study on the algal dynamic model for west lake, hangzhou. *Ecological Modelling*, 148(1):67–77, 2002. 165

[HKCCB15] Tobias Houska, Philipp Kraft, Alejandro Chamorro-Chavez, and Lutz Breuer. Spotting model parameters using a ready-made python package. *PloS one*, 10(12):e0145180, 2015. 186

[HKR00] Jonathan L. Herlocker, Joseph A. Konstan, and John Riedl. Explaining collaborative filtering recommendations. In *CSCW*, pages 241–250. ACM, 2000. 4, 64, 82

[HLZ+17] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *WWW*, pages 173–182. ACM, 2017. 147

[HMW+20] Zhen Han, Yunpu Ma, Yuyi Wang, Stephan Günnemann, and Volker Tresp. Graph hawkes neural network for forecasting on temporal knowledge graphs. In *AKBC*, 2020. 192, 198, 199, 204, 205

[HS06] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006. 13, 208, 219, 220

[HW79] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979. 219, 220

[HYL17] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1025–1035, 2017. 19, 22, 35, 148, 154

[HZC21] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowl. Based Syst.*, 212:106622, 2021. 151

[JHRK19] Di Jin, Mark Heimann, Ryan A. Rossi, and Danai Koutra. node2bits: Compact time- and attribute-aware node representations for user stitching. In *ECML/PKDD (1)*, volume 11906 of *Lecture Notes in Computer Science*, pages 483–506. Springer, 2019. 154

[JJSK16] ByungSoo Jeon, Inah Jeon, Lee Sael, and U. Kang. Scout: Scalable coupled matrix-tensor factorization - algorithm and discoveries. In *ICDE*, pages 811–822, 2016. 97, 203

[JL21] Weiwei Jiang and Jiayun Luo. Graph neural network for traffic forecasting: A survey. *CoRR*, abs/2101.11174, 2021. 7, 136

[JPB20] ByungSoo Jeon, Namyong Park, and Seojin Bang. Dropout prediction over weeks in moocs via interpretable multi-layer representation learning. *CoRR*, abs/2002.01598, 2020. 161

[JPC+20] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S. Yu. A survey on knowledge graphs: Representation, acquisition and applications. *CoRR*, abs/2002.00388, 2020. 187

[JPF+16] Inah Jeon, Evangelos E. Papalexakis, Christos Faloutsos, Lee Sael, and U. Kang. Mining billion-scale tensors: algorithms and discoveries. *VLDB J.*, 25(4):519–544, 2016. 97, 98

[JPSK17] Jinhong Jung, Namyong Park, Lee Sael, and U. Kang. Bepi: Fast and memory-efficient method for billion-scale random walk with restart. In *SIGMOD*, 2017. 34, 35, 42, 82

[JQJR20] Woojeong Jin, Meng Qu, Xisen Jin, and Xiang Ren. Recurrent event network: Autoregressive structure inferenceover temporal knowledge graphs. In *EMNLP (1)*, pages 6669–6683. Association for Computational Linguistics, 2020. 188, 190, 199, 200, 204, 205

[JS97] Aravind K Joshi and Yves Schabes. Tree-adjoining grammars. In *Handbook of formal languages*, pages 69–123. Springer, 1997. 167, 168

[JTT06] Liping Ji, Kian-Lee Tan, and Anthony K. H. Tung. Mining frequent closed cubes in 3d datasets. In *VLDB*, pages 811–822, 2006. 97

[JZT+17] Zhuxi Jiang, Yin Zheng, Huachun Tan, Bangsheng Tang, and Hanning Zhou. Variational deep embedding: An unsupervised and generative approach to clustering. In *IJCAI*, pages 1965–1972. ijcai.org, 2017. 225

[KAF+17] Anuj Karpatne, Gowtham Atluri, James H. Faghmous, Michael S. Steinbach, Arindam Banerjee, Auroop R. Ganguly, Shashi Shekhar, Nagiza F. Samatova, and Vipin Kumar. Theory-guided data science: A new paradigm for scientific discovery from data. *TKDE*, 29(10):2318–2331, 2017. 162, 183

[KB09] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009. 60, 88, 93, 95, 203

[KBV09] Yehuda Koren, Robert M. Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8), 2009. 64

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998. 226

[KKNK18] Kangil Kim, Dong-Kyun Kim, Junhyug Noh, and Minhyeok Kim. Stable forecasting of environmental time series via long short term memory recurrent neural network. *IEEE Access*, 6:75216–75228, 2018. 183

[KKU16] Lars Karlsson, Daniel Kressner, and Andre Uschmajew. Parallel algorithms for tensor completion in the cp format. *Parallel Computing*, 57:222 – 234, 2016. 97

[Kle99] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999. 34, 42, 77, 82

[KLR04] Eamonn J. Keogh, Stefano Lonardi, and Chotirat (Ann) Ratanamahatana. Towards parameter-free data mining. In *KDD*, pages 206–215, 2004. 80, 85

[KMS+10] Dong-Kyun Kim, Bob McKay, Haisoo Shin, Yun-Geun Lee, and Xuan Hoai Nguyen. Ecological application of evolutionary computation: Improving water quality forecasts for the nakdong river, korea. In *CEC*, pages 1–8, 2010. 9, 161, 183, 185

[KMST10] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC - instance-specific algorithm configuration. In *ECAI*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 751–756. IOS Press, 2010. 147, 152

[Koz93] John R. Koza. *Genetic programming - on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, 1993. 162

[KPHF12] U. Kang, Evangelos E. Papalexakis, Abhay Harpale, and Christos Faloutsos. Gigatensor: scaling tensor analysis up by 100 times - algorithms and discoveries. In *KDD*, pages 316–324, 2012. 97, 99

[KPJY16] Hanjoo Kim, Jaehong Park, Jaehee Jang, and Sungroh Yoon. Deepspark: Spark-based deep learning supporting asynchronous updates and caffe compatibility. *CoRR*, abs/1602.08191, 2016. 99

[KPM+14] MinHyeok Kim, Namyong Park, RI Bob McKay, Haisoo Shin, Yun-Geun Lee, Kwang-Seuk Jeong, and Dong-Kyun Kim. Improvement of complex and refractory ecological models: Riverine water quality modelling using evolutionary computation. *Ecological Modelling*, 291:205–217, 2014. 183

[KS08] Tamara G. Kolda and Jimeng Sun. Scalable tensor decompositions for multi-aspect data mining. In *ICDM*, pages 363–372, 2008. 98

[KT06] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education, 2006. 73

[KTF09] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A peta-scale graph mining system. In *ICDM*, pages 229–238, 2009. 99

[KTS+] U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. GBASE: a scalable and general graph management system. In *KDD*. 99

[KTW+20] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised contrastive learning. *CoRR*, abs/2004.11362, 2020. 209

[KU15] Oguz Kaya and Bora Uccar. Scalable sparse tensor decompositions in distributed memory systems. In *SC*, pages 1–11, 2015. 98

[KU16] Oguz Kaya and Bora Uccar. High performance parallel algorithms for the tucker decomposition of sparse tensors. In *ICPP*, 2016. 98

[Kuh55] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955. 222

[KV13] Vasiliki Kalavri and Vladimir Vlassov. Mapreduce: Limitations, optimizations and open issues. In *TrustCom*, pages 1031–1038, 2013. 99

[KW16] Thomas N. Kipf and Max Welling. Variational graph auto-encoders. *CoRR*, abs/1611.07308, 2016. 13, 208, 219, 220, 221, 224, 226

[KW17] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR (Poster)*. OpenReview.net, 2017. 19, 22, 35, 56, 60, 146, 152, 154, 203, 219

[KWRK17] Anuj Karpatne, William Watkins, Jordan S. Read, and Vipin Kumar. Physics-guided neural networks (PGNN): an application in lake temperature modeling. *CoRR*, abs/1710.11431, 2017. 183

[KZL19] Srijan Kumar, Xikun Zhang, and Jure Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. In *KDD*, pages 1269–1278. ACM, 2019. 204

[LC18] Julien Leblay and Melisachew Wudage Chekol. Deriving validity time in knowledge graph. In *WWW (Companion Volume)*, pages 1771–1776. ACM, 2018. 188, 196, 204

[LCH17] Junying Li, Deng Cai, and Xiaofei He. Learning graph-level representation for drug discovery. *CoRR*, abs/1709.03741, 2017. 7, 136

[LCP+17] Jiajia Li, Jee Choi, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. Model-driven sparse cp decomposition for higher-order tensors. In *IPDPS*, 2017. 97

[LDZ19] Rui Lu, Zhiyao Duan, and Changshui Zhang. Audio-visual deep clustering for speech separation. *IEEE ACM Trans. Audio Speech Lang. Process.*, 27(11):1697–1712, 2019. 13, 208

[LF12] Andrea Lancichinetti and Santo Fortunato. Consensus clustering in complex networks. *Scientific reports*, 2(1):1–7, 2012. 226

[Lib21] Deep Graph Library. Dgi. https://github.com/dmlc/dgl/tree/master/examples/pytorch/dgi, 2021. Accessed: 2021-10-20. 231, 232

[Libne] ACM Digital Library. https://dl.acm.org, 2021 [Online]. Accessed: 2021-10-01. 218

[LIJ+15] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey,

Patrick van Kleef, Sören Auer, and Christian Bizer. Dbpedia - A large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015. 3, 17, 44

[LJD+17] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18:185:1–185:52, 2017. 152

[LL19] Petro Liashchynskyi and Pavlo Liashchynskyi. Grid search, random search, genetic algorithm: A big comparison for NAS. *CoRR*, abs/1912.06059, 2019. 152

[LLC09] Chi-Jie Lu, Tian-Shyug Lee, and Chih-Chou Chiu. Financial time series forecasting using independent component analysis and support vector regression. *Decision Support Systems*, 47(2):115–125, 2009. 9, 161

[LM01] Ronny Lempel and Shlomo Moran. SALSA: the stochastic approach for link-structure analysis. *ACM Trans. Inf. Syst.*, 19(2), 2001. 77, 82

[LMV00] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. On the best rank-1 and rank-(R_1 , R_2, ... , R_N) approximation of higher-order tensors. *SIMAX*, 21(4):1324–1342, 2000. 97

[LNY12] Xutao Li, Michael K. Ng, and Yunming Ye. HAR: hub, authority and relevance scores in multi-relational data for query search. In *SDM*, pages 141–152, 2012. 3, 19, 21, 29, 34, 35, 38, 42, 56, 59, 75, 77, 82, 83

[LRC+15] Alessandro Lulli, Laura Ricci, Emanuele Carlini, Patrizio Dazzi, and Claudio Lucchese. Cracker: Crumbling large graphs into connected components. In *ISCC*, pages 574–581, 2015. 99

[LRK18] John Boaz Lee, Ryan A. Rossi, and Xiangnan Kong. Graph classification using structural attention. In *KDD*, pages 1666–1674, 2018. 60

[LS13] Kalev Leetaru and Philip A Schrodt. Gdelt: Global data on events, location, and tone, 1979–2012. In *ISA annual convention*, volume 2, pages 1–49. Citeseer, 2013. 188, 196

[LVMDBR99] Iwin Leenen, Iven Van Mechelen, Paul De Boeck, and Seymour Rosenberg. Indclas: A three-way hierarchical classes model. *Psychometrika*, 64(1):9–24, 1999. 88, 97

[LWH03] Bin Luo, Richard C. Wilson, and Edwin R. Hancock. Spectral embedding of graphs. *Pattern Recognit.*, 36(10):2213–2230, 2003. 154

[LXY+20] J. Liu, C. Xu, C. Yin, W. Wu, and Y. Song. K-core based temporal graph convolutional network for dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2020. 219, 221, 224, 227, 232

[LZL20] Peizhao Li, Han Zhao, and Hongfu Liu. Deep fair clustering for visual learning. In *CVPR*, pages 9067–9076. Computer Vision Foundation / IEEE, 2020. 225

[MBS15] Farzaneh Mahdisoltani, Joanna Biega, and Fabian M. Suchanek. YAGO3: A knowledge base from multilingual wikipedias. In *CIDR*. www.cidrdb.org, 2015. 196

[Mie11] Pauli Miettinen. Boolean tensor factorizations. In *ICDM*, 2011. 88, 96, 97, 99, 102, 120

[MM15] Saskia Metzler and Pauli Miettinen. Clustering boolean tensors. *DMKD*, 29(5):1343–1373, 2015. 88, 97, 104

[MMG+08] Pauli Miettinen, Taneli Mielikäinen, Aristides Gionis, Gautam Das, and Heikki Mannila. The discrete basis problem. *TKDE*, 20(10), 2008. 121

[MNL+16] Cataldo Musto, Fedelucio Narducci, Pasquale Lops, Marco de Gemmis, and Giovanni Semeraro. Explod: A framework for explaining recommendations based on the linked open data cloud. In *RecSys*, pages 151–154, 2016. 75, 76, 79, 82, 83

[Mon95] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995. 163

[Mos09] Brian R Moss. *Ecology of fresh waters: man and medium, past to future*. John Wiley & Sons, 2009. 164

[MS17] Mustafa Misir and Michèle Sebag. Alors: An algorithm recommender system. *Artif. Intell.*, 244:291–314, 2017. 147, 152

[MSC+13] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013. 203

[MSFK18] Naveen Sai Madiraju, Seid M. Sadat, Dimitry Fisher, and Homa Karimabadi. Deep temporal clustering : Fully unsupervised learning of time-domain features. *CoRR*, abs/1802.01059, 2018. 13, 208

[Ngu04] Xuan Hoai Nguyen. *A flexible representation for genetic programming: lessons from natural language processing*. PhD thesis, University of New South Wales, Australian Defence Force Academy, 2004. 163, 168, 169, 170

[NLM19] Jianmo Ni, Jiacheng Li, and Julian J. McAuley. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In *EMNLP-IJCNLP*, pages 188–197, 2019. 64

[NLR+18] Giang Hoang Nguyen, John Boaz Lee, Ryan A. Rossi, Nesreen K. Ahmed, Eunyee Koh, and Sungchul Kim. Continuous-time dynamic network

embeddings. In *WWW (Companion Volume)*, pages 969–976. ACM, 2018. 204, 219, 221, 224, 227

[NMA03] Xuan Hoai Nguyen, Robert I. McKay, and Hussein A. Abbass. Tree adjoining grammars, language bias, and genetic programming. In *EuroGP*, 2003. 169, 170

[NMJ13] Mladen Nikolić, Filip Marić, and Predrag Janičić. Simple algorithm portfolio for sat. *Artificial Intelligence Review*, 40(4):457–465, 2013. 147, 152

[NTK11] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A three-way model for collective learning on multi-relational data. In *ICML*, pages 809–816. Omnipress, 2011. 204

[NW78] George L. Nemhauser and Laurence A. Wolsey. Best algorithms for approximating the maximum of a submodular set function. *Math. Oper. Res.*, 3(3):177–188, 1978. 73

[OPJ+19] Sejoon Oh, Namyong Park, Jun-Gi Jang, Lee Sael, and U Kang. High-performance tucker factorization on heterogeneous platforms. *IEEE TPDS.*, 30(10):2237–2248, 2019. 60, 203

[OPSK18] Sejoon Oh, Namyong Park, Lee Sael, and U Kang. Scalable tucker factorization for sparse tensors - algorithms and discoveries. In *ICDE*, pages 1120–1131. IEEE Computer Society, 2018. 98, 226

[OR01] Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001. 163

[OSP+17] Jinoh Oh, Kijung Shin, Evangelos E. Papalexakis, Christos Faloutsos, and Hwanjo Yu. S-hot: Scalable high-order tucker decomposition. In *WSDM*, 2017. 98

[PAS14] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: online learning of social representations. In *KDD*, pages 701–710. ACM, 2014. 154, 203

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999. 3, 19, 21, 29, 34, 42, 56, 59, 79, 82

[PDC+20] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. EvolveGCN: Evolving graph convolutional networks for dynamic graphs. In *AAAI*, pages 5363–5370. AAAI Press, 2020. 190, 199, 204, 219, 221, 224, 227

[PFS12] Evangelos E. Papalexakis, Christos Faloutsos, and Nicholas D. Sidiropoulos. Parcube: Sparse parallelizable tensor decompositions. In *ECML PKDD*, pages 521–536, 2012. 97

[PHF+20] Shirui Pan, Ruiqi Hu, Sai-Fu Fung, Guodong Long, Jing Jiang, and Chengqi Zhang. Learning graph embedding with adversarial training methods. *IEEE Trans. Cybern.*, 50(6):2475–2487, 2020. 13, 208, 219, 220, 221, 224, 226

[PJLK16] Namyong Park, Byungsoo Jeon, Jungwoo Lee, and U Kang. Bigtensor: Mining billion-scale tensor made easy. In *CIKM*, pages 2457–2460. ACM, 2016. doi:10.1145/2983323.2983332 60, 97, 203

[PKD+19] Namyong Park, Andrey Kan, Xin Luna Dong, Tong Zhao, and Christos Faloutsos. Estimating node importance in knowledge graphs using graph neural networks. In *KDD*, pages 596–606. ACM, 2019. doi:10.1145/3292500.3330855 7, 17, 42, 52, 53, 56, 136, 161

[PKD+20] Namyong Park, Andrey Kan, Xin Luna Dong, Tong Zhao, and Christos Faloutsos. MultiImport: Inferring node importance in a knowledge graph from multiple input signals. In *KDD*, pages 503–512. ACM, 2020. doi:10.1145/3394486.3403093 41, 161

[PKFD20] Namyong Park, Andrey Kan, Christos Faloutsos, and Xin Luna Dong. J-Recs: Principled and scalable recommendation justification. In *ICDM*, pages 1208–1213. IEEE, 2020. doi:10.1109/ICDM50108.2020.00151 63

[PKH+21] Namyong Park, Minhyeok Kim, Nguyen Xuan Hoai, Robert I. McKay, and Dong-Kyun Kim. Knowledge-based dynamic systems modeling: A case study on modeling river water quality. In *ICDE*, pages 2231–2236. IEEE, 2021. doi:10.1109/ICDE51399.2021.00229 161

[PKP+18] Namyong Park, Eunjeong Kang, Minsu Park, Hajeong Lee, Hee-Gyung Kang, Hyung-Jin Yoon, and U. Kang. Predicting acute kidney injury in cancer patients using heterogeneous and irregular data. *PLOS ONE*, 13, 07 2018. 9, 161

[PLJH21] Zhihao Peng, Hui Liu, Yuheng Jia, and Junhui Hou. Attention-driven graph clustering network. In *ACM Multimedia*, pages 935–943. ACM, 2021. 13, 208, 219, 220, 221, 224, 226, 231, 232

[PLM+22] Namyong Park, Fuchen Liu, Purvanshi Mehta, Dana Cristofor, Christos Faloutsos, and Yuxiao Dong. EvoKG: Jointly modeling event time and network structure for reasoning over temporal knowledge graphs. In *WSDM*, pages 794–803. ACM, 2022. doi:10.1145/3488560.3498451 187, 227

[PMK16] Ha-Myung Park, Sung-Hyon Myaeng, and U. Kang. Pte: Enumerating trillion triangles on distributed systems. In *KDD*, pages 1115–1124, 2016. 99

[POK17] Namyong Park, Sejoon Oh, and U Kang. Fast and scalable distributed boolean tensor factorization. In *ICDE*, pages 1071–1082. IEEE Computer Society, 2017. doi:10.1109/ICDE.2017.152 60, 87, 97, 226

[POK19] Namyong Park, Sejoon Oh, and U Kang. Fast and scalable method for distributed boolean tensor factorization. *VLDB J.*, 28(4):549–574, 2019. doi:10.1007/s00778-019-00538-z 60, 87, 203, 226

[POvdO+19] Ben Poole, Sherjil Ozair, Aäron van den Oord, Alex Alemi, and George Tucker. On variational bounds of mutual information. In *ICML*, volume 97 of *Proceedings of Machine Learning Research*, pages 5171–5180. PMLR, 2019. 211

[PPMK16] Ha-Myung Park, Namyong Park, Sung-Hyon Myaeng, and U Kang. Partition aware connected component computation in distributed systems. In *ICDM*, pages 420–429. IEEE Computer Society, 2016. doi:10.1109/ICDM.2016.0053 99, 226

[PPMK20] Ha-Myung Park, Namyong Park, Sung-Hyon Myaeng, and U Kang. PACC: Large scale connected component computation on hadoop and spark. *PLOS ONE*, 15(3):1–25, 03 2020. 226

[PRK+22] Namyong Park, Ryan Rossi, Eunyee Koh, Iftikhar Ahamath Burhanuddin, Sungchul Kim, Fan Du, Nesreen Ahmed, and Christos Faloutsos. CGC: Contrastive graph clustering for community detection and tracking. In *WWW*. ACM / IW3C2, 2022. doi:10.1145/3485447.3512160 207

[Prone] Yahoo Webscope Program. https://webscope.sandbox.yahoo.com, 2021 [Online]. Accessed: 2021-10-01. 219

[PXF+16] Xi Peng, Shijie Xiao, Jiashi Feng, Wei-Yun Yau, and Zhang Yi. Deep subspace clustering with sparsity prior. In *IJCAI*, pages 1925–1931. IJCAI/AAAI Press, 2016. 225

[RA15] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. 142

[RAK18] Ryan A. Rossi, Nesreen K. Ahmed, and Eunyee Koh. Higher-order network representation learning. In *WWW (Companion Volume)*, pages 3–4. ACM, 2018. 148, 154

[RB08] Martin Rosvall and Carl T Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4):1118–1123, 2008. 226

[RB11] Martin Rosvall and Carl T Bergstrom. Multilevel compression of random walks on networks reveals hierarchical organization in large integrated systems. *PloS one*, 6(4):e18209, 2011. 226

[RCF+20] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael M. Bronstein. Temporal graph networks for deep learning on dynamic graphs. *CoRR*, abs/2006.10637, 2020. 204

[RZA20] Ryan A. Rossi, Rong Zhou, and Nesreen K. Ahmed. Deep inductive graph representation learning. *IEEE Trans. Knowl. Data Eng.*, 32(3):438–452, 2020. 148, 154

[SBG20] Oleksandr Shchur, Marin Bilos, and Stephan Günnemann. Intensity-free learning of temporal point processes. In *ICLR*. OpenReview.net, 2020. 204

[SBMJ09] Kunwar P Singh, Ankita Basant, Amrita Malik, and Gunja Jain. Artificial neural network modeling of the river water quality–a case study. *Ecological Modelling*, 220(6):888–895, 2009. 183

[SCMN13] Richard Socher, Danqi Chen, Christopher D. Manning, and Andrew Y. Ng. Reasoning with neural tensor networks for knowledge base completion. In *NIPS*, pages 926–934, 2013. 204

[SDNT19] Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. RotatE: Knowledge graph embedding by relational rotation in complex space. In *ICLR (Poster)*. OpenReview.net, 2019. 188, 199, 204

[SDVB18] Youngjoo Seo, Michaël Defferrard, Pierre Vandergheynst, and Xavier Bresson. Structured sequence modeling with graph convolutional recurrent networks. In *ICONIP (1)*, volume 11301 of *Lecture Notes in Computer Science*, pages 362–373. Springer, 2018. 199, 204

[SFPY07] Jimeng Sun, Christos Faloutsos, Spiros Papadimitriou, and Philip S. Yu. Graphscope: parameter-free mining of large time-evolving graphs. In *KDD*, pages 687–696. ACM, 2007. 226

[SGR19] Uriel Singer, Ido Guy, and Kira Radinsky. Node embedding over temporal graphs. In *IJCAI*, pages 4605–4612. ijcai.org, 2019. 199, 204

[Sin21] Uriel Singer. Ctdne. https://github.com/urielsinger/CTDNE, 2021. 232

[SJK15] Lee Sael, Inah Jeon, and U Kang. Scalable tensor mining. *Big Data Research*, 2(2):82 – 86, 2015. Visions on Big Data. 97

[SK16] Shaden Smith and George Karypis. A medium-grained algorithm for distributed sparse tensor factorization. In *IPDPS*, 2016. 98

[SK17] Shaden Smith and George Karypis. Accelerating the tucker decomposition with compressed sparse tensors. In *Europar*, 2017. 98

[SKB+18] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph

convolutional networks. In *ESWC*, volume 10843 of *Lecture Notes in Computer Science*, pages 593–607. Springer, 2018. 188, 193, 199, 204, 205

[SKW07] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A core of semantic knowledge. In *WWW*, pages 697–706, 2007. 3, 17

[SL09] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *science*, 324(5923):81–85, 2009. 183

[sl21] scikit learn. scikit-learn. https://github.com/scikit-learn/scikit-learn, 2021. Accessed: 2021-10-20. 231

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, pages 2960–2968, 2012. 137, 152

[SLZ20] Ke Sun, Zhouchen Lin, and Zhanxing Zhu. Multi-stage self-supervised learning for graph convolutional networks on graphs with few labeled nodes. In *AAAI*, pages 5892–5899. AAAI Press, 2020. 226

[SPK16] Shaden Smith, Jongsoo Park, and George Karypis. An exploration of optimization algorithms for high performance tensor completion. *SC*, 2016. 97

[SSK17] Kijung Shin, Lee Sael, and U. Kang. Fully scalable methods for distributed tensor factorization. *TKDE*, 29(1), 2017. 97, 98

[STM+05] Aravind Subramanian, Pablo Tamayo, Vamsi K Mootha, Sayan Mukherjee, Benjamin L Ebert, Michael A Gillette, Amanda Paulovich, Scott L Pomeroy, Todd R Golub, Eric S Lander, et al. Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles. *Proceedings of the National Academy of Sciences*, 102(43):15545–15550, 2005. 45

[STZ+20] Chang Su, Jie Tong, Yongjun Zhu, Peng Cui, and Fei Wang. Network embedding in biomedical data science. *Briefings Bioinform.*, 21(1):182–197, 2020. 7, 136

[SWG+20] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. DySAT: Deep neural representation learning on dynamic graphs via self-attention networks. In *WSDM*, pages 519–527. ACM, 2020. 204

[TDWS17] Rakshit Trivedi, Hanjun Dai, Yichen Wang, and Le Song. Know-Evolve: Deep temporal reasoning for dynamic knowledge graphs. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, pages 3462–3471. PMLR, 2017. 188, 190, 192, 196, 197, 198, 199, 204, 205

[TF06] Hanghang Tong and Christos Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *KDD*, pages 404–413, 2006. 75, 76, 79, 83, 84, 85

[TFBZ19] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. DyRep: Learning representations over dynamic graphs. In *ICLR (Poster)*. OpenReview.net, 2019. 188, 199, 204

[TFP08] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. Random walk with restart: fast solutions and applications. *Knowl. Inf. Syst.*, 14(3):327–346, 2008. 34, 35, 42, 59, 82

[TGC+14] Fei Tian, Bin Gao, Qing Cui, Enhong Chen, and Tie-Yan Liu. Learning deep representations for graph clustering. In *AAAI*, pages 1293–1299. AAAI Press, 2014. 13, 208, 226

[Tin07] Nava Tintarev. Explanations of recommendations. In *RecSys*, pages 203–206. ACM, 2007. 82

[TM07] Nava Tintarev and Judith Masthoff. A survey of explanations in recommender systems. In *ICDE*, pages 801–810, 2007. 5, 64

[TM15] Nava Tintarev and Judith Masthoff. Explaining recommendations: Design and evaluation. In *Recommender Systems Handbook*, pages 353–382. 2015. 5, 64, 82

[TMC+19] Ke Tu, Jianxin Ma, Peng Cui, Jian Pei, and Wenwu Zhu. Autone: Hyperparameter optimization for massive network embedding. In *KDD*, pages 216–225. ACM, 2019. 152

[TPF+10] Hanghang Tong, Spiros Papadimitriou, Christos Faloutsos, Philip S. Yu, and Tina Eliassi-Rad. BASSET: scalable gateway finder in large graphs. In *PAKDD*, 2010. 75, 76, 83

[TPF+12] Hanghang Tong, Spiros Papadimitriou, Christos Faloutsos, Philip S. Yu, and Tina Eliassi-Rad. Gateway finder in large graphs: problem definitions and fast solutions. *Inf. Retr.*, 15(3-4):391–411, 2012. 83

[TPPM20] Anton Tsitsulin, John Palowitch, Bryan Perozzi, and Emmanuel Müller. Graph clustering with graph neural networks. *arXiv preprint arXiv:2006.16904*, 2020. 226

[TQW+15] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. LINE: large-scale information network embedding. In *WWW*, pages 1067–1077. ACM, 2015. 154, 203

[TZY+08] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. Arnetminer: extraction and mining of academic social networks. In *KDD*, 2008. 78

[VCC+18] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *ICLR (Poster)*. OpenReview.net, 2018. 19, 22, 29, 35, 39, 44, 56, 60, 203

[vdOLV18] Aäron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *CoRR*, abs/1807.03748, 2018. 209, 211

[VFH+19] Petar Velickovic, William Fedus, William L. Hamilton, Pietro Liò, Yoshua Bengio, and R. Devon Hjelm. Deep graph infomax. In *ICLR (Poster)*. OpenReview.net, 2019. 211, 219, 220, 221, 224, 226

[VGT15] Renu Vyas, Purva Goel, and Sanjeev S. Tambe. Genetic programming applications in chemical sciences and engineering. In *Handbook of Genetic Programming Applications*, pages 99–140. 2015. 162

[Vru16] Jasper A Vrugt. Markov chain monte carlo simulation using the dream software package: Theory, concepts, and matlab implementation. *Environmental Modelling & Software*, 75:273–316, 2016. 179

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pages 6000–6010, 2017. 35

[VSR09] Jesse Vig, Shilad Sen, and John Riedl. Tagsplanations: explaining recommendations using tags. In *IUI*, pages 47–56, 2009. 5, 64, 82

[VTBC+08] Jasper A Vrugt, Cajo JF Ter Braak, Martyn P Clark, James M Hyman, and Bruce A Robinson. Treatment of input uncertainty in hydrologic modeling: Doing hydrology backward with markov chain monte carlo simulation. *Water Resources Research*, 44(12), 2008. 179

[WCG+15] Yichen Wang, Robert Chen, Joydeep Ghosh, Joshua C. Denny, Abel N. Kho, You Chen, Bradley A. Malin, and Jimeng Sun. Rubik: Knowledge guided tensor factorization and completion for health data analytics. In *KDD*, pages 1265–1274, 2015. 80, 84

[WCY+18] Xiting Wang, Yiru Chen, Jie Yang, Le Wu, Zhengtao Wu, and Xing Xie. A reinforcement learning framework for explainable recommendation. In *ICDM*, 2018. 82

[WCZ+19] Jia Wu, Xiu-Yun Chen, Hao Zhang, Li-Dong Xiong, Hang Lei, and Si-Hao Deng. Hyperparameter optimization for machine learning models based on bayesian optimization. *Journal of Electronic Science and Technology*, 17(1):26–40, 2019. 137, 152

[WGM+14] Robert West, Evgeniy Gabrilovich, Kevin Murphy, Shaohua Sun, Rahul Gupta, and Dekang Lin. Knowledge base completion via search-based question answering. In *WWW*, pages 515–526, 2014. 18

[WHF+18] Xiang Wang, Xiangnan He, Fuli Feng, Liqiang Nie, and Tat-Seng Chua. TEM: tree-enhanced embedding model for explainable recommendation. In *WWW*, 2018. 82

[Whi96] Peter Alexander Whigham. *Grammatical Bias for Evolutionary Learning*. PhD thesis, New South Wales, Australia, Australia, 1996. AAI0597571. 163

[WJS+19] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S. Yu. Heterogeneous graph attention network. In *WWW*, pages 2022–2032. ACM, 2019. 204

[WJZ+19] Felix Wu, Amauri H. Souza Jr., Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. Simplifying graph convolutional networks. In *ICML*, volume 97 of *Proceedings of Machine Learning Research*, pages 6861–6871. PMLR, 2019. 154

[WLHS21] Xiao Wang, Nian Liu, Hui Han, and Chuan Shi. Self-supervised heterogeneous graph neural network with co-contrastive learning. In *KDD*, pages 1726–1736. ACM, 2021. 226

[WM97] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Trans. Evol. Comput.*, 1(1):67–82, 1997. 8, 137

[WMP+14] Marek S. Wiewiórka, Antonio Messina, Alicja Pacholewska, Sergio Maffioletti, Piotr Gawrysiak, and Michal J. Okoniewski. Sparkseq: fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision. *Bioinformatics*, 30(18), 2014. 99

[WMWG17] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE TKDE*, 29(12), 2017. 47

[WPB01] Geoffrey I. Webb, Michael J. Pazzani, and Daniel Billsus. Machine learning for user modeling. *User Model. User-Adapt. Interact.*, 11(1-2), 2001. 9, 161

[WPH+19] Chun Wang, Shirui Pan, Ruiqi Hu, Guodong Long, Jing Jiang, and Chengqi Zhang. Attributed graph clustering: A deep attentional embedding approach. In *IJCAI*, pages 3670–3676. ijcai.org, 2019. 13, 208, 219, 220, 226

[WWY15] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. Collaborative deep learning for recommender systems. In *KDD*, pages 1235–1244, 2015. 64

[WZG+19] Qitian Wu, Hengrui Zhang, Xiaofeng Gao, Peng He, Paul Weng, Han Gao, and Guihai Chen. Dual graph attention networks for deep latent representation of multifaceted social effects in recommender systems. In *WWW*, 2019. 60

[XGF16] Junyuan Xie, Ross B. Girshick, and Ali Farhadi. Unsupervised deep embedding for clustering analysis. In *ICML*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 478–487. JMLR.org, 2016. 13, 208, 209, 219, 220, 225, 226

[XHS+12] Lin Xu, Frank Hutter, Jonathan Shen, Holger H Hoos, and Kevin Leyton-Brown. Satzilla2012: Improved algorithm selection based on cost-sensitive classification models. *Proceedings of SAT Challenge*, pages 57–58, 2012. 147, 152

[XLT+18] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. In *ICML*, pages 5449–5458, 2018. 22, 60

[XNA+19] Chengjin Xu, Mojtaba Nayyeri, Fouad Alkhoury, Jens Lehmann, and Hamed Shariat Yazdi. Temporal knowledge graph embedding model based on additive time series decomposition. *CoRR*, abs/1911.07893, 2019. 198

[XRK+20] Da Xu, Chuanwei Ruan, Evren Körpeoglu, Sushant Kumar, and Kannan Achan. Inductive representation learning on temporal graphs. In *ICLR*. OpenReview.net, 2020. 204

[XSY+21] Feng Xia, Ke Sun, Shuo Yu, Abdul Aziz, Liangtian Wan, Shirui Pan, and Huan Liu. Graph learning: A survey. *IEEE Trans. Artif. Intell.*, 2(2):109–127, 2021. 7, 136, 152

[YCWS20] Yuning You, Tianlong Chen, Zhangyang Wang, and Yang Shen. When does self-supervision help graph convolutional networks? In *ICML*, volume 119 of *Proceedings of Machine Learning Research*, pages 10871–10880. PMLR, 2020. 226

[YFSH17] Bo Yang, Xiao Fu, Nicholas D. Sidiropoulos, and Mingyi Hong. Towards k-means-friendly spaces: Simultaneous deep learning and clustering. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, pages 3861–3870. PMLR, 2017. 13, 208, 225

[YHC+18] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *KDD*, pages 974–983, 2018. 19, 22, 35, 44, 60

[YHY08] Xiaoxin Yin, Jiawei Han, and Philip S. Yu. Truth discovery with multiple conflicting information providers on the web. *IEEE TKDE*, 20(6):796–808, 2008. 59

[YLY+19] Mingxuan Yue, Yaguang Li, Haoze Yang, Ritesh Ahuja, Yao-Yi Chiang, and Cyrus Shahabi. DETECT: deep trajectory clustering for mobility-behavior analysis. In *IEEE BigData*, pages 988–997. IEEE, 2019. 13, 208

[YWCP21] Yingfang Yuan, Wenjun Wang, George M. Coghill, and Wei Pang. A novel genetic algorithm with hierarchical evaluation strategy for hyperparame-

ter optimisation of graph neural networks. *CoRR*, abs/2101.09300, 2021. 152

[YYH+15] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. In *ICLR*, 2015. 188, 199, 204

[YZ20] Tong Yu and Hong Zhu. Hyper-parameter optimization: A review of algorithms and applications. *CoRR*, abs/2003.05689, 2020. 151

[YZZ+17] Di Yao, Chao Zhang, Zhihua Zhu, Jian-Hui Huang, and Jingping Bi. Trajectory clustering via deep representation learning. In *IJCNN*, pages 3880–3887. IEEE, 2017. 13, 208

[ZC20] Yongfeng Zhang and Xu Chen. Explainable recommendation: A survey and new perspectives. *Found. Trends Inf. Retr.*, 14(1), 2020. 64, 82

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012. 89, 99, 113

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010. 99

[ZCP+18] Ziwei Zhang, Peng Cui, Jian Pei, Xiao Wang, and Wenwu Zhu. TIMERS: error-bounded SVD restart on dynamic networks. In *AAAI*, pages 224–231. AAAI Press, 2018. 219, 221, 224, 227

[ZCZ22] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *IEEE Trans. Knowl. Data Eng.*, 34(1):249–270, 2022. 7, 136, 152

[ZGY+16] Linhong Zhu, Dong Guo, Junming Yin, Greg Ver Steeg, and Aram Galstyan. Scalable temporal latent space inference for link prediction in dynamic social networks. *IEEE Trans. Knowl. Data Eng.*, 28(10):2765–2777, 2016. 204

[ZLZ+14] Yongfeng Zhang, Guokun Lai, Min Zhang, Yi Zhang, Yiqun Liu, and Shaoping Ma. Explicit factor models for explainable recommendation based on phrase-level sentiment analysis. In *SIGIR*, pages 83–92, 2014. 82

[ZMU+16] Reza Bosagh Zadeh, Xiangrui Meng, Alexander Ulanov, Burak Yavuz, Li Pu, Shivaram Venkataraman, Evan R. Sparks, Aaron Staple, and Matei Zaharia. Matrix computations and optimization in apache spark. In *KDD*, pages 31–38, 2016. 99

[ZPW+19] Wen Zhang, Bibek Paudel, Liang Wang, Jiaoyan Chen, Hai Zhu, Wei Zhang, Abraham Bernstein, and Huajun Chen. Iteratively learning em-

beddings and rules for knowledge graph reasoning. In *WWW*, pages 2366–2377, 2019. 60

[ZTLL21] Ronghang Zhu, Zhiqiang Tao, Yaliang Li, and Sheng Li. Automated graph learning via population based self-tuning GCN. In *SIGIR*, pages 2096–2100. ACM, 2021. 152

[ZWFM06] Sheng Zhang, Weihong Wang, James Ford, and Fillia Makedon. Learning from incomplete ratings using non-negative matrix factorization. In *SDM*, 2006. 64

[ZYL+16] Fuzheng Zhang, Nicholas Jing Yuan, Defu Lian, Xing Xie, and Wei-Ying Ma. Collaborative knowledge base embedding for recommender systems. In *KDD*, 2016. 3, 17

[ZYR+18] Le-kui Zhou, Yang Yang, Xiang Ren, Fei Wu, and Yueting Zhuang. Dynamic network embedding by modeling triadic closure process. In *AAAI*, pages 571–578. AAAI Press, 2018. 204