Level Aware Bootstrapping Placement for Fully Homomorphic Encryption Using MaxSAT

William Seo

CMU-CS-24-134

August 2024

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

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Keywords: Fully Homomorphic Encryption, Automatic Bootstrapping Placement, Maximum Satisfiability, Graph Reduction

To my family.

Abstract

Fully Homomorphic Encryption (FHE) is a cryptographic technique that allows computations to be performed on encrypted data without having to decrypt it. This property preserves data privacy, enabling a wide range of applications in fields such as cloud computing, secure data analysis, and privacy-preserving machine learning.

In FHE computations, each ciphertext can only handle a limited number of operations before the accumulation of noise makes decryption impossible. To resolve this, the bootstrapping operation must be used to reset the noise in the ciphertext. Bootstrapping is a computationally expensive computation, and it also influences the costs of other operations. Consequently, the strategic placement of bootstrapping operations is a critical aspect of FHE performance.

This thesis introduces Saturn, a novel method for automatically determining the optimal placement of bootstrapping operations to minimize program runtime. Given a directed acyclic graph (DAG) representing an FHE computation, Saturn leverages the Maximum Satisfiability (MaxSAT) optimization problem to find the most efficient bootstrapping placement. A key innovation in Saturn is the introduction of two methods for reducing the complexity of the input computational DAG: Quadratic Behavior Profile (QBP) Reduction and Auto-Compression. These methods significantly decrease the solve time of our MaxSAT formulation by simplifying the DAG while preserving the optimality of the solution. Saturn's effectiveness is evaluated on various deep learning models, demonstrating its potential to enhance FHE performance through efficient bootstrapping placement.

Acknowledgments

I would like to express my deepest gratitude to those who have supported me throughout my research journey at Carnegie Mellon University.

First and foremost, I extend my heartfelt thanks to my advisor, Wenting Zheng. Wenting's invaluable support, and insightful guidance have been crucial throughout this project and all my prior research endeavors. Her mentorship has been instrumental in shaping my academic growth and success.

I would also like to thank my thesis committee member, Fraser Brown, for her valuable feedback and support throughout this project.

I am grateful to Alex for his guidance, particularly in the writing of the correctness proofs for my various ideas.

A special thanks to my mentor, Edward. Your assistance with all my research projects at CMU has been immensely helpful. Edward was the one to first introduce me to the concept of optimal bootstrapping placement, and this project would not have existed without his influence and encouragement.

Finally, I would also like to extend my appreciation to my friends and family for their unwavering support. Your encouragement and understanding have been a source of motivation throughout this journey.

Contents

1 Introduction **1**

2 Background **3**

2.1 RNS-CKKS Background . . . 3

2.1.1 Scale and Level . . . 3

2.1.2 Operations . . . 4

2.1.3 The Naive Rescaling Assumption . . . 4

2.2 Partial Weighted Maximum Satisfiability Problem . . . 4

2.2.1 Problem Formulation . . . 5

2.2.2 Solving Partial Weighted MaxSAT . . . 5

3 Motivation and Overview **7**

3.1 Prior Works . . . 7

3.2 Limitations of Prior Works . . . 7

3.3 Why Saturn? . . . 8

3.4 Saturn Framework Overview . . . 8

4 Problem Formulation **11**

4.1 Definitions . . . 11

4.2 The Level-Aware Bootstrapping Problem (LA-BTS) . . . 12

4.2.1 Inputs . . . 13

4.2.2 Output . . . 13

4.3 MaxSAT Model Generation . . . 13

4.3.1 Variables . . . 13

4.3.2 Hard Clauses . . . 14

4.3.3 Soft Clauses . . . 14

4.4 Correctness of the MaxSAT Construction . . . 15

5 DAG Reduction Methods **19**

5.1 Quadratic Behavior Profile (QBP) Reduction . . . 19

5.1.1 QBP Definition . . . 20

5.1.2 Replacement . . . 20

5.1.3 Correctness . . . 22

5.1.4 QBP Truncation . . . 27

5.2 Auto-Compression 28
5.2.1 DAG Compression and Correctness 28
5.2.2 Algorithm for Finding Compression Opportunities 34

6 Evaluation **37**
6.1 Experimental Setup 37
6.2 DAG Reduction Performance 37
6.3 QBP Truncation for ReLU and SiLU 38
6.4 Bootstrap Selection Time 39
6.5 Estimated Performance for CIFAR-10 Inference 40
6.6 Bootstrapping Counts 41

7 Conclusion **43**

Bibliography **45**

List of Figures

3.1 A diagram showcasing the Saturn framework. The yellow and blue units represent data, and the gray units represent procedures. 9

5.1 A DAG calculating the polynomial that approximates ReLU (degree 15) 20

List of Tables

6.1 The performance of the DAG reduction methods on deep learning DAGs 38
6.2 QBP size reduction from QBP truncation with various values of Δ for ReLU and SiLU 38
6.3 The bootstrap selection times (s) of DaCapo and Saturn ($\Delta = 1$) for the benchmarks. Note that a "-" represents a timeout ($\geq 600s$). 39
6.4 The bootstrap selection times (s) of DaCapo and Saturn ($\Delta = 10$) for the benchmarks. Note that a "-" represents a timeout ($\geq 300s$). 39
6.5 The simplified cost model used to estimate the inference times. The numbers are extrapolated from the cost model in [7]. 40
6.6 The estimated inference times for the benchmarks bootstrapped by DaCapo and Saturn 40
6.7 The number of bootstraps placed by DaCapo and Saturn for the benchmarks . . . 41

Chapter 1

Introduction

Fully homomorphic encryption [13] (FHE) is a cryptographic primitive that allows computations on encrypted data. Using FHE, a user can securely outsource their computation to an untrusted third party. This paradigm has many promising applications, such the analysis of sensitive medical data [3], private genome analysis [21], and privacy preserving machine learning [34]. Following Gentry's groundbreaking work on FHE [13], many additional FHE schemes have been developed [4, 6, 12]. To facilitate the development of FHE applications, several comprehensive FHE libraries [1, 15, 29] and compilers [7, 9, 24, 32] have also been created.

A major challenge in the development of FHE applications is noise management. To ensure security, noise is introduced when data is encrypted into an FHE ciphertext [25]. However, with each subsequent operation on the encrypted data, the noise increases. Once the noise exceeds a certain threshold, the ciphertext can not be decrypted accurately. FHE schemes [6] utilize a mechanism called *bootstrapping* [5], which can reset the noise level of a ciphertext. For programs with long computations, bootstrappings must be inserted to periodically reset the noise in ciphertexts. Unfortunately, bootstrapping is 2-3 orders of magnitude slower than other FHE operations, make it the main bottleneck of FHE applications. As a result, the strategic placement of bootstrapping operations is a critical aspect of FHE performance.

One common approach to bootstrapping placement is manual placement [23], where the programmer determines the specific places in a program at which to inject bootstrappings. However, for larger FHE programs, this places a great burden on the programmer and could lead to suboptimal performance. Achieving optimal bootstrapping placement is challenging due to the need to balance multiple factors at once.

- First, a sufficient number of bootstraps must be placed to ensure that at any point in the program, the noise level remains within acceptable limits.
- Second, since bootstrapping incurs significant runtime overhead, the frequency of bootstrapping operations should generally be minimized.
- Lastly, it is essential to consider the impact of bootstrap insertions on the performance of other FHE operations. The runtime of arithmetic FHE operations, such as multiplication, varies based on the amount of noise in the operands. Since bootstrap placement determines the noise at all the points in the computation, it influences the performance of all other operations that occur.

To alleviate the burden of bootstrapping placement on programmers, numerous previous works have explored methods to automatically determine or approximate the optimal bootstrapping placement [2, 7, 28, 33].

This work introduces Saturn (**SAT U**sing DAG **R**eductio**N**), a novel method for finding the optimal bootstrapping placement of any FHE application. Given an input FHE program represented as a directed acyclic graph (DAG), Saturn aims to identify the bootstrapping placement that minimizes the total runtime of the FHE program. This is achieved by solving an instance of the Partial Weighted Maximum Satisfiability (Weighted MaxSAT) problem, formulated based on the structure of the input DAG. The CGSS2 solver [19] from MaxSAT Evaluation 2023 [20] is used to solve Saturn's MaxSAT formulation. Additionally, Saturn applies two methods of DAG size reduction (QBP Reduction 5.1 and Auto-Compression 5.2) prior to the MaxSAT formulation to reduce the MaxSAT solve times.

While the Saturn method can be applied to any FHE scheme, we choose to validate it using RNS-CKKS [6] due to its support for fixed-point arithmetic and SIMD operations, making it particularly well-suited for privacy-preserving machine learning models. We evaluate Saturn with five deep learning models (ResNet [16], AlexNet [22], VGG16 [31], SqueezeNet [18], and MobileNet [17]). On average Saturn achieves a $5.25\times$ speedup in bootstrapping placement selection time compared to DaCapo [7], the state-of-the-art automatic bootstrapping method. Additionally, the benchmarks generally have faster estimated inference time for a CIFAR-10 image, when bootstrapped by Saturn versus when bootstrapped by DaCapo [7].

The main contributions of this work are as follows:

- Development of a Partial Weighted Maximum Satisfiability model for the optimal bootstrapping placement problem, which aims to minimize the total FHE program runtime.
- Introduction of two DAG reduction techniques that preserve the optimality of the solution while significantly reducing problem complexity.
- Demonstration of the practicality and effectiveness of Saturn in optimizing bootstrapping placements for real-world FHE applications.

Chapter 2

Background

The first section of this chapter presents the necessary background on RNS-CKKS [6], the FHE scheme utilized for evaluating the Saturn bootstrapping method. We use RNS-CKKS because its support for fixed-point arithmetic and SIMD operations makes it particularly well-suited for privacy-preserving machine learning models, which is one of the most challenging FHE application archetypes to manually bootstrap. The second section provides an overview the Partial Weighted MaxSAT problem, which is at the core of the Saturn method.

2.1 RNS-CKKS Background

In RNS-CKKS, plaintexts are polynomial encodings of real number vectors. The encryption process involves altering the plaintext polynomial with a secret key to obtain a pair of polynomials, which is the ciphertext. During this encryption process, noise is added to the ciphertext for security. The security of RNS-CKSS relies on the Ring Learning with Errors problem [26].

2.1.1 Scale and Level

Each RNS-CKKS has a *scale* and a *level*. The scheme encodes real numbers as an integer over a scale. For instance, the real number $x = 2.1$ is encoded as the integer $v = 21$ over the scale $m = 10^1$. Multiplication results in the accumulation of scales. For example, consider the following computation:

- $x = 2.1$ is expressed with $v = 21$, $m = 10^1$
- $y = 3.0$ is expressed with $v = 30$, $m = 10^1$
- $x \times y = 6.30$ is expressed with $v = 630$, $m = 10^2$

There exists a maximum scale capacity which is determined by encryption parameters. If a ciphertext's *scale* exceeds the maximum capacity, it can no longer be decrypted correctly. Thus, FHE programs must periodically apply the *rescale* operation to reset the *scale* of a ciphertext. To *rescale* a ciphertext C, we must consume one of C's scale factors. Each ciphertext has a limited number of scale factors, and the *level* of a ciphertext denotes the number of scale factors it has available. If a ciphertext's *level* reaches 0 (ie. it runs out of scale factors), the *bootstrap* operation can used to refresh the ciphertext's *level*. The concept of ciphertext *scale* and *level* can be summarized as follows:

- Each ciphertext starts with a default *scale* and *level*.
- Multiplication causes the accumulation of *scale*.
- The *rescale* operation consumes one *level*, but resets the ciphertext's *scale*.
- The *bootstrap* operation resets the ciphertext's *level*.

2.1.2 Operations

Below are the computational operations supported in RNS-CKKS. For the descriptions below, keep in mind that each ciphertext encodes a vector of real numbers.

- **Addition**: Given two ciphertexts operands, this operation performs element-wise addition. Requires the operands to have the same *scale* and *level*.
- **Multiplication**: Given two ciphertexts operands, this operation performs element-wise multiplication. Requires the operands to have the same *level*. The scale of the result is the product of the operand scales.
- **Rotation**: Given one ciphertext operand, this operation performs a circular shift on the encoded vector.

Note how the arithmetic operations (add/mul) have *scale* and *level* requirements for its operands. For these requirements to be met, the following scale/level management operations are needed between arithmetic operations. The key challenge to optimizing FHE programs is figuring out where in the computation to insert the following operations.

- **Rescale**: Resets the *scale* of a ciphertext. The ciphertext's *level* decreases from l to $l-1$.
- **Modswitch**: Changes the ciphertext's *level* from l to $l-1$ without changing its *scale*.
- **Bootstrap**: Resets the ciphertext's *scale* and *level* back to the default values. This is the slowest operation of RNS-CKKS. It is $3774\times$ slower than addition, and $112\times$ slower than multiplication [7].

2.1.3 The Naive Rescaling Assumption

For this project, we assume that *rescales* are placed naively, specifically following each *multiplication* operation. This simplification facilitates the determination of bootstrap placement by removing the complexity associated with managing *scale*. Consequently, under this assumption, each *multiplication* consistently consumes one *level*.

The immediate next step for this research project is to lift this simplifying assumption.

2.2 Partial Weighted Maximum Satisfiability Problem

The satisfiability problem (SAT) is a fundamental problem in computer science and mathematical logic. It involves determining if there exists an assignment of truth values to variables that makes a given Boolean formula true. SAT was the first problem proven to be NP-complete, and it serves as a cornerstone for complexity theory.

The Maximum Satisfiability (MaxSAT) problem is an extension of the SAT problem. While SAT asks whether there exists an assignment that satisfies a given Boolean formula, MaxSAT seeks to find an assignment that satisfies the maximum number of clauses in a formula. This makes MaxSAT an optimization problem, where the goal is to maximize the number of satisfied clauses. The Weighted MaxSAT problem goes a step further by associating a weight with each clause. The objective in Weighted MaxSAT is to find an assignment that maximizes the sum of the weights of satisfied clauses, making it a more nuanced optimization problem.

2.2.1 Problem Formulation

Formally, a Weighted MaxSAT problem can be defined follows. Given a Boolean formula in conjunctive normal form (CNF)[1], consisting of a set of clauses $C_1, C_2, ..., C_m$ over a set of variables $X = \{x_1, x_2, ..., x_n\}$, where each clause C_i is associated with a weight w_i, the objective is to find an assignment of truth values to the variables in X that maximizes the toal weight of the satisfied clauses. The Weighted MaxSAT problem can be expressed as:

$$\text{Maximize} \sum_{i=1}^{m} w_i \cdot sat(C_i)$$

where $sat(C_i)$ is 1 if clause C_i is satisfied by the assignment and 0 otherwise.

For Saturn, we utilize a variant of this problem called Partial Weighted MaxSAT, which introduces a distinction between hard and soft clauses. Hard clauses are those that must be satisfied for a solution, whereas soft clauses can be violated, but each violation incurs a penalty based on the clause's weight. The objective in Partial Weighted MaxSAT is to maximize the sum of the weights of the satisfied soft clauses, while ensuring that all hard clauses are satisfied. This variant is particularly useful in scenarios where certain constraints are non-negotiable, while others are flexible, allowing for more realistic and practical problem-solving approaches [11, 30].

2.2.2 Solving Partial Weighted MaxSAT

Solving Partial Weighted MaxSAT is computationally challenging, as it is an NP-hard problem. A wide range of algorithms has been proposed for solving Partial Weighted MaxSAT, including branch-and-bound, local search, and SAT-based methods. Recent advancements in MaxSAT solvers have significantly improved their efficiency and scalability. Notable solvers include MaxHS [10], Open-WBO [27], and CGSS [19]. For this work, we experimented with the solvers submitted to MaxSAT Evaluation 2023 [20]. Of the solvers tested, the CGSS solver [19] had the best performance on the MaxSAT models generated by Saturn. Thus, CGSS was used for our system.

Going forward, we will have "MaxSAT" denote the weighted partial variant.

[1]A formula in CNF is a conjunction (AND) of one or more clauses, where each clause is a disjunction (OR) of one or more literals. For instance, the following is a CNF formula: $(A \lor \neg B) \land (C \lor D)$

Chapter 3

Motivation and Overview

This chapter reviews the limitations of existing methods for automatic bootstrapping placement, providing the motivation for our work: a novel method capable of identifying more optimal bootstrapping placements without sacrificing significant runtime.

3.1 Prior Works

Prior works have explored various methods to address the optimal bootstrapping problem. The earliest research in this field concentrated on minimizing the number of bootstraps inserted [2, 28]. Benhamouda et al. [2] introduced an integer linear programming (ILP) model to identify the minimal bootstrapping placement, accompanied by an approximation algorithm that rounds the proposed linear program. In FHE-Booster [33], White et al. introduced a heuristic method for minimizing the number of bootstrappings based concepts from the ILP formulation in [2].

The state-of-the-art method for optimal bootstrapping is DaCapo [7], which focuses on minimizing the total latency of the FHE program rather than merely reducing the number of bootstraps. DaCapo achieves this through a heuristic-based approach that considers the impact of bootstrapping placement on the costs of other FHE operations.

3.2 Limitations of Prior Works

Prior methods of automatic bootstrapping can be categorized into exact and approximate methods:

- Exact methods focus on a specific objective, such as minimizing the total number of bootstraps, and exhaustively search for the optimal bootstrapping plan. Though exact methods find the most optimal solution, they share the drawback of scaling poorly for large input FHE computations. The exact methods from prior works involve solving an ILP formulation [2, 28], and ran into solve times several hours long for larger computation DAGs and maximum noise budgets.
- Approximate methods, while also targeting a specific objective, employ heuristics to find a bootstrapping plan that performs well. However, these plans are not guaranteed to be

the most optimal. For instance, DaCapo's automatic bootstrapping method [7] targets the objective of minimizing the total latency of the FHE program. To reduce the search space, DaCapo splits the input program DAG into layers, and determines which layers to bootstrap rather than which vertices to bootstrap. This layer-based approach can result in a loss of precision when identifying the optimal bootstrapping points.

3.3 Why Saturn?

As detailed in the previous section, exact methods are superior in terms of solution quality, they suffer from poorly scaling solve times. At its core, Saturn is an exact method, capable of finding the most optimal solution. It scales significantly better than prior exact method for two reasons:

- First, Saturn employs two DAG reduction methods 5 which provably preserve optimally prior to finding the optimal bootstrapping points. This greatly reduces the search space.
- Second, Saturn employs a MaxSAT model to find the optimal solution, in contrast to previous works that rely on integer linear programming (ILP) formulations. Our evaluations revealed that for optimization problems that can be expressed as both MaxSAT and ILP formulations, the best MaxSAT solvers [19] can solve the problem 2-3 orders of magnitude faster than the best ILP solvers [14].

For most realistic FHE programs and noise parameters, Saturn is capable of exhaustively searching for the optimal bootstrapping plan in less than 2 minutes.

Specifically for FHE deep learning models with more than 15 calls to the SiLU activation function, an exhaustive search via MaxSAT for the optimal solution is not feasible. For such programs, Saturn can use an approximation during QBP Reduction to reduce the solve time to minutes. Evaluations demonstrate that solutions found via this approximate method differs insignificantly from the optimal.

3.4 Saturn Framework Overview

Saturn is a method for solving the Level-Aware Bootstrapping Problem for a specified FHE computation DAG and noise budget. That is, Saturn determines the bootstrapping placement that minimizes the total latency of the FHE computation while adhering to the noise budget constraints.

Figure 3.4 provides an overview of the Saturn automatic bootstrapping framework. The input to Saturn is a DAG representing the FHE computation that needs to be bootstrapped. Firstly, Saturn reduces the size of the input DAG using QBP Reduction and Auto Compression to obtain a Reduced DAG of significantly smaller size. Using the MaxSAT Model Generation procedure, Saturn generates the set of MaxSAT variables and constraints which can solves the Level-Aware Bootstrapping Problem for the Reduced DAG. A state-of-the-art MaxSAT solver [19] is then used to obtain the solution to the MaxSAT problem, which indicates the optimal way to bootstrap the Reduced DAG. Finally, the two DAG reduction methods are reverted to obtain the optimal way to bootstrap the original input DAG.

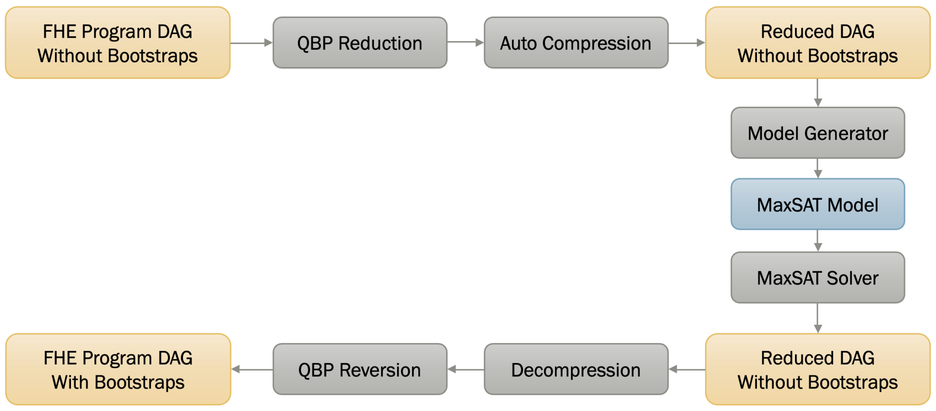
FHE Program DAG Without Bootstraps
QBP Reduction
Auto Compression
Reduced DAG Without Bootstraps
Model Generator
MaxSAT Model
MaxSAT Solver
Reduced DAG Without Bootstraps
Decompression
QBP Reversion
FHE Program DAG With Bootstraps

Figure 3.1: A diagram showcasing the Saturn framework. The yellow and blue units represent data, and the gray units represent procedures.

Chapter 4

Problem Formulation

This chapter formally defines the Level-Aware Bootstrapping Problem (LA-BTS). The goal of LA-BTS is to find the feasible bootstrapping placement which minimizes the total runtime of the FHE program. For a bootstrapping placement to be feasible, it must ensure that the ciphertext noise level never exceeds the maximum level during the computation. Additionally, this chapter describes a MaxSAT formulation that can be used to solve any instance of the LA-BTS problem.

For the following sections, let L_{max} denote the maximum ciphertext level and let $\mathbf{L}$ denote the set $\{0, 1, ..., L_{max}\}$.

4.1 Definitions

FHE computations can be represented as a direct acyclic graph (DAG) where vertices represent computational steps (operations), and edges represent the flow of data. Below is the formal definition of an FHE Computation DAG.

Definition 1 (FHE Computation DAG)**.** An FHE Computation DAG $\mathbf{D}$ consists of two parts, the vertex set $\mathbf{V}$ and the edge set $\mathbf{E}$:

- $\mathbf{V}$ is the set of vertices. Each vertex $v \in \mathbf{V}$ has four attributes:
  - $v.\mathsf{cpr} \in \mathbb{N}$ denotes the compression factor of v. This attribute is utilized in Section 5.2 to have one vertex represent a collection of vertices. For now, think of $v.\mathsf{cpr}$ as always being equal to 1.
  - $v.\mathsf{op}$ denotes the operation at v. [1]
  - $v.\mathsf{lset}$ denotes the set of possible levels vertex v can be at. For most vertices, this set is a subset of $\mathbf{L}$. Aside from a special type of vertex introduced in Section 5.1, $v.\mathsf{lset}$ is always either $\{0, 1, ..., L_{max}\}$ or $\{0, 1, ..., L_{max} - 1\}$.

    For example, if $v.\mathsf{op} =$ Mult, then $v.\mathsf{lset} = \{1, 2, ..., L_{max}\}$[2].
    If $v.\mathsf{op} =$ Add, then $v.\mathsf{lset} = \{0, 1, ..., L_{max}\}$
  - $v.\mathsf{costs} : v.\mathsf{lset} \mapsto \mathbb{N}$ is a map indicating the cost (μs) of running the operation at v at each possible level $i \in v.\mathsf{lset}$.

[1]Note that $v.\mathsf{op}$ determines $v.\mathsf{lset}$, $v.\mathsf{costs}$, and $e.\mathsf{lmap}$ for all outgoing edges.

[2]This set does not include 0 because multiplication is an operation that consumes one level.

- $\mathbf{E}$ is the set of edges. An edge is a parent/child pair of vertices. Edge $(v_p \to v_c)$ indicates that the output from the operation at v_p is used as an input for the operation at v_c.

  Each edge $e = (v_p \to v_c)$ has an attribute $e.\mathsf{lmap} : v_p.\mathsf{lset} \mapsto \mathcal{P}(v_c.\mathsf{lset})$. [3]
  For each $i \in v_p.\mathsf{lset}$, $e.\mathsf{lmap}[i]$ is the set of possible levels that v_c can have, under the condition that v_p's level is i and it is not bootstrapped. Note that $e.\mathsf{lmap}$ is determined entirely by $v_p.\mathsf{op}$.
  For instance, if $e = (v_p \to v_c)$, $L_{max} = 3$ and $v_p.\mathsf{op} = \text{Mult}$,

$$e.\mathsf{lmap} = \begin{Bmatrix} 3 : \{2,1,0\} \\ 2 : \{1,0\} \\ 1 : \{0\} \\ 0 : \{\} \end{Bmatrix}$$

  If $v_p.\mathsf{op} = \text{Add}$,

$$e.\mathsf{lmap} = \begin{Bmatrix} 3 : \{3,2,1,0\} \\ 2 : \{2,1,0\} \\ 1 : \{1,0\} \\ 0 : \{0\} \end{Bmatrix}$$

Given an FHE Computation DAG $\mathbf{D} = (\mathbf{V}, \mathbf{E})$, we can assign it a bootstrapping $B \subseteq \mathbf{V}$. B is the subset of vertices at which we place a bootstrapping. Placing a bootstrapping at vertex v means that the result of v's operation is bootstrapped before being forwarded to v's child vertices. Additionally, $\mathbf{D}$ can be assigned a levels map levels which maps each vertex $v \in \mathbf{V}$ to a level in $v.\mathsf{lset}$.

A pair of (B, levels) describes a bootstrapping plan for $\mathbf{D}$. The pair (B, levels) describes a *valid bootstrapping* if the bootstrapping placements determined by B allows each vertex $v \in \mathbf{V}$ to have level $\mathsf{levels}[v]$ with the addition of modswithces, if needed. A more formal definition is as follows:

Definition 2 (Valid Bootstrapping)**.** Let pair $P = (B, \mathsf{levels})$, where $B \subseteq \mathbf{V}$ is a bootstrapping subset and levels is a mapping of vertices to levels. $P = (B, \mathsf{levels})$ is a *valid bootstrapping* for $\mathbf{D} = (\mathbf{V}, \mathbf{E})$ if for every $(v_{parent} \to v_{child}) \in \mathbf{E}$, either

- $v_{parent} \in B$, or
- $\mathsf{levels}[v_{child}] \in (v_{parent} \to v_{child}).\mathsf{lmap}[\mathsf{levels}[v_{parent}]]$

4.2 The Level-Aware Bootstrapping Problem (LA-BTS)

This section formally defines the Level-Aware Bootstrapping Problem (LA-BTS).

[3] $\mathcal{P}(v_c.\mathsf{lset})$ denotes set of all possible subsets of $v_c.\mathsf{lset}$.

4.2.1 Inputs

The inputs to the problem are as follows:

- $L_{max} \in \mathbb{N}$, the maximum noise budget
- $\mathbf{D} = (\mathbf{V}, \mathbf{E})$, the FHE Computation DAG
- $\mathsf{c_{bts}} \in \mathbb{N}$, the cost ($\mu s$) of performing a bootstrapping

4.2.2 Output

To solve LA-BTS, we must find the *valid bootstrapping* $P = (B, \mathsf{levels})$ that minimizes the following expression:

$$\left(\mathsf{c_{bts}} \sum_{v_b \in B} v_b.\mathsf{cpr}\right) + \left(\sum_{v \in V} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]]\right)$$

Intuitively, the expression incentivises finding the bootstrapping placement that minimizes total runtime including bootstraps and computations. As this expression will be used extensively in the following sections, we set up the following notation.

Definition 3 (σ Function)**.** Let $P = (B, \mathsf{levels})$. Then,[4]

$$\sigma(P) = \left(\mathsf{c_{bts}} \sum_{v_b \in B} v_b.\mathsf{cpr}\right) + \left(\sum_{v \in V} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]]\right)$$

4.3 MaxSAT Model Generation

The LA-BTS problem can be solved by generating and solving an instance of the Partial Weighted MaxSAT problem. In this section, we begin with the set of inputs to LA-BTS described in 4.2.1 and construct the set of MaxSAT variables and clauses that allows us to find the optimal *valid bootstrapping* as described in 4.2.2.

4.3.1 Variables

For each vertex $v \in \mathbf{V}$, define the following boolean variables in the MaxSAT model:

- Level Variables: $\Gamma_v = \{\gamma_v^i | i \in v.\mathsf{lset}\}$[5]

  The truth value assignments of the variables in each Γ_v indicates the level assigned to vertex v. Using the clauses, we will ensure that in the MaxSAT solution, exactly one variable in each Γ_v is assigned truth value of $\top$. If γ_v^i is the single variable in Γv assigned $\top$, it indicates vertex v is assigned level i.

- Bootstrapping Variable: β_v

  The truth value of each β_v indicates whether or not we bootstrap at vertex v.

[4]Note that V is given by the keys of levels.

[5]Note that superscripts represent indices (not powers) throughout.

4.3.2 Hard Clauses

Hard clauses are boolean expressions that must be satisfied in any feasible solution to the MaxSAT problem.

Vertex-Based Hard Clauses: For each vertex $v \in \mathbf{V}$, we add the following hard clause to ensure that at least one boolean variable in Γ_v is assigned $\top$.

$$\bigvee_{i \in v.\mathsf{lset}} \gamma_v^i = \top$$

Later, we will use soft clauses to make sure that exactly one boolean variable in each Γ_v is assigned $\top$.

Edge-Based Hard Clauses: For each edge $e = (v_{parent} \to v_{child}) \in \mathbf{E}$, we define hard clauses to ensure that either:

- $v_{parent} \in B$, or
- $\mathsf{levels}[v_{child}] \in (v_{parent} \to v_{child}).\mathsf{lmap}[\mathsf{levels}[v_{parent}]]$

To ensure this condition, we add the following hard clause for every $i \in v_{parent}.\mathsf{lset}$:

$$\beta_{v_{parent}} \vee \neg\gamma_{v_{parent}}^i \vee \left(\bigvee_{j \in e.\mathsf{lmap}[i]} \gamma_{v_{child}}^j \right) = \top$$

4.3.3 Soft Clauses

Soft clauses are clauses that the solver tries to satisfy, but they they do not all have to be satisfied for the solution to be valid. Each soft clause has a weight, and the goal is to maximize the total weight of the satisfied soft clauses while ensuring all hard clauses are satisfied.

Bootstrapping Cost Soft Clauses: For each $v \in \mathbf{V}$, add the following soft clauses:

$$\neg\beta_v = 1 \qquad\qquad (\text{clause weight} = v.\mathsf{cpr} \cdot \mathsf{c_{bts}})$$

Operations Cost Soft Clause: For each $v \in \mathbf{V}$, for all $i \in v.\mathsf{lset}$, add the following soft clause:

$$\neg\gamma_v^i = 1 \qquad\qquad (\text{clause weight} = \mathsf{bigNum} + v.\mathsf{cpr} \cdot v.\mathsf{costs}[i])$$

Here, bigNum is a large constant. It is added to the clause weights to ensure that the solution to the MaxSAT problem assigns exactly one γ_v^i to value $\top$ in each Γv. Intuitively, adding this large constant greatly amplifies the penalty of assigning any γ_v^i to $\top$. Thus, the solution assigns $\top$ to just one variable in each Γv, which is the minimum required to satisfy the hard clauses.

For correctness, bigNum has the following lower bound:

$$\mathsf{bigNum} > \left(\mathsf{c_{bts}} \sum_{v \in \mathbf{V}} v_b.\mathsf{cpr} \right) + \left(\sum_{v \in \mathbf{V}} v.\mathsf{cpr} \cdot \max_{i \in v.\mathsf{lset}} (v.\mathsf{costs}[i]) \right)$$

4.4 Correctness of the MaxSAT Construction

In this section, let **L-MaxSAT** denote the MaxSAT problem constructed in the previous section.

- A *valid assignment* to L-MaxSAT is an assignment of truth values to the variables of L-MaxSAT that satisfies all the hard clauses.
- A *solution* to L-MaxSAT is a *valid assignment* that maximizes the sum of satisfied soft clauses.

Lemma 1 (One Noise)**.** In the *solution* to L-MaxSAT, $\forall v \in \mathbf{V}$, the set Γ_v has exactly one variable assigned value $True$.

Proof. $\forall v \in \mathbf{V}$, we have the following hard clause:

$$\bigvee_{\gamma_v^i \in \Gamma_v} \gamma_v^i = 1$$

This ensures that every γ_v^i as at least one variable assigned value $True$.

Assume for the sake of contradiction that in the *solution* A_{opt} to L-MaxSAT, there exists a $v' \in \mathbf{V}$ such that $\gamma_{v'}^i = \top,\ \gamma_{v'}^j = \top,\ i \neq j$.

For any *valid assignment* to L-MaxSAT, the sum of the weights of the unsatisfied soft clauses is at least $(x_{\gamma=\top} \cdot \mathsf{bigNum})$, where $x_{\gamma=\top}$ is the number of γ_v^i variables assigned the value $\top$. By the assumption, we know that $x_{\gamma=1} \geq |V| + 1$ for A_{opt}. Thus, for A_{opt}, the sum of the weights of the unsatisfied soft clauses is at least

$$(|V| + 1) \cdot \mathsf{bigNum}$$

Consider $A_{alternate}$, an alternate *valid assignment* to L-MaxSAT where $\forall v$, β_v is assigned $\top$, and a single arbitrary variable $\gamma_v^{l_v}$ in each Γ_v is assigned value $\top$. This is the equivalent of bootstrapping at every vertex. $A_{alternate}$ is a *valid assignment*, as all the hard clauses are satisfied. The sum of the weights of the unsatisfied soft clauses is

$$S_{alternate} = \left(\mathsf{c_{bts}} \sum_{v \in V} v_b.\mathsf{cpr}\right) + \left(\sum_{v \in V} v.\mathsf{cpr} \cdot v.\mathsf{costs}[l_v]\right) + |V| \cdot \mathsf{bigNum}$$

Note that $S_{alternate} < (|V| + 1) \cdot \mathsf{bigNum}$ by the definition of bigNum. Thus, $A_{alternate}$ has a smaller unsatisfied soft clause weight sum than the *solution* P_{opt}, which is supposed to be the optimal *valid assignment*. We have a contradiction, as we found a *valid assignment* better than the *solution*.

By contradiction, we conclude that in the *solution* to L-MaxSAT, every Γ_v has exactly one variable assigned $True$. □

Lemma 2. (MaxSAT to Bootstrapping)

Let M be a *valid assignment* to L-MaxSAT which the property that every Γ_v has exactly one variable assigned to 1.

Let function $\mathsf{convert}()$ take in M and convert it to a bootstrapping plan $P = (B, \mathsf{levels})$. That is, $\mathsf{convert}(M) = (B, \mathsf{levels})$, where

- $B = \{v \text{ for each } \beta_v \text{ assigned } \top \text{ in } M\}$
- levels is mapping of vertices to levels such that $\mathsf{levels}[v] = i$, where γ_v^i is the single variable in Γ_v assigned $\top$.

Given this,

$$(B, \mathsf{levels}) = \mathsf{convert}(M) \text{ describes a valid bootstrapping of } \mathbf{D}.$$

Proof. Let M be a *valid assignment* to L-MaxSAT which the property that every Γ_v has exactly one variable assigned to 1. Also, let $(B, \mathsf{levels}) = \mathsf{convert}(M)$.

Suppose $e = (v_{parent} \to v_{child})$ is an arbitrary edge in $\mathbf{E}$. M being a *valid assignment* means that every hard clause is satisfied. Thus, for all $i \in v_{parent}.\mathsf{lset}$,

$$\beta_{v_{parent}} \vee \neg\gamma^i_{v_{parent}} \vee \left(\bigvee_{j \in e.\mathsf{lmap}[i]} \gamma^j_{v_{child}} \right) = \top$$

Suppose $k = \mathsf{levels}[v_{parent}]$. Then $\neg\gamma^k_{v_{parent}} = 0$. Thus, we have that

$$\beta_{v_{parent}} \vee \left(\bigvee_{j \in e.\mathsf{lmap}[\mathsf{levels}[v_{parent}]]} \gamma^j_{v_{child}} \right) = \top$$

Therefore, either:

- $\beta_{v_{parent}} = \top \iff v_{parent} \in B$, or
- $\bigvee_{j \in e.\mathsf{lmap}[k]} \gamma^j_{v_{child}} \iff \mathsf{levels}[v_{child}] \in e.\mathsf{lmap}[k]$

In conclusion, (B, levels) describes a valid bootstrapping of $\mathbf{D}$. □

Lemma 3. (Bootstrapping to Variable Assignment) Given $P = (B, \mathsf{levels})$, which describes a *valid bootstrapping*, we can construct a *valid assignment* M to L-MaxSAT such that the weight sum of the unsatisfied soft clauses is:

$$\mathsf{c_{bts}} \sum_{v_b \in B} v_b.\mathsf{cpr} + \sum_{v \in V} (\mathsf{bigNum} + v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]])$$

Proof. For every v_B in B, assign truth value $\top$ to variable β_{v_B}. For each vertex $v \in \mathbf{V}$, assign truth value $\top$ to variable $\gamma_v^{\mathsf{levels}[v]}$. For every other variable assign truth value $\bot$. This completes our variable assignment M.

- Every vertex-based hard clause (4.3.2) is satisfied since for each Γ_v, there is at least variable in the set assigned true, as $\gamma_v^{\mathsf{levels}[v]} = \top$.

  Consider the set of the edge-based hard clauses (4.3.2) for an arbitrary edge $e = (v_{parent} \to v_{child})$. Let $k = \mathsf{levels}[v_{parent}]$.

  - For each $i \in v_{parent}.\mathsf{lset}$ such that $i \neq k$, the hard clause is satisfied since $\neg\gamma^i_{v_{parent}} = \top$

$$\beta_{v_{parent}} \vee \neg\gamma^i_{v_{parent}} \vee \left(\bigvee_{j \in e.\mathsf{lmap}[i]} \gamma^j_{v_{child}} \right) = \top$$

- This leaves one more hard clause:

$$\beta_{v_{parent}} \vee \neg\gamma^{k}_{v_{parent}} \vee \left(\bigvee_{j \in e.\mathsf{lmap}[k]} \gamma^{j}_{v_{child}} \right) = \top$$

  Since B is a valid bootstrapping, we know that either:
  - $v_{parent} \in B \iff \beta_{v_{parent}} = 1$, or
  - $\mathsf{levels}[v_{child}] \in e.\mathsf{lmap}[k] \iff \bigvee_{j \in e.\mathsf{lmap}[k]} \gamma^{j}_{v_{child}}$

  Thus, this hard clause is satisfied.

In conclusion, value assignment M is a *valid assignment* to L-MaxSAT.

- For each $v_B \in B$, we the following soft clause is not satisfied:

$$\neg\beta_{v_B} = \top \qquad\qquad \text{(clause weight} = v.\mathsf{cpr} \cdot \mathsf{c_{bts}})$$

For each vertex $v \in V$, the following soft clause is not satisfied:

$$\neg\gamma^{\mathsf{levels}[v]}_{v} = \top \qquad\qquad \text{(clause weight} = \mathsf{bigNum} + v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]])$$

All other soft clauses are satisfied. This gives a unsatisfied soft clause weight sum of:

$$\mathsf{c_{bts}} \sum_{v_b \in B} v_b.\mathsf{cpr} + \sum_{v \in V} (\mathsf{bigNum} + v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]])$$

□

Theorem 1. Let M denote the *solution* to the L-MaxSAT problem.

Then $(B, \mathsf{levels}) = \mathit{convert}(M)$, is the *valid bootstrapping* which minimizes

$$\left(\mathsf{c_{bts}} \sum_{v_b \in B} v_b.\mathsf{cpr} \right) + \left(\sum_{v \in V} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]] \right)$$

Proof. Let M_{msat} be the *solution* to the L-MaxSAT problem. That is , M_{msat} is the *valid assignment* maximizes the soft clause weight sum. Let $(B_{msat}, \mathsf{levels}_{msat} = \mathsf{convert}(M_{msat})$

By Lemma 1, for each $v \in \mathbf{V}$, M_{msat} assigns exactly one γ^{i}_{v} to value $\top$. Then, by Lemma 2, $(B_{msat}, \mathsf{levels}_{msat})$ is a *valid bootstrapping* of DAG $\mathbf{D}$.

Denote weight sum of unsatisfied soft clauses for L-MaxSAT *solution* M_{msat} as S_{msat}.

$$S_{msat} = \left(\mathsf{c_{bts}} \sum_{v_b \in B_{msat}} v_b.\mathsf{cpr} \right) + \left(\sum_{v \in V} (\mathsf{bigNum} + v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}_{msat}[v]]) \right)$$

Consider the set Υ_{bts} of all valid bootstrappings of $\mathbf{D}$. Let $(B_{valid}, \mathsf{levels}_{valid})$ be an arbitrary element in Υ_{bts}. By Lemma 3, we can construct a *valid assignment* M_{valid} which has the following weight sum of unsatisfied soft clauses:

$$S_{valid} = \left(\mathsf{c_{bts}} \sum_{v_b \in B_{valid}} v_b.\mathsf{cpr}\right) + \left(\sum_{v \in V} (\mathsf{bigNum} + v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}_{valid}[v]])\right)$$

As M_{msat} is a *solution* to L-MaxSAT (ie. the most optimal *valid assignment*), and M_{valid} is an arbitrary *valid assignment*,

$$\begin{aligned} S_{msat} \leq S_{valid} \implies & \left(\mathsf{c_{bts}} \sum_{v_b \in B_{msat}} v_b.\mathsf{cpr}\right) + \left(\sum_{v \in V} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}_{msat}[v]]\right) \\ & \leq \left(\mathsf{c_{bts}} \sum_{v_b \in B_{valid}} v_b.\mathsf{cpr}\right) + \left(\sum_{v \in V} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}_{valid}[v]]\right) \end{aligned}$$

Thus, $(B_{msat}, \mathsf{levels}_{msat})$ is a valid bootstrapping that minimizes

$$\left(\mathsf{c_{bts}} \sum_{v_b \in B} v_b.\mathsf{cpr}\right) + \left(\sum_{v \in V} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]]\right)$$

for all possible $(B, \mathsf{levels}) \in \Upsilon_{bts}$ $\square$

Chapter 5

DAG Reduction Methods

In the Chapter 4, we introduced the LA-BTS, an optimization problem where we want to find the *valid bootstrapping* $P = (B, \text{levels})$ that minimizes the estimated cost of an FHE program. Additionally, we show that for any instance I of LA-BTS, we can construct MaxSAT problem that can be solved to find the solution to I. We use **L-MaxSAT** denote the MaxSAT problem constructed to solve an instance of LA-BTS.

Our end goal is solve LA-BTS for large FHE computations, such as deep neural networks. These computations have computation DAGs $\mathbf{D}$ and consequently end up with L-MaxSAT constructions with too many variables and clauses to be solved in a reasonable time. Thus, we want a way to reduce $\mathbf{D}$ to a more compact DAG $\mathbf{D}'$ which results in a L-MaxSAT construction that is much smaller and easier to solve. This chapter introduces two methods for reducing the size of an FHE Computation DAG: QBP Reduction (5.1) and Auto-Compression (5.2).

5.1 Quadratic Behavior Profile (QBP) Reduction

Quadratic Behavior Profile (QBP) Reduction method involves profiling the behavior of specific functions in an FHE computation. This profile, which has a number of entries quadratic to L_{max}, is used to represent the calls to the profiled function in a very compact manner.

The main insight is the following: given a single-input single-output (SISO) DAG $\mathbf{D}_{SISO}$, we can efficiently find the optimal bootstrapping for $\mathbf{D}_{SISO}$ under all possible combinations of input and output levels. The optimal bootstrapping plan for each IO level combination is the QBP of $\mathbf{D}_{SISO}$. With the QBP, each instance of $\mathbf{D}_{SISO}$ can be replaced with a pair of vertices that behaves the same way under all possible combinations of input and output levels.

The motivation for QBP Reduction is the complexity of activation functions such as ReLU and SiLU in an HE setting. Such activation functions are implemented using approximation polynomials. These polynomials have a high degree, so their computation process is long and complex, as shown in figure 5.1. Fortunately, activation functions are SISO, meaning that each call to them can be replaced by a pair of behaviorally equivalent vertices via the QBP reduction.

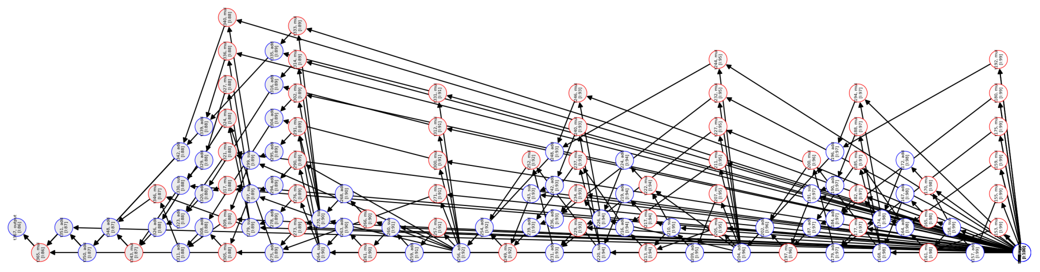

Figure 5.1: A DAG calculating the polynomial that approximates ReLU (degree 15)

5.1.1 QBP Definition

Definition 4 (Quadratic Behavior Profile)**.** Suppose $\mathbf{D} = (\mathbf{V}, \mathbf{E})$ describes a SISO DAG with input vertex v_{in} and output vertex v_{out}. The Level Aware Quadratic Behavior Profile of $\mathbf{D}$, is a nested map $\mathsf{QBProfile}_{\mathbf{D}} : v_{in}.\mathsf{lset} \to v_{out}.\mathsf{lset} \to \mathbb{N}$.

$$\mathsf{QBProfile}_{\mathbf{D}}[i][j] = \min_{(B,\mathsf{levels})\in\Upsilon_{\mathbf{D}}^{(i,j)}} \left(\mathsf{c}_{\mathsf{bts}} \sum_{v_b \in B} v_b.\mathsf{cpr} \right) + \left(\sum_{v\in V} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]] \right)$$

Here, $\Upsilon_{\mathbf{D}}^{(i,j)}$ is a subset of Υ_{bts}, the set of all *valid bootstrappings* of $\mathbf{D}$. $(B, \mathsf{levels}) \in \Upsilon_{\mathbf{D}}^{(i,j)}$ iff $\mathsf{levels}[v_{in}] = i$ and $\mathsf{levels}[v_{out}] = j$.

The value of each entry $\mathsf{QBProfile}_{\mathbf{D}}[i][j]$ in the QBP can be found by generating and solving the MaxSAT model from Section 4.3, with two additional hard constraints:

- $\gamma^{i}_{v_{in}} = \top$
- $\gamma^{j}_{v_{out}} = \top$

5.1.2 Replacement

In this section, we describe the process for replacing an instance of a SISO sub-DAG $\mathbf{d}$ with a pair of vertices, utilizing the information in $\mathsf{QBProfile}_{\mathbf{d}}$. First, we define what it means to be a sub-DAG.

Definition 5 (Sub-DAG)**.** Let $\mathbf{D} = (\mathbf{V}, \mathbf{E})$ be a computation DAG. $\mathbf{D}_A = (\mathbf{V}_A, \mathbf{E}_A)$ is a *sub-DAG* of $\mathbf{D}$ iff:

- $\mathbf{V}_A \subseteq \mathbf{V}$
- $\mathbf{E}_A \subseteq \mathbf{E}$
- $\mathbf{D}_A$ is fully connected
- All edges $e \in \mathbf{E}_A$ only involve vertices in $\mathbf{V}_A$

Suppose $\mathbf{d} = (\mathbf{V}_d, \mathbf{E}_d)$ is a *sub-DAG* of $\mathbf{D} = (\mathbf{V}, \mathbf{E})$ and we want to use QBP reduction to simplify an instance of $\mathbf{d}$ in $\mathbf{D}$ to a pair of vertices via the QBP reduction. Let v_{in} and v_{out} denote the single input and output vertices of $\mathbf{d}$. The replacement process is as follows:

1. All edges fully internal to $\mathbf{d}$ are removed from $\mathbf{E}$. That is, all edges involving a vertex in $\mathbf{d}$ except for the edges going into v_{in} and edges going out of v_{out} are removed.
2. All vertices in $\mathbf{d}$ aside from v_{in} and v_{out} are removed.
3. The attributes of v_{in} are updated as follows:
   - $v.\mathsf{cpr} = 1$
   - $v.\mathsf{op} = \mathbf{d}$-qbp-input
   - $v.\mathsf{lset}$:

     This is the most involved step of the replacement. Recall that $v.\mathsf{lset}$ is the set of the possible levels vertex v can be at. For standard vertices v, which represent intermediate ciphertexts in the FHE-computation, $v.\mathsf{lset}$ is simply a set of integers, as a ciphertext's levels range from $\{0, 1, ..., L_{max}\}$ or $\{0, 1, ..., L_{max} - 1\}$.

     After the QBP reduction, v_{in} no longer represents an intermediate ciphertext; it represents an entire sub-computation. The level of v_{in} does not represent a ciphertext noise level, but rather represents the input/output level combination for the sub-computation it represents. Thus, we have the following:

     $$v.\mathsf{lset} = \left\{\begin{matrix} (0,0), (0,1), ..., (0, L_{max}), \\ (1,0), (1,1), ..., (1, L_{max}), \\ \vdots \\ (L_{max},0), (L_{max},1), ..., (L_{max}, L_{max}) \end{matrix}\right\}$$

     Note that for nodes in $\mathbf{V} \setminus \mathbf{d}$, the original $v.\mathsf{lset}$ types are kept.
   - $v.\mathsf{costs} = \{(i,j) \mapsto \mathsf{QBProfile}_{\mathbf{d}}[i][j]$ for each $(i,j) \in v.\mathsf{lset}\}$

     The entries from the QBP are used to assign the cost of the sub-computation represented by v_{in} for the possible levels (ie. the IO level combinations).
4. The attributes of v_{out} are updated as follows:
   - $v.\mathsf{cpr} = 1$
   - $v.\mathsf{op} = \mathbf{d}$-qbp-output
   - $v.\mathsf{lset}$ is not changed.
   - $v.\mathsf{costs} = \{(i \mapsto 0$ for each $i \in v.\mathsf{lset}\}$
5. Edge $e = (v_{in} \to v_{out})$ is added.

   $$e.\mathsf{lmap} = \{(i,j) \mapsto j \text{ for each } (i,j) \in v_{in}.\mathsf{lset}\}$$

6. Each incoming edge to v_{in} is updated.

   $$(v_{parent} \to v_{in}).\mathsf{lmap} = \left\{ i \mapsto (\text{range}(i) \times \mathbf{L}) \text{ for each } i \in \mathbf{L} \right\}^{1}$$

[1]range(i) is the set of non-negative integers less than or equal to i.

5.1.3 Correctness

Suppose we have a main DAG $\mathbf{D} = (\mathbf{V}, \mathbf{E})$, which contains an instance of a SISO *sub-DAG* $\mathbf{d}$. In this section, we will prove the correctness of the QBP reduction. First, we define a couple functions needed for the proof.

Definition 6 (reduce Function)**.** This function applies the QBP reduction on a dag.

- **Input:** An instance of LA-BTS $I = (L_{max}, \mathbf{D}, \mathsf{c_{bts}})$, as well as $\mathbf{d}$, a SISO *sub-DAG* of $\mathbf{D}$.
- **Output:** A new instance $I' = (L_{max}, G', \mathsf{c_{bts}})$, where $\mathbf{D}' = (\mathbf{V}', \mathbf{E}')$ is $\mathbf{D}$ with the replacement process in subsection 5.1.2 applied.

For the remainder of this section, let (v_{in}, v_{out}) denote the input and output vertex of $\mathbf{d}$. Let (v'_{in}, v'_{out}) denote the pair of vertices used to replace $\mathbf{d}$ in $\mathbf{D}'$

Definition 7 (reconstruct Function)**.** This function takes a solution to the reduced DAG and constructs a solution to the original DAG.

- **Input:** A solution (B', levels') to I', as well as $\mathbf{d}$, a SISO *sub-DAG* of $\mathbf{D}$.
  For this definition and following proofs, we operate under the assumption that v'_{in} and v'_{out} are not in B'. Though it is possible to proceed without this assumption, it would make the reconstruct function more complex and the following proofs much lengthier.
- **Output:** The solution (B, levels) to I is constructed from (B', levels') as follows. For each $v \in \mathbf{V}$, we case based on whether or not v is a part of $\mathbf{d}$:
  - **Step 1:** We handle the bootstrapping and level assignments for $v \in \mathbf{V}$, where v is not a part of $\mathbf{d}$. For such v, there is an equivalent vertex $v' \in \mathbf{V}'$.
    - Add v to B iff $v' \in B'$ in the solution I'
    - Set $\mathsf{levels}[v]$ to be equal to $\mathsf{levels}'[v']$ in the solution I'
  - **Step 2:** We handle the bootstrapping and level assignments for the vertices v in $\mathbf{d}$. Suppose $(i, j) = \mathsf{levels}'[v'_{in}]$. Recall the process for finding the value of the entry of $\mathsf{QBProfile}_{\mathbf{D}}[i][j]$ in 5.1.1. Obtain $(B_{\mathbf{d}}, \mathsf{levels}_{\mathbf{d}})$ by using argmin instead of min.

$$(B_{\mathbf{d}}, \mathsf{levels}_{\mathbf{d}}) = \underset{(B,\mathsf{levels}) \in \Upsilon_{\mathbf{d}}^{(i,j)}}{\operatorname{argmin}} \left(\mathsf{c_{bts}} \sum_{v_b \in B} v_b.\mathsf{cpr}\right) + \left(\sum_{v \in V} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]]\right)$$

    - Add v to B iff $v \in B_d$
    - Set $\mathsf{levels}[v]$ to be equal to $\mathsf{levels}_{\mathbf{d}}[v]$

Lemma 4. If (B', levels') describes a *valid bootstrapping* for I',

$$(B, \mathsf{levels}) = \mathsf{reconstruct}(B', \mathsf{levels}', \mathbf{d})$$

describes a *valid bootstrapping* for I.

Proof. Define $(i, j) = \mathsf{levels}'[v'_{in}]$. This also means that $j = \mathsf{levels}'[v'_{out}]$ because v'_{in} is not bootstrapped. Let $e = (v_{parent} \to v_{child})$ be an arbitrary edge in $\mathbf{E}$.

- If neither v_{parent} nor v_{child} are not a part of $\mathbf{d}$, then there is an equivalent edge $e' = (v'_{parent} \to v'_{child})$ in $\mathbf{E}'$. Since (B', levels') describes a *valid bootstrapping*, either

- $v'_{parent} \in B' \implies v_{parent} \in B$, or
- $\mathsf{levels}'[v'_{child}] \in (v'_{parent} \to v'_{child}).\mathsf{lmap}[\mathsf{levels}'[v'_{parent}]]$
  $\implies \mathsf{levels}[v_{child}] \in (v_{parent} \to v_{child}).\mathsf{lmap}[\mathsf{levels}[v_{parent}]]$

- If $e = (v_{parent} \to v_{in})$:
  Consider the edge $e' = (v'_{parent} \to v'_{in})$ in $\mathbf{E}$. Since (B', levels') describes a *valid bootstrapping*, either
  - $v'_{parent} \in B' \implies v_{parent} \in B$, or
  - $\mathsf{levels}'[v'_{in}] \in (v'_{parent} \to v'_{in}).\mathsf{lmap}[\mathsf{levels}'[v'_{parent}]]$

$$\begin{aligned}&\implies (i,j) \in (v'_{parent} \to v'_{in}).\mathsf{lmap}[\mathsf{levels}'[v'_{parent}]]\\ &\implies i \in (v_{parent} \to v_{in}).\mathsf{lmap}[\mathsf{levels}[v_{parent}]] && \text{(By step 6 in 5.1.2)}\\ &\implies \mathsf{levels}[v_{in}] \in (v_{parent} \to v_{in}).\mathsf{lmap}[\mathsf{levels}[v_{parent}]]\\ &&& \text{(Via construction, } \mathsf{levels}[v_{in}] = i)\end{aligned}$$

- If e is fully internal to $\mathbf{d}$:
  The edge satisfies the *valid bootstrapping* condition, since the bootstrap and level assignments for the vertices in $\mathbf{d}$ were done according to an element of $\Upsilon_{\mathbf{d}}^{(i,j)}$, which is a set exclusively containing *valid bootstrappings*.
- If $e = (v_{out} \to v_{child})$:
  Consider the edge $e' = (v'_{out} \to v'_{child})$ in $\mathbf{E}'$. Since (B', levels') describes a *valid bootstrapping* and a bootstrap is not placed at v'_{out},

$$\begin{aligned}\mathsf{levels}'[v'_{child}] \in &(v'_{out} \to v'_{child}).\mathsf{lmap}[\mathsf{levels}'[v'_{out}]]\\ &\implies \mathsf{levels}'[v'_{child}] \in (v'_{out} \to v'_{child}).\mathsf{lmap}[j]\\ &\implies \mathsf{levels}[v_{child}] \in (v'_{out} \to v'_{child}).\mathsf{lmap}[j] && \text{(As } v_{child} \text{ is not in } \mathbf{d})\\ &\implies \mathsf{levels}[v_{child}] \in (v'_{out} \to v'_{child}).\mathsf{lmap}[\mathsf{levels}[v_{out}]]\\ &&& \text{(Via construction, } \mathsf{levels}[v_{out}] = j)\\ &\implies \mathsf{levels}[v_{child}] \in (v_{out} \to v_{child}).\mathsf{lmap}[\mathsf{levels}[v_{out}]]\\ &&& \text{(As reduce does not change the lmap of this edge)}\end{aligned}$$

□

Lemma 5. Applying the reconstruct algorithm can only reduce the σ value. That is,

$$\sigma(B', \mathsf{levels}') \geq \sigma(\mathsf{reconstruct}(B', \mathsf{levels}', \mathbf{d}))$$

Proof. Let $(B, \mathsf{levels}) = \mathsf{reconstruct}(B', \mathsf{levels}', \mathbf{d})$. We want to compare the σ values of solutions (B', levels') for I' and and (B, levels) for I. In the comparison below, let $(i,j) = (\mathsf{levels}[v_{in}], \mathsf{levels}[v_{out}])$.

Note that both solutions are identical for all vertices not in $\mathbf{d}$ so we just have to compare the σ contribution of (v'_{in}, v'_{out}) for I' and the σ contribution of sub-DAG $\mathbf{d}$ for I:

- v'_{in} and v'_{out}'s bootstrap and level assignments have the following σ contribution for I':

  From the bootstrapping placements, the σ contribution is at least 0.

From the level assignments, the σ contribution is

$$\begin{aligned} v'_{in}.\mathsf{cpr} \cdot v'_{in}.\mathsf{costs}[(i,j)] + v'_{out}.\mathsf{cpr} \cdot v'_{out}.\mathsf{costs}[j] &= 1 \cdot v'_{in}.\mathsf{costs}[(i,j)] + 0 \\ &= \mathsf{QBProfile}_{\mathbf{d}}[i][j] \\ &= \min_{(B,\mathsf{levels}) \in \Upsilon_{\mathbf{d}}^{(i,j)}} \left(\mathsf{c_{bts}} \sum_{v_b \in B} v_b.\mathsf{cpr} \right) + \left(\sum_{v \in V} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]] \right) \end{aligned}$$

- Sub-DAG $\mathbf{d}$'s bootstrap and level assignments have the following σ contribution for I:

  The σ contribution is exactly

$$\min_{(B,\mathsf{levels}) \in \Upsilon_{\mathbf{d}}^{(i,j)}} \left(\mathsf{c_{bts}} \sum_{v_b \in B} v_b.\mathsf{cpr} \right) + \left(\sum_{v \in V} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]] \right)$$

  because the **Step 2** in the reconstruct function assigns bootstraps and levels based on the argmin of the formula above.

Thus, $\sigma(B', \mathsf{levels}') \geq \sigma(\mathsf{reconstruct}(B', \mathsf{levels}', \mathbf{d}))$.

$\square$

Definition 8 (revreconstruct Function)**.** This function does the reverse of the reconstruct function. It takes a solution to the original DAG, and constructs a solution to the reduced DAG.

- **Input:** A solution (B, levels) to I, as well as $\mathbf{d}$, a SISO *sub-DAG* of $\mathbf{D}$.
  Similarly, this definition and the following proofs operate under the assumption that v_{in} and v_{out} are not in B. Though it is possible to proceed without this assumption, we do not do this to prevent the section from becoming too verbose.
- **Output:** The solution (B', levels') to I' is constructed from (B, levels) as follows:
  - **Step 1:** We handle the bootstrapping and level assignments for v'_{in} and v'_{out}, the pair of vertices used to replace $\mathbf{d}$ in $\mathbf{D}'$.
    - Neither vertex is added to B'.
    - Let v_{in} and v_{out} denote the input and output vertices to sub-DAG $\mathbf{d}$ in $\mathbf{D}$.
      - Set $\mathsf{levels}'[v'_{in}] = (\mathsf{levels}[v_{in}], \mathsf{levels}[v_{out}])$
      - Set $\mathsf{levels}'[v'_{out}] = \mathsf{levels}[v_{out}]$
  - **Step 2:** We handle the bootstrapping and level assignments for the remaining vertices v' in $\mathbf{D}'$. For such v', there is an equivalent vertex $v \in \mathbf{V}$
    - Add v' to B' iff $v \in B$
    - Set $\mathsf{levels}'[v']$ to be equal to $\mathsf{levels}[v]$

Lemma 6. If (B, levels) describes a *valid bootstrapping* for I,

$$(B', \mathsf{levels}') = \mathsf{revreconstruct}(B, \mathsf{levels}, \mathbf{d})$$

describes a *valid bootstrapping* for I'.

Proof. Define $(i, j) = (\mathsf{levels}[v_{in}], \mathsf{levels}[v_{out}])$. Let $e' = (v'_{parent} \to v'_{child})$ be an arbitrary edge in $\mathbf{E}'$.

- If neither v'_{parent} nor v'_{child} are not a part of $\{v'_{in}, v'_{out}\}$, then there is an equivalent edge $e = (v_{parent} \to v_{child})$ in $\mathbf{E}$. Since (B, levels) describes a *valid bootstrapping*, either
  - $v_{parent} \in B \implies v'_{parent} \in B'$, or
  - $\mathsf{levels}[v_{child}] \in (v_{parent} \to v_{child}).\mathsf{lmap}[\mathsf{levels}[v_{parent}]]$
    $\implies \mathsf{levels}'[v'_{child}] \in (v'_{parent} \to v'_{child}).\mathsf{lmap}[\mathsf{levels}'[v'_{parent}]]$
- If $e' = (v'_{parent} \to v'_{in})$:
  Consider the edge $e = (v_{parent} \to v_{in})$ in $\mathbf{E}$. Since (B, levels) describes a *valid bootstrapping*, either
  - $v_{parent} \in B \implies v'_{parent} \in B'$, or
  - $\mathsf{levels}[v_{in}] \in (v_{parent} \to v_{in}).\mathsf{lmap}[\mathsf{levels}[v_{parent}]]$

$$\begin{aligned}
&\implies i \in (v_{parent} \to v_{in}).\mathsf{lmap}[\mathsf{levels}[v_{parent}]] \\
&\implies (i, j) \in (v'_{parent} \to v'_{in}).\mathsf{lmap}[\mathsf{levels}'[v'_{parent}]] && \text{(By step 6 in 5.1.2)} \\
&\implies \mathsf{levels}'[v'_{in}] \in (v'_{parent} \to v'_{in}).\mathsf{lmap}[\mathsf{levels}'[v'_{parent}]] \\
& && \text{(via construction, } \mathsf{levels}'[v'_{in}] = (i, j))
\end{aligned}$$

- If $e' = (v'_{in} \to v'_{out})$: By construction, we know that:

$$j \in (v'_{in} \to v'_{out}).\mathsf{lmap}[\mathsf{levels}'[(i, j))]]$$

  Thus,

$$\mathsf{levels}'[v'_{out}] \in (v'_{in} \to v'_{out}).\mathsf{lmap}[\mathsf{levels}'[v'_{in}]]$$

- If $e' = (v'_{out} \to v'_{child})$:
  Consider the edge $e = (v_{out} \to v_{child})$ in $\mathbf{E}$. Since (B, levels) describes a *valid bootstrapping* and a bootstrap is not placed at v_{out},

$$\begin{aligned}
\mathsf{levels}[v_{child}] \in &(v_{out} \to v_{child}).\mathsf{lmap}[\mathsf{levels}[v_{out}]] \\
&\implies \mathsf{levels}[v_{child}] \in (v_{out} \to v_{child}).\mathsf{lmap}[j] \\
&\implies \mathsf{levels}'[v'_{child}] \in (v_{out} \to v_{child}).\mathsf{lmap}[j] \\
&\qquad\qquad \text{(As } v'_{child} \text{ is not a qbp vertex)} \\
&\implies \mathsf{levels}'[v'_{child}] \in (v_{out} \to v_{child}).\mathsf{lmap}[\mathsf{levels}'[v'_{out}]] \\
&\qquad\qquad \text{(Via construction)} \\
&\implies \mathsf{levels}'[v'_{child}] \in (v'_{out} \to v'_{child}).\mathsf{lmap}[\mathsf{levels}'[v'_{out}]] \\
&\qquad\qquad \text{(As reduce does not change the lmap of this edge)}
\end{aligned}$$

□

Lemma 7. Applying the revreconstruct algorithm can only reduce the σ value. That is,

$$\sigma(B, \mathsf{levels}) \geq \sigma(\mathsf{revreconstruct}(B, \mathsf{levels}, \mathbf{d}))$$

Proof. Let $(B', \mathsf{levels}') = \mathsf{revreconstruct}(B, \mathsf{levels}, \mathbf{d})$.

We want to compare the σ values of solutions (B', levels') for I' and and (B, levels) for I. In the comparison below, let $(i, j) = (\mathsf{levels}[v_{in}], \mathsf{levels}[v_{out}])$

Note that both solutions are identical for all vertices outside of sub-DAG $\mathbf{d}$ so we just have to compare the σ contribution of (v'_{in}, v'_{out}) for I' and the σ contribution of $\mathbf{d}$ for I:

- v'_{in} and v'_{out}'s bootstrap and level assignments have the following σ contribution for I':

  As neither vertex is bootstrapped, the σ contribution is

$$\begin{aligned} &v'_{in}.\mathsf{cpr}\cdot v'_{in}.\mathsf{costs}[(i,j)]] + v'_{out}.\mathsf{cpr}\cdot v'_{out}.\mathsf{costs}[j] \\ &\quad = 1 \cdot v'_{in}.\mathsf{costs}[(i,j)]] + 0 \\ &\quad = \mathsf{QBProfile}_{\mathbf{d}}[i][j] \\ &\quad = \min_{(B,\mathsf{levels})\in\Upsilon_{\mathbf{d}}^{(i,j)}} \left(\mathsf{c_{bts}} \sum_{v_b \in B} v_b.\mathsf{cpr}\right) + \left(\sum_{v\in V} v.\mathsf{cpr}\cdot v.\mathsf{costs}[\mathsf{levels}[v]]\right) \end{aligned}$$

- Sub-DAG $\mathbf{d}$'s bootstrap and level assignments have the following σ contribution for I:

  The σ contribution is at least as large as

$$\min_{(B,\mathsf{levels})\in\Upsilon_{\mathbf{d}}^{(i,j)}} \left(\mathsf{c_{bts}} \sum_{v_b \in B} v_b.\mathsf{cpr}\right) + \left(\sum_{v\in V} v.\mathsf{cpr}\cdot v.\mathsf{costs}[\mathsf{levels}[v]]\right)$$

Thus, we can conclude that $\sigma(B', \mathsf{levels}') \leq \sigma(B, \mathsf{levels}, \mathbf{d})$. □

Theorem 2. An instance $I = (L_{max}, \mathbf{D}, \mathsf{c_{bts}})$ of LA-BTS is *feasible* if there exists a *valid bootstrapping* (B, levels) for computation $\mathbf{D}$ with max noise level L_{max}.

All instances of LA-BTS I are *feasible*.

Proof. This is trivially true, as for any LA-BTS instance, a *valid bootstrapping* can be created by making B equal to the entire vertex set and setting the level map levels arbitrarily. □

Theorem 3. If (B', levels') is an optimal solution to I', then $(B, \mathsf{levels}) = \mathsf{reconstruct}(B', \mathsf{levels}', \mathbf{d})$ is an optimal solution to I.

Proof. We prove the contra-positive:

If $\exists$ some better solution $(\hat{B}, \widehat{\mathsf{levels}})$ to I, then $\exists$ a better solution $(\hat{B}', \widehat{\mathsf{levels}}')$ to I'

Suppose $(\hat{B}, \widehat{\mathsf{levels}})$ is a better solution than (B, levels) for I. That is,

$$\sigma(\hat{B}, \widehat{\mathsf{levels}}) < \sigma(B, \mathsf{levels})$$

Construct the solution $(\hat{B}', \widehat{\mathsf{levels}}')$ to I' as follows:

$$(\hat{B}', \widehat{\mathsf{levels}}') = \mathsf{revreconstruct}(\hat{B}, \widehat{\mathsf{levels}}, \mathbf{d})$$

$$\begin{aligned}
\sigma(\hat{B}', \widehat{\mathsf{levels}}') &\leq \sigma(\hat{B}, \widehat{\mathsf{levels}}) && \text{(By Lemma 7)}\\
&< \sigma(B, \mathsf{levels}) && \text{(By assumption that } (\hat{B}, \widehat{\mathsf{levels}}) \text{ is more optimal)}\\
&= \sigma(\mathsf{reconstruct}(B', \mathsf{levels}', \mathbf{d})) && \text{(By definition of } (B, \mathsf{levels}))\\
&\leq \sigma(B', \mathsf{levels}') && \text{(By Lemma 5)}
\end{aligned}$$

We have found a better solution for I' as desired. □

The combination of Theorem 2 and Theorem 3 shows that given a DAG $\mathbf{D} = (\mathbf{V}, \mathbf{E})$, which contains an instance of a SISO *sub-DAG* $\mathbf{d}$, we can solve LA-BTS instance $I = (L_{max}, \mathbf{D}, \mathsf{c}_{\mathsf{bts}})$ by:

- Obtaining new LA-BTS instance $I' = \mathsf{reduce}(I, \mathbf{d})$
- Solving I' to obtain solution (B', levels')
- Obtaining solution (B, levels) to I by computing $(B, \mathsf{levels}) = \mathsf{reconstruct}(B', \mathsf{levels}', \mathbf{d})$

5.1.4 QBP Truncation

The QBP of a *sub-DAG* $\mathbf{d}$ has the number of entries equal to the number of possible combinations of input and output levels, which is $(L_{max} + 1)^2$.

Recall how when a *sub-DAG* is reduced to a pair of vertices v'_{in} and v'_{out}, the size of $v'_{in}.\mathsf{lset}$ is equal to the number of entries in the QBP. In our evaluations, we use $L_{max} = 16$, resulting in a vertex with $17^2 = 289$ level options for each call to ReLU/SiLU. Consequently, this adds a lot of additional variables and clauses to our MaxSAT model.

To combat this we use the insight that many entries in QBPs are redundant. For example, the following may occur in the QBP of *sub-DAG* $\mathbf{d}$:

$$\mathsf{QBProfile}_{\mathbf{d}}[i][j-1] = \mathsf{QBProfile}_{\mathbf{d}}[i][j] \qquad \textbf{(i)}$$

In this case, the IO-level option of $(i, j-1)$ is strictly inferior to the option of (i, j), as it provides a lower output level at the same cost (estimated runtime).

Definition 9 (QBP Truncation)**.** Here, we describe a method for truncating redundant/inferior IO-level options from a QBP. The Δ parameter is used to control the magnitude of truncation.

Below, we define function the function $\mathsf{truncateQBP}$:

- **Input:** A QBP $\mathsf{QBProfile}_{\mathbf{d}}$ and non-negative integer Δ
- **Output:** A truncated QBP $\mathsf{QBProfile}_{\mathbf{d}}^{T_\Delta}$ is constructed as follows:
  For each input level option $i \in \mathbf{L}$:
  1. Set $\mathsf{QBProfile}_{\mathbf{d}}^{T_\Delta}[i][L_{max}] = \mathsf{QBProfile}_{\mathbf{d}}[i][L_{max}]$
  2. All QBP entries $\mathsf{QBProfile}_{\mathbf{d}}[i][j]$ such that

$$\mathsf{QBProfile}_{\mathbf{d}}[i][j] \geq \mathsf{QBProfile}_{\mathbf{d}}[i][L_{max}] - \Delta$$

  are truncated.

3. Set $\mathsf{QBProfile}^{T_\Delta}_{\mathbf{d}}[i][L_{next}] = \mathsf{QBProfile}_{\mathbf{d}}[i][L_{next}]$, where $\mathsf{QBProfile}_{\mathbf{d}}[i][L_{next}]$ is the entry with the largest cost that is less than $\mathsf{QBProfile}_{\mathbf{d}}[i][L_{max}] - \Delta$.
4. All QBP entries $\mathsf{QBProfile}_{\mathbf{d}}[i][j]$ such that

$$\mathsf{QBProfile}_{\mathbf{d}}[i][j] \geq \mathsf{QBProfile}_{\mathbf{d}}[i][L_{next}] - \Delta$$

are truncated.
5. Repeat until all the output level options have been set or truncated.

By using computing $\mathsf{QBProfile}^{T_\Delta}_{\mathbf{d}}[i][L_{next}] = \mathsf{truncateQBP}(\mathsf{QBProfile}_{\mathbf{d}}, \Delta)$ prior to doing QBP reduction, and only including the non-truncated options in step 3 of the Replacement (5.1.2) phase, we can greatly reduce the number of variables and constraints in the *L-MaxSAT* formulation.

Note that applying QBP Reduction with $\Delta = 1$ maintains optimality, as it only truncates an IO-level option only when it is strictly inferior to another option, such as in **(i)**. Applying QBP Reduction with larger Δ allows a greater reduction of MaxSAT variables and constraints. However, it has the drawback that we lose the guarantee of finding the most optimal *valid bootstrapping* (B, levels).

5.2 Auto-Compression

While the QBP Reduction improves performance by compressing a single instance of a SISO computation, we also find that multiple, repetitive instances of the same computation occur in many workloads. Such computations are defined as *regular computations*, and are characterized by predictable structures and repetitive patterns, which provide opportunities for optimization.

The Auto-Compression method involves identifying such instances of regular computations, and expressing them in the DAG with vectorized form. Single vertices are used to represent a collection of ciphertexts that follow a similar computational pattern, resulting in a smaller total DAG size. The cmp attribute of vertices is used to express vectorization.

The motivation for Auto-Compression is the abundance of regular computations in FHE programs. Linear algebra operations, such as matrix multiplication and convolution, are mostly regular because they operate on data structures with predictable, organized formats, such as matrices and arrays. Auto-Compression is used to simplify the computation DAGs of such linear algebra operations by several orders of magnitude.

Additionally, the Auto-Compression method composes with the QBP Reduction method. For instance, suppose we have a computation where n ReLU calls occur in a regular manner. The QBP reduction can reduce the n ReLU instances to $2 \cdot n$ vertices, and Auto-Compression can further reduce these instances to just two vertices: one compressed v'_{in} vertex and one compressed v'_{out} vertex.

5.2.1 DAG Compression and Correctness

In this section, we describe and prove the correctness of a reduction that compresses identical computational branches in a regular computation. First, we define two vertex attributes which will be helpful in making the proofs below more readable.

Definition 10 (v.parents and v.children)**.** Suppose we have FHE computation DAG $\mathbf{D} = (\mathbf{V}, \mathbf{E})$, and $v \in \mathbf{V}$.

- v.parents is the set of parent vertices of vertex v. Formally, it is defined as follows:

$$v.\mathsf{parents} = \{p \in \mathbf{V} \text{ such that } (p \to v) \in \mathbf{E}\}$$

- v.children is the set of child vertices of vertex v. Formally, it is defined as follows:

$$v.\mathsf{children} = \{c \in \mathbf{V} \text{ such that } (v \to c) \in \mathbf{E}\}$$

Next, we determine what it means to be a *regular computation* in the scope of a computation DAG. A *regular computation* is characterized by a collection of identical *sub-DAGs* which split from the same point. Below is a formal definition of what it means for two *sub-DAGs* to be *semi-equivalent*, a shorthand for: "identical sub-DAGs splitting from the same point."

Definition 11 (Sub-DAG Semi-equivalence)**.** Let $\mathbf{D} = (\mathbf{V}, \mathbf{E})$ be a computation DAG, and let $\mathbf{D}_A = (\mathbf{V}_A, \mathbf{E}_A)$ and $\mathbf{D}_B = (\mathbf{V}_B, \mathbf{E}_B)$ be *sub-DAGs* of $\mathbf{D}$. $\mathbf{D}_A$ and $\mathbf{D}_B$ are *semi-equivalent* (ie. identical *sub-DAGs* that split from the same point)

$$\iff$$

There exists a bijection $f : \mathbf{V}_A \leftrightarrow \mathbf{V}_B$ such that for each $v_A \in \mathbf{V}_A$, all the conditions below are true: [2]

- $f(v_A).\mathsf{op} = v_A.\mathsf{op}$. [3]
- $\forall p_A \in v_A.\mathsf{parents}, \exists p_B \in f(v_A).\mathsf{parents}$ such that $f(p_A) = p_B$.
- $\forall c_A \in v_A.\mathsf{children}, \exists c_B \in f(c_A).\mathsf{children}$ such that $f(c_A) = c_B$.

Definition 12 (reduce Function)**.** This function applies the compression of the *semi-equivalent* sub-DAGs. When specified a set of n *semi-equivalent* sub-DAGs, this function simplifies the n instances into one compressed instance.

- **Input:** An instance of LA-BTS $I = (L_{max}, \mathbf{D}, \mathsf{c_{bts}})$, as well a set of *sub-DAGs* $\{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n\}$. The following two conditions must hold:
  - For all $v \in \mathbf{V}$, $v.\mathsf{cpr} = 1$
  - $\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n$ are all *sub-DAGs* of $\mathbf{D}$ and are all *semi-equivalent*.
- **Output:** A new instance $I' = (L_{max}, \mathbf{D}', \mathsf{c_{bts}})$, where $\mathbf{D}' = (\mathbf{V}', \mathbf{E}')$ is constructed as follows:
  - We start with the original $\mathbf{D}$
  - The vertices and edges internal to *sub-DAGs* $\mathbf{d}_2, \mathbf{d}_3..., \mathbf{d}_n$ are removed
  - All vertices $v_{\mathbf{d}_1}$ in $\mathbf{d}_1$ instance is updated such that $v_{\mathbf{d}_1}.\mathsf{cpr} = n$

Definition 13 (reconstruct Function)**.** This function takes a solution to the reduced DAG and constructs a solution to the original DAG.

[2]Note that the conditions below are automatically satisfied if $f(v_A) = v_A$

[3]The equality of operators also implies the equality of all lset, costs, lmap attributes of the bijectively corresponding pairs.

- **Input:** A solution (B', levels') to $I' = (L_{max}, \mathbf{D}', \mathsf{c_{bts}})$, as well as $\{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n\}$, a set of *sub-DAGs* of $\mathbf{D}$. There must be an instance of $\mathbf{d}_1$ in $\mathbf{D}'$, where each internal vertex $v_{\mathbf{d}_1}$ has $v_{\mathbf{d}_1}.\mathsf{cpr} = n$.
- **Output:** The solution (B, levels) to I is constructed from (B', levels') as follows:
  - **Step 1:** We handle the bootstrapping and level assignments for the vertices v in $\mathbf{D}$ which are on one of $\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n$. For such v, there is a corresponding compressed vertex v' in $\mathbf{V}'$ (on the compressed $\mathbf{d}_1$).
    - If $v' \in B'$, add v to B
    - Set $\mathsf{levels}[v]$ to be equal to $\mathsf{levels}'[v']$
  - **Step 2:** We handle the bootstrapping and level assignments for the remaining vertices v in $\mathbf{D}$. For such v, there is an equivalent vertex $v' \in \mathbf{D}'$ (not on the compressed $\mathbf{d}_1$).
    - Add v' to B' iff $v \in B$
    - Set $\mathsf{levels}[v]$ to be equal to $\mathsf{levels}'[v']$

Lemma 8. If (B', levels') describes a *valid bootstrapping* for I',

$$(B, \mathsf{levels}) = \mathsf{reconstruct}(B', \mathsf{levels}', \{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n\})$$

describes a *valid bootstrapping* for I.

Proof. Let $e = (v_{parent} \to v_{child})$ be an arbitrary edge in $\mathbf{E}$. As described in the reconstruct function, both of v_{parent} and v_{child} have corresponding/equivalent vertices v'_{parent} and v'_{child} in $\mathbf{D}'$.

Since (B', levels') describes a *valid bootstrapping*, either

- $v'_{parent} \in B' \implies v_{parent} \in B$, or
- $\mathsf{levels}'[v'_{child}] \in (v'_{parent} \to v'_{child}).\mathsf{lmap}[\mathsf{levels}'[v'_{parent}]]$
  $\implies \mathsf{levels}[v_{child}] \in (v_{parent} \to v_{child}).\mathsf{lmap}[\mathsf{levels}[v_{parent}]]$

□

Lemma 9. Applying the reconstruct does not change the σ value. That is,

$$\sigma(B', \mathsf{levels}') = \sigma(\mathsf{reconstruct}(B', \mathsf{levels}', \{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n\}))$$

Proof. Let $(B, \mathsf{levels}) = \mathsf{reconstruct}(B', \mathsf{levels}', \{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n\})$. We want to compare the σ values of solutions (B', levels') for I' and (B, levels) for I.

Note that both solutions are identical for all vertices outside of the *semi-equivalent sub-DAGs*. Thus, we just have to compare the σ contribution of the compressed instance of $\mathbf{d}_1$ for I' and the σ contribution of $\{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n\}$ for I.

Let $\mathbf{V}'_{\mathbf{d}}$ denote the set of vertices in $\mathbf{D}'$ which are a part of the compressed instance of $\mathbf{d}_1$. Let $\mathbf{V}_{\mathbf{d}}$ denote the set of vertices in $\mathbf{D}$ which are a part of the n *semi-equivalent sub-DAGs* $\{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n\}$.

- The bootstrap and level assignments of $\mathbf{V'_d}$ has the following σ contribution for I':

$$\left(\mathsf{c_{bts}} \cdot |B'_{\mathbf{d}}| \cdot v_b.\mathsf{cpr}\right) + \left(\sum_{v' \in \mathbf{V'_d}} v'.\mathsf{cpr} \cdot v'.\mathsf{costs}[\mathsf{levels}'[v']]\right)$$
$$= \left(n \cdot \mathsf{c_{bts}} \cdot |B'_{\mathbf{d}}|\right) + \left(n \cdot \sum_{v' \in \mathbf{V'_d}} v'.\mathsf{costs}[\mathsf{levels}'[v']]\right)$$

Here, $|B'_{\mathbf{d}}|$ denotes the number of vertices in $\mathbf{V'_d}$ that is a part of B'.
- The bootstrap and level assignments $\mathbf{V_d}$ has the following σ contribution for I:

$$\left(\mathsf{c_{bts}} \cdot |B_{\mathbf{d}}| \cdot v_b.\mathsf{cpr}\right) + \left(\sum_{v \in \mathbf{V_d}} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]\right)$$

Here, $|B_{\mathbf{d}}|$ denotes the number of vertices in $\mathbf{V_d}$ that is a part of B. Note that the reconstruct function bootstraps n vertices on $\mathbf{V_d}$ for each vertex bootstrapped on $\mathbf{V'_d}$. Thus,

$$|B_{\mathbf{d}}| = n \cdot |B'_{\mathbf{d}}| \qquad \textbf{(i)}$$

Additionally, $\mathbf{V_d}$ consists of n *semi-equivalent sub-DAGs*, which are each bootstrapped and assigned levels equivalently. Thus,

$$\sum_{v \in \mathbf{V_d}} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]] = n \cdot \sum_{v \in \mathbf{V_{d_1}}} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]] \qquad \textbf{(ii)}$$

where $\mathbf{V_{d_1}}$ is a subset of $\mathbf{V_d}$ consisting of the vertices from just $\mathbf{d}_1$. Thus, we can simplify the σ contribution as follows:

$$\left(\mathsf{c_{bts}} \cdot |B_{\mathbf{d}}| \cdot v_b.\mathsf{cpr}\right) + \left(\sum_{v \in \mathbf{V_d}} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]]\right)$$
$$= \left(\mathsf{c_{bts}} \cdot n \cdot |B'_{\mathbf{d}}| \cdot v_b.\mathsf{cpr}\right) + \left(n \cdot \sum_{v \in \mathbf{V_{d_1}}} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\mathsf{levels}[v]]\right)$$
(Via **(i)** and **(ii)**)
$$= \left(\mathsf{c_{bts}} \cdot n \cdot |B'_{\mathbf{d}}|\right) + \left(n \cdot \sum_{v \in \mathbf{V_{d_1}}} v.\mathsf{costs}[\mathsf{levels}[v]]\right) \qquad (\text{As all } v.\mathsf{cpr} = 1)$$
$$= \left(\mathsf{c_{bts}} \cdot n \cdot |B'_{\mathbf{d}}|\right) + \left(n \cdot \sum_{v' \in \mathbf{V'_d}} v'.\mathsf{costs}[\mathsf{levels}'[v']]\right)$$
(As reconstruct assigns levels to $\mathbf{V_{d_1}}$ in I based on $\mathbf{V'_d}$'s assignments in I')

Thus, the two solutions have the same σ contribution. □

Theorem 4. If (B', levels') is an optimal solution to I', then $(B, \mathsf{levels}) = \mathsf{reconstruct}(B', \mathsf{levels}')$ is an optimal solution to I.

Proof. We prove the contra-positive:

$$\text{If } \exists \text{ some better solution } (\hat{B}, \widehat{\mathsf{levels}}) \text{ to } I, \text{ then } \exists \text{ a better solution } (\hat{B}', \widehat{\mathsf{levels}}') \text{ to } I'$$

Suppose $(\hat{B}, \widehat{\mathsf{levels}})$ is a better solution than (B, levels) for I. That is,

$$\sigma(\hat{B}, \widehat{\mathsf{levels}}) < \sigma(B, \mathsf{levels})$$

In the DAG $\mathbf{D}$, there are n *semi-equivalent sub-DAGs* $\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n$. Given the new solution $(\hat{B}, \widehat{\mathsf{levels}})$, we locate the sub-DAG instance $\mathbf{d}_i$ with the lowest σ contribution. In the case of ties, we can break them randomly. Then, we can construct a new solution $(\hat{B}', \widehat{\mathsf{levels}}')$ to I' as follows:

- **Step 1:** We handle the bootstrapping and level assignments for the vertices v' in $\mathbf{D}'$ which are in the compressed instance of *sub-DAG* $\mathbf{d}_1$. For such v', there is a corresponding uncompressed vertex $v \in \mathbf{d}_i$.
  - If $v \in \hat{B}$, add v' to $\hat{B}'$
  - Set $\widehat{\mathsf{levels}}'[v']$ to be equal to $\widehat{\mathsf{levels}}[v]$
- **Step 2:** We handle the bootstrapping and level assignments for the remaining vertices v' in $\mathbf{D}'$. For such v', there is an equivalent vertex $v \in \mathbf{D}$.
  - If $v \in \hat{B}$, add v' to $\hat{B}'$
  - Set $\widehat{\mathsf{levels}}'[v']$ to be equal to $\widehat{\mathsf{levels}}[v]$

We can conclude two things about $(\hat{B}', \widehat{\mathsf{levels}}')$:

1. $(\hat{B}', \widehat{\mathsf{levels}}')$ describes a *valid bootstrapping* for I':
   To justify this, we use the same reasoning as in Lemma 8. Let $e' = (v'_{parent} \to v'_{child})$ be an arbitrary edge in $\mathbf{E}'$. As described in the solution construction above, both of v'_{parent} and v'_{child} have corresponding/equivalent vertices v_{parent} and v_{child} in $\mathbf{D}$.
   Since $(\hat{B}, \widehat{\mathsf{levels}})$ describes a *valid bootstrapping*, either
   - $v_{parent} \in \hat{B} \implies v'_{parent} \in \hat{B}'$, or
   - $\widehat{\mathsf{levels}}[v_{child}] \in (v_{parent} \to v_{child}).\mathsf{lmap}[\widehat{\mathsf{levels}}[v_{parent}]]$
     $\implies \widehat{\mathsf{levels}}'[v'_{child}] \in (v'_{parent} \to v'_{child}).\mathsf{lmap}[\widehat{\mathsf{levels}}'[v'_{parent}]]$

   Thus, $(\hat{B}', \widehat{\mathsf{levels}}')$ describes a *valid bootstrapping*.
2. $\sigma(\hat{B}', \widehat{\mathsf{levels}}') \leq \sigma(\hat{B}, \widehat{\mathsf{levels}})$:
   Note that both solutions are identical for all vertices outside of the *semi-equivalent sub-DAGs* $\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n$.
   Thus, we just have to compare the σ contribution of the compressed instance of $\mathbf{d}_1$ for I' and the σ contribution of the n *semi-equivalent sub-DAGs* $\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n$ for I.
   Let $\mathbf{V}'_\mathbf{d}$ denote the set of vertices in $\mathbf{D}'$ which are a part of the compressed instance of $\mathbf{d}_1$.
   Let $\mathbf{V}_\mathbf{d}$ denote the set of vertices in $\mathbf{D}$ which are a part of the n *semi-equivalent sub-DAGs* $\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n$.

- The bootstrap and level assignments of $\mathbf{V}'_{\mathbf{d}}$ has the following σ contribution for I':

$$\left(\mathsf{c_{bts}} \cdot |\hat{B}'_{\mathbf{d}}| \cdot v_b.\mathsf{cpr}\right) + \left(\sum_{v' \in \mathbf{V}'_{\mathbf{d}}} v'.\mathsf{cpr} \cdot v'.\mathsf{costs}[\widehat{\mathsf{levels}}'[v']]\right)$$
$$= \left(n \cdot \mathsf{c_{bts}} \cdot |\hat{B}'_{\mathbf{d}}|\right) + \left(n \cdot \sum_{v' \in \mathbf{V}'_{\mathbf{d}}} v'.\mathsf{costs}[\widehat{\mathsf{levels}}'[v']]\right)$$

  Here, $|\hat{B}'_{\mathbf{d}}|$ denotes the number of vertices in $\mathbf{V}'_{\mathbf{d}}$ that is a part of $\hat{B}'$.
- The bootstrap and level assignments $\mathbf{V}_{\mathbf{d}}$ has the following σ contribution for I:
  The σ contribution is lower bounded by n times that sigma contribution of $\mathbf{d}_i$, the sub-DAG instance with the lowest σ contribution. Thus, the lower bound is

$$n \cdot \left(\mathsf{c_{bts}} \cdot |\hat{B}_{\mathbf{d}_i}| \cdot v_b.\mathsf{cpr}\right) + n \cdot \left(\sum_{v \in \mathbf{V}_{\mathbf{d}_i}} v.\mathsf{cpr} \cdot v.\mathsf{costs}[\widehat{\mathsf{levels}}[v]]\right)$$
$$= (n \cdot \mathsf{c_{bts}} \cdot |\hat{B}_{\mathbf{d}_i}|) + \left(n \cdot \sum_{v \in \mathbf{V}_{\mathbf{d}_i}} v.\mathsf{costs}[\widehat{\mathsf{levels}}[v]]\right)$$

  Here, $|\hat{B}_{\mathbf{d}_i}|$ denotes the number of vertices in $\mathbf{V}_{\mathbf{d}_i}$ that is a part of $\hat{B}$. Note that based on how $\hat{B}'$ was constructed,

$$|\hat{B}_{\mathbf{d}_i}| = |\hat{B}'_{\mathbf{d}}| \tag{\textbf{i}}$$

  Additionally, based on how $\hat{B}'$ was constructed,

$$\sum_{v \in \mathbf{V}_{\mathbf{d}_i}} v.\mathsf{costs}[\widehat{\mathsf{levels}}[v]] = \sum_{v' \in \mathbf{V}'_{\mathbf{d}}} v'.\mathsf{costs}[\widehat{\mathsf{levels}}'[v']] \tag{\textbf{ii}}$$

  By **(i)** and **(ii)**, the lower bound for $\mathbf{V}_{\mathbf{d}}$'s σ contribution to I is equal to:

$$\left(n \cdot \mathsf{c_{bts}} \cdot |\hat{B}'_{\mathbf{d}}|\right) + \left(n \cdot \sum_{v' \in \mathbf{V}'_{\mathbf{d}}} v'.\mathsf{costs}[\widehat{\mathsf{levels}}'[v']]\right)$$

This allows us to conclude that $\sigma(\hat{B}', \widehat{\mathsf{levels}}') \leq \sigma(\hat{B}, \widehat{\mathsf{levels}})$.

Given the two conclusions about $(\hat{B}', \widehat{\mathsf{levels}}')$, we can construct the following chain of inequalities.

$$\begin{aligned}
\sigma(\hat{B}', \widehat{\mathsf{levels}}') &\leq \sigma(\hat{B}, \widehat{\mathsf{levels}}) && \text{(Concluded directly above)}\\
&< \sigma(B, \mathsf{levels}) && \text{(By assumption that } (\hat{B}, \widehat{\mathsf{levels}}) \text{ is more optimal)}\\
&= \sigma(\mathsf{reconstruct}(B', \mathsf{levels}', \{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n\})) && \text{(By definition of } (B, \mathsf{levels}))\\
&\leq \sigma(B', \mathsf{levels}') && \text{(By Lemma 9)}
\end{aligned}$$

We have found a better solution for I' as desired. □

The combination of Theorem 2 and Theorem 4 shows that given a DAG $\mathbf{D} = (\mathbf{V}, \mathbf{E})$, which contains *semi-equivalent sub-DAGs* $\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n$, we can solve LA-BTS instance $I = (L_{max}, \mathbf{D}, \mathsf{c_{bts}})$ by:

- Obtaining new LA-BTS instance $I' = \mathsf{reduce}(I, \{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n\})$
- Solving I' to obtain solution (B', levels')
- Obtaining solution (B, levels) to I by computing:

$$(B, \mathsf{levels}) = \mathsf{reconstruct}(B', \mathsf{levels}', \{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n\})$$

5.2.2 Algorithm for Finding Compression Opportunities

This section describes an algorithm for finding opportunities for DAG compression.

- As input, the algorithm takes in a DAG $\mathbf{D} = (\mathbf{V}, \mathbf{E})$ be a DAG where for every vertex $v \in \mathbf{V}$, $v.\mathsf{cpr} = 1$.
- As output, the algorithm returns a partitioning of $\mathbf{V}$.

  If two different vertices v_1 and v_2 are assigned to the same partition, it means that v_1 and v_2 are corresponding vertices on *semi-equivalent sub-DAGs* $\mathbf{d}_1$ and $\mathbf{d}_2$, respectively.

The algorithm is based on equivalence relations for DAG vertices called n-Level Similarity

Definition 14 (n-Level Similarity)**.** For non-negative integer n, n-level similarity ($\approx_n$) is defined as follows:

If $v_A = v_B$, then $v_A \approx_n v_B$. Otherwise, we case on n as follows:

- If $n = 0$ (Base Case): $v_A \approx_0 v_B$ only if the following conditions hold:
  - $v_A.\mathsf{parents}$, $v_B.\mathsf{parents}$ are non-empty, and $v_A.\mathsf{parents} \approx_0^{set} v_B.\mathsf{parents}$ [4]
  - $v_A.\mathsf{op} = v_B.\mathsf{op}$
- If $n > 0$ (General Case): $v_A \approx_n v_B$ only if the following conditions hold:
  - $v_A.\mathsf{parents}$, $v_B.\mathsf{parents}$ are non-empty, and $v_A.\mathsf{parents} \approx_n^{set} v_B.\mathsf{parents}$
  - $v_A.\mathsf{op} = v_B.\mathsf{op}$
  - $v_A.\mathsf{children} \approx_{n-1}^{set} v_B.\mathsf{children}$

Definition 15 (partitionByDepthSim)**.** The function partitionByDepthSim() takes a computation DAG $\mathbf{D} = (\mathbf{V}, \mathbf{E})$ as input, and returns a partitioning of $\mathbf{V}$ based on the equivalent relation $\approx_{\mathrm{depth}(\mathbf{D})}$. [5] The pseudocode for the function is as follows:

1. Get a pre-order traversal of DAG $\mathbf{D} = (\mathbf{V}, \mathbf{E})$. That is, the parent nodes are traversed before the children nodes.
2. Traverse the vertices, partitioning them based on equivalence relation $\approx_0$
3. Traverse the vertices again, partitioning them based on equivalence relation $\approx_1$

[4] $\approx_0^{set}$ indicates set-wise 0-level similarity. $A \approx_n^{set} B \iff \forall v_A \in A, \exists v_B \in B$ such that $v_A \approx_n v_B$, and $\forall v_B \in B, \exists v_A \in A$ such that $v_A \approx_n v_B$.

[5] $\mathrm{depth}(\mathbf{D})$ denotes the length of the longest path in $\mathbf{D}$, where a path is a sequence of vertices such that each vertex is a child of the last.

4. Repeat until we obtain a partitioning based on equivalence relation
   Since each partition depends solely on the previous iteration, it suffices to stop iterating when $\approx_i$'s partitioning is equal to $\approx_{i-1}$'s.
5. Return the partitioning based on the $\approx_{\text{depth}(\mathbf{D})}$ equivalence relation.

Now, we want to show that the following is true about the partitioning P returned by partitionByDepthSim$(\mathbf{D})$:

If P assignes two different vertices v_1 and v_2 to the same partition,
it means that v_1 and v_2 are corresponding vertices on
semi-equivalent sub-DAGs $\mathbf{d}_1$ and $\mathbf{d}_2$, respectively.

To prove this statement, we define the following function:

Definition 16 (findSemiEqSubDags)**.** The inputs, output, and algorithm for this function is as follows:

- **Inputs:** A computation DAG $\mathbf{D} = (\mathbf{V}, \mathbf{E})$ and two unique vertices $v_A, v_B \in \mathbf{V}$ such that $v_A \approx_{\text{depth}(\mathbf{D})} v_B$.
- **Output:** Two *semi-equivalent sub-DAGs* of $\mathbf{D}$, denoted $\mathbf{d}_A$ and $\mathbf{d}_B$, such that v_A and v_B are corresponding vertices on the two *sub-DAGs*.
- **Algorithm:** As input, we are given $\mathbf{D} = (\mathbf{V}, \mathbf{E})$, and $v_A, v_B \in \mathbf{V}$, such that $v_A \neq v_B$, and $v_A \approx_{\text{depth}(\mathbf{D})} v_B$.

  We will construct sub-DAGs $\mathbf{d}_A = (\mathbf{V}_A, \mathbf{E}_A)$, $\mathbf{d}_B = (\mathbf{V}_B, \mathbf{E}_B)$ and bijection $f : \mathbf{V}_A \leftrightarrow \mathbf{V}_B$ using the following procedure:

  Phase 1: We begin by adding v_A to $\mathbf{V}_A$, v_B to $\mathbf{V}_B$, and set $f(v_A) = v_B$. Throughout this phase, we maintain the invariant that $f(v) \approx_{\text{depth}(\mathbf{D})} v$.
  First, we perform an upwards BFS from v_A (ie. towards the parents). The BFS frontier begins as v_A. On each iteration, we take vertex u_A from the frontier and complete the following steps:
  1. Let $u_B = f(u_A)$. It holds that $u_A \approx_{\text{depth}(\mathbf{D})} u_B$.
  2. For each incoming edge $e_A = (p_A \to u_A)$, we find edge $e_B = (p_B \to u_B)$ such that $p_A \approx_{\text{depth}(\mathbf{D})} p_B$. Such an edge is guaranteed to exist by the definition of $\approx_{\text{depth}(\mathbf{D})}$.
  3. Add e_A to $\mathbf{E}_A$, e_B to $\mathbf{E}_B$, p_A to $\mathbf{V}_A$, p_B to $\mathbf{V}_B$, and set $f(p_A) = p_B$.
  4. p_A and p_B are different vertices, add p_A to the frontier.

  We repeat this process until the frontier is empty. This ends *Phase 1.* At the end of Phase 1, the roots nodes of $\mathbf{V}_A$ and $\mathbf{V}_B$ will be identical. These roots represent the point in the computation where it splits into identical sub-DAGs. Additionally, the the bijection $f : \mathbf{V}_A \leftrightarrow \mathbf{V}_B$ maintains the property that $v \approx_{\text{depth}(\mathbf{D})} f(v)$.

  Phase 2: We begin with the $\mathbf{d}_A, \mathbf{d}_B, f$ constructed from the previous phase. Throughout this phase, we maintain the invariant that for each $v \in \mathbf{V}_A$, $f(v) \approx_n v$ for some $n \geq 0$.
  We perform a downward BFS, that is, towards the children. The BFS frontier begins as the roots of $\mathbf{V}_A$. On each iteration, we take vertex u_A from the frontier and complete the following steps:

1. Let $u_B = f(u_A)$. It holds that $u_A \approx_n u_B$ for some positive n.
2. For each outgoing edge $e_A = (u_A \to c_A)$, we find edge $e_B = (u_B \to c_B)$ such that $p_A \approx_{n-1} p_B$. Such an edge is guaranteed to exist by the definition of $\approx_n$. Note that at this step, $\approx_{n-1}$ is always non-negative, since the roots start with $n = \mathrm{depth}(\mathbf{D})$.
3. Add e_A to $\mathbf{E}_A$, e_B to $\mathbf{E}_B$, c_A to $\mathbf{V}_A$, c_B to $\mathbf{V}_B$, and set $f(c_A) = c_B$.
4. c_A and c_B are different vertices, add c_A to the frontier.

We repeat this process until the frontier is empty. This ends *Phase 2*. At the end of this phhase, the the bijection $f : \mathbf{V}_A \leftrightarrow \mathbf{V}_B$ maintains the property that for each $v_A \in \mathbf{V}$, $v_A \approx_n f(v_A)$ for some non-negative n. Return $\mathbf{d}_A$ and $\mathbf{d}_B$.

Finally, we will show that findSemiEqSubDags()'s algorithm is correct. For the theorems below, let $\mathbf{D} = (\mathbf{V}, \mathbf{E})$ denote a computation DAG and v_A, v_B denote two unique vertices in $\mathbf{V}$ such that $v_A \approx_{\mathrm{depth}(\mathbf{D})} v_B$. Compute $\mathbf{d}_A, \mathbf{d}_B$ as follows:

$$(\mathbf{d}_A, \mathbf{d}_B) = \mathsf{findSemiEqSubDags}(\mathbf{D}, v_A, v_B)$$

Theorem 5. $\mathbf{d}_A$ and $\mathbf{d}_B$ are **sub-DAGs** of $\mathbf{D}$.

Proof. $\mathbf{D}_A$ and $\mathbf{D}_B$ are *sub-DAGs* of $\mathbf{D}$, as the BFS process ensures that both $\mathbf{D}_A$ and $\mathbf{D}_B$ are fully connected and only involve vertices from their respective vertex sets $\mathbf{V}_A$ and $\mathbf{V}_B$. □

Theorem 6. $\mathbf{d}_A$ and $\mathbf{d}_B$ are *semi-equivalent*, and v_A and v_B are corresponding vertices on $\mathbf{d}_A$ and $\mathbf{d}_B$, respectively.

Proof. Recall the condition for $\mathbf{d}_A$ and $\mathbf{d}_B$ *semi-equivalent* (from Definition 11): there must exist a bijection $f' : v_A \leftrightarrow v_b$ such that for each $v_A \in \mathbf{V}_A$, all the conditions below are true:

1. $f'(v_A).\mathsf{op} = v_A.\mathsf{op}$.
2. $\forall p_A \in v_A.\mathsf{parents}, \exists p_B \in f'(v_A).\mathsf{parents}$ such that $f'(p_A) = p_B$.
3. $\forall c_A \in v_A.\mathsf{children}, \exists c_B \in f'(c_A).\mathsf{children}$ such that $f'(c_A) = c_B$.

The bijection f we constructed in findSemiEqSubDags()'s algorithm satisfies these conditions. The first condition is satisfied from the fact that for all $v_A \in \mathbf{V}_A$, $v_A \approx_n f(v_A)$ for some non-negative n. The latter two conditions are satisfied because for every edge $e_A = (p_A \to c_A)$ added to $\mathbf{E}_A$ during the BFS searches there is an edge $e_B = (p_B \to c_B)$ added to $\mathbf{E}_B$ such that $f(p_A) = p_B$ and $f(c_A) = c_B$.

Additionally, the steps in the BFS searches ensure that edges are branched out v_A and v_B in an equivalent manner when constructing $\mathbf{D}_A$ and $\mathbf{D}_B$. This ensures v_A and v_B are corresponding vertices on the two *sub-DAGs*. □

Finally, recall that the function call $\mathsf{partitionByDepthSim}(\mathbf{D} = (\mathbf{V}, \mathbf{E}))$ returns a partitioning P of $\mathbf{V}$ based on the equivalence relation $\approx_{\mathrm{depth}(\mathbf{D})}$.

If two unique vertices v_A and v_B are in the same partition for P, then $v_A \approx_{\mathrm{depth}(\mathbf{D})} v_B$. By computing $(\mathbf{d}_A, \mathbf{d}_B) = \mathsf{findSemiEqSubDags}(\mathbf{D}, v_A, v_B)$, we can apply Theorems 5 and 6 ensure that v_A and v_B are corresponding vertices on *semi-equivalent sub-DAGs* as desired.

Chapter 6

Evaluation

This chapter begins by describing the experimental setup. In 6.2, we present the performance of the DAG reduction methods presented in Chapter 5. In the remaining sections, we compare DaCapo [7], the current state-of-the-art automatic bootstrapping placement scheme, with Saturn for various performance metrics. We compare the bootstrapping selection times, with various degrees of QBP Truncation (5.1.4) for Saturn, in 6.4. Furthermore, we compare estimated end-to-end program runtime in 6.5 and the bootstrapping counts in 6.6.

6.1 Experimental Setup

For our evaluation, we use the deep learning benchmark suite from DaCapo [7]. The suite consists of ResNet-20 [16], AlexNet [22], VGG16 [31], SqueezeNet [18], and MobileNet [17] with two different activation functions (ReLU, SiLU). To obtain the computation DAGs of each benchmark, we use the Hecate [24] compiler's trace functionality on the front-end implementations from DaCapo's GitHub repository [8]. The evaluations run on Intel(R) Xeon(R) Gold 6226R CPU @ 2.90Ghz.

6.2 DAG Reduction Performance

Table 6.1 demonstrates the effectiveness of Saturn's two DAG reduction methods. For each column, *Original DAG Size* is the number of vertices in the original computation DAG $\mathbf{D}$ representing each benchmark. *Reduced DAG Size* is the number vertices in the benchmark's computation DAG $\mathbf{D}''$ after DAG reduction via the following steps:

- All instances of ReLU/SiLU are simplified via QBP Reduction (5.1) to obtain $\mathbf{D}'$.
- Vertices of $\mathbf{D}'$ are partitioned via $P = \mathsf{partitionByDepthSim}(\mathbf{D}')$.
- For each partition in P, the included vertices are compressed as described in Section 5.2.1 to obtain DAG $\mathbf{D}''$

DAG Reduction Time is the time it takes to perform the steps above for a given benchmark.

Model	**ResNet-20 [ReLU]**	**AlexNet [ReLU]**	**VGG16 [ReLU]**	**SqueezeNet [ReLU]**	**MobileNet [ReLU]**
Original DAG Size	9411	50136	44518	21101	45267
Activ. Function Calls	19	8	15	10	27
Reduced DAG Size	127	60	113	125	171
DAG Reduction Time (s)	1.69	41.27	60.11	14.80	117.65

Table 6.1: The performance of the DAG reduction methods on deep learning DAGs

The results demonstrate that Saturn's DAG reduction methods are able to decrease DAG size by 2-3 orders of magnitude for the given benchmarks. Additionally, the reductions are efficient, taking 2 minutes at worst. Note that the current implementation is done in Python, so faster reduction times could be achieved by rewriting the implementation with a fast programming language.

6.3 QBP Truncation for ReLU and SiLU

Table 6.2 shows the sizes of ReLU and SiLU's quadratic behavior profiles after QBP truncation with various Δs. Note that $\Delta = 1$ truncation does not reduce SiLU's QBP by much. The effects of this can be observed in Figure x of the next section.

Activation	**ReLU**	**SiLU**
Size of Original QBP	289	289
Size After $\Delta = 1$ Truncation	109	283
Size After $\Delta = 10$ Truncation	100	111
Size After $\Delta = 100$ Truncation	52	79

Table 6.2: QBP size reduction from QBP truncation with various values of Δ for ReLU and SiLU

6.4 Bootstrap Selection Time

Table 6.3 compares the bootstrap selection time of DaCapo versus Saturn with $\Delta = 1$ QBP truncation. Recall that $\Delta = 1$ QBP truncation maintains optimality. Saturn's bootstrap selection time is the sum of the time it takes to find the Auto-Compression opportunties and the time it take to solve the *L-MaxSAT* problem corresponding to the benchmark DAG after the reduction techniques are applied.

Model	**ResNet-20**		**AlexNet**		**VGG16**		**SqueezeNet**		**MobileNet**	
	ReLU	**SiLU**	**ReLU**	**SiLU**	**ReLU**	**SiLU**	**ReLU**	**SiLU**	**ReLU**	**SiLU**
DaCapo	15.8	14.4	1042.3	336.5	230.1	188.1	89.1	44.1	222.8	218.0
AutoComp Time	1.7	1.7	41.3	41.3	60.1	60.1	14.8	14.8	117.7	117.7
MaxSAT Solve Time	6.0	-	1.2	28.5	6.1	-	4.8	34.7	8.0	-
Saturn ($\Delta = 1$)	7.7	-	42.5	69.8	66.2	-	19.6	49.5	125.7	-

Table 6.3: The bootstrap selection times (s) of DaCapo and Saturn ($\Delta = 1$) for the benchmarks. Note that a "-" represents a timeout ($\geq 600s$).

Note that for the SiLU benchmarks, Saturn's bootstrap selection time is dominated by the MaxSAT solve times. The poor truncatability of SiLU's QBP for $\Delta = 1$ results in the corresponding *L-MaxSAT* problems being difficult to solve. These solve times can be greatly reduced via QBP truncation with larger Δ.

In Table 6.4, we again compare DaCapo's and Saturn's bootstrap selection time. Here the Saturn method is configured to use QBP truncation with $\Delta = 10$. With a higher Δ, Saturn is able to achieve faster bootstrap selection times. Note that with this configuration, Saturn is not guaranteed to find the most optimal *valid bootstrapping* (B, levels). The evaluations show that with a $\Delta = 10$ QBP truncation, Saturn consistently has faster bootstrap selection times than DaCapo. On average, Saturn provides a $5.25\times$ speedup.

Model	**ResNet-20**		**AlexNet**		**VGG16**		**SqueezeNet**		**MobileNet**	
	ReLU	**SiLU**	**ReLU**	**SiLU**	**ReLU**	**SiLU**	**ReLU**	**SiLU**	**ReLU**	**SiLU**
DaCapo	15.8	14.4	1042.3	336.5	230.1	188.1	89.1	44.1	222.8	218.0
AutoComp Time	1.7	1.7	41.3	41.3	60.1	60.1	14.8	14.8	117.7	117.7
MaxSAT Solve Time	4.8	6.8	1.4	1.0	4.5	3.2	7.0	5.7	26.1	10.1
Saturn ($\Delta = 10$)	6.5	8.5	42.7	42.3	64.6	63.3	21.8	20.5	143.8	127.8

Table 6.4: The bootstrap selection times (s) of DaCapo and Saturn ($\Delta = 10$) for the benchmarks. Note that a "-" represents a timeout ($\geq 300s$).

6.5 Estimated Performance for CIFAR-10 Inference

For each benchmark, Figure 6.5 compares the estimated inference time for a CIFAR-10 image, when bootstrapped by each method (DaCapo, Saturn). The inference time for each benchmark is estimated using the simplified cost model shown in Table 6.5. Note that this cost model may be inaccurate, as it does not factor in the costs of scale management operations (eg. *rescale*, *downscale*), which must be inserted between arithmetic operations in FHE programs.

Operation	**Level**																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
CMult	0.6	0.7	0.8	1.0	1.2	1.3	1.5	1.6	1.7	2.0	2.3	2.4	2.5	2.7	2.9	3.0	3.1
Boot	354.7																

Table 6.5: The simplified cost model used to estimate the inference times. The numbers are extrapolated from the cost model in [7].

We utilize this limited cost model for our evaluations due to the current lack of support for optimized scale management operations in the Saturn method. To obtain the data points for DaCapo, we use our own custom implementation, which has been altered from the original to place bootstraps under the assumption of naive scale management[1].

Implementing support for optimized scale management is the next step of this research project. Once this functionality is integrated, we will not only be able to employ a more accurate cost model, but also execute inferences with Saturn-compiled benchmarks and compare the end-to-end runtimes with the open source implementation [8] of DaCapo.

Model	**ResNet-20**		**AlexNet**		**VGG16**		**SqueezeNet**		**MobileNet**	
	ReLU	**SiLU**	**ReLU**	**SiLU**	**ReLU**	**SiLU**	**ReLU**	**SiLU**	**ReLU**	**SiLU**
DaCapo*	13881.4	8989.3	13160.5	11868.7	16700.8	14606.3	11235.8	10587.5	23210.3	15970.3
Saturn ($\Delta = 10$)	13881.4	8954.8	13160.5	11869.9	16700.8	14040.5	11002.6	10345.5	23210.3	15940.6

Table 6.6: The estimated inference times for the benchmarks bootstrapped by DaCapo and Saturn

[1]With naive scale management, a *rescale* is placed after each *multiply*. Under this assumption, each *multiply* operation consumes 1 level.

6.6 Bootstrapping Counts

Table 6.7 compares the number of bootstraps placed by DaCapo versus Saturn with $\Delta = 10$ QBP truncation. The data points for DaCapo are obtained using the open source implementation [8]. Overall, the two methods place similar number of bootstraps. These results should be interpreted with caution because as of now, DaCapo and Saturn assume different methods of scale management (*rescale* placements).

Model	**ResNet-20**		**AlexNet**		**VGG16**		**SqueezeNet**		**MobileNet**	
	ReLU	**SiLU**	**ReLU**	**SiLU**	**ReLU**	**SiLU**	**ReLU**	**SiLU**	**ReLU**	**SiLU**
DaCapo	37	19	20	12	29	20	20	19	60	27
Saturn ($\Delta = 10$)	37	20	14	10	29	21	20	19	53	28

Table 6.7: The number of bootstraps placed by DaCapo and Saturn for the benchmarks

Chapter 7

Conclusion

In this thesis, we presented Saturn, a novel framework for optimizing bootstrapping placement in Fully Homomorphic Encryption (FHE) applications. Saturn leverages the Maximum Satisfiability (MaxSAT) problem to determine the most efficient points for performing the bootstrapping operation, thereby minimizing the total runtime of FHE programs.

Our approach begins by representing the FHE computation as a directed acyclic graph (1) and then applies two reduction methods — Quadratic Behavior Profile Reduction (5.1) and Auto-Compression (5.2) — to simplify the DAG. This simplification significantly reduces the complexity of the MaxSAT problem, making it feasible to solve within reasonable timeframes even for large-scale computations, such as neural network inferences.

We validated Saturn using the RNS-CKKS [6] FHE scheme, chosen for its support for fixed-point arithmetic and SIMD operations, which are critical for privacy-preserving machine learning applications. Through extensive evaluations on various deep learning models such as ResNet20 [16], AlexNet [22], VGG16 [31], SqueezeNet [18], and MobileNet [17], Saturn demonstrated improvements in both bootstrapping placement selection time and estimated overall program runtime compared to existing methods.

The key contributions of this work include:

1. The development of a Partial Weighted Maximum Satisfiability model (4.3) tailored to the optimal bootstrapping placement problem (4.2).
2. Introduction of two DAG reduction techniques (5.1), 5.2) that preserve the optimality of the solution while significantly reducing problem complexity.
3. Demonstration of the practicality and effectiveness of Saturn in optimizing bootstrapping placements for real-world FHE applications (6).

Despite the success of Saturn, there are several avenues for future research. One immediate next step is to lift the naive rescaling assumption (2.1.3) to handle more complex scaling scenarios, and allow Saturn to directly compare end-to-end runtimes versus DaCapo [8]. Additionally, exploring the integration of Saturn with other FHE schemes beyond RNS-CKKS could further validate its versatility and robustness. Finally, optimizing Saturn's performance for even larger and more complex FHE computations, such as large language models, remains a crucial area for further research.

Bibliography

[1] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. Openfhe: Open-source fully homomorphic encryption library. In *proceedings of the 10th workshop on encrypted computing & applied homomorphic cryptography*, pages 53–63, 2022. 1

[2] Fabrice Benhamouda, Tancrède Lepoint, Claire Mathieu, and Hang Zhou. Optimization of bootstrapping in circuits. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2423–2433. SIAM, 2017. 1, 3.1, 3.2

[3] Joppe W. Bos, Kristin Lauter, and Michael Naehrig. Private predictive analysis on encrypted medical data. *Journal of Biomedical Informatics*, 50:234–243, 2014. ISSN 1532-0464. doi: https://doi.org/10.1016/j.jbi.2014.04.003. URL `https://www.sciencedirect.com/science/article/pii/S1532046414000884`. Special Issue on Informatics Methods in Medical Privacy. 1

[4] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 409–437. Springer, 2017. 1

[5] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part I 37*, pages 360–384. Springer, 2018. 1

[6] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography – SAC 2018*, pages 347–368, Cham, 2019. Springer International Publishing. ISBN 978-3-030-10970-7. 1, 2, 7

[7] Seonyoung Cheon, Yongwoo Lee, Dongkwan Kim, Ju Min Lee, Sunchul Jung, Taekyung Kim, Dongyoon Lee, and Hanjun Kim. Dacapo: Automatic bootstrapping management for efficient fully homomorphic encryption. In *USENIX Security*, 2024. (document), 1, 2.1.2, 3.1, 3.2, 6, 6.1, 6.5

[8] Seonyoung Cheon, Yongwoo Lee, Ju Min Lee, Dongkwan Kim, Sunchul Jung, Taekyung Kim, Dongyoon Lee, and Hanjun Kim. Dacapo: Automatic bootstrapping man-

agement compiler for fully homomorphic encryption. `https://github.com/corelab-src/dacapo`, 2024. 33rd USENIX Security Symposium (USENIX Security), August 2024. 6.1, 6.5, 6.6, 7

[9] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Chet: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, pages 142–156, 2019. 1

[10] Jessica Davies. *Solving MaxSAT by decoupling optimization and satisfaction*. PhD thesis, University of Toronto, 2013. 2.2.2

[11] Emir Demirovic and Nysret Musliu. Modeling high school timetabling as partial weighted maxsat. In *LaSh 2014: The 4th Workshop on Logic and Search (a SAT/ICLP workshop at FLoC 2014), July 18, Vienna, Austria*, 2014. 2.2.1

[12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012. 1

[13] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, page 169–178, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585062. doi: 10.1145/1536414.1536440. URL `https://doi.org/10.1145/1536414.1536440`. 1

[14] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2021. 3.3

[15] Shai Halevi and Victor Shoup. Helib, 2024. URL `https://github.com/homenc/HElib`. Accessed: 2024-07-17. 1

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 1, 6.1, 7

[17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 1, 6.1, 7

[18] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016. 1, 6.1, 7

[19] Hannes Ihalainen, Jeremias Berg, and Matti J"arvisalo. Refined core relaxation for core-guided maxsat solving. In *CP*, volume 210 of *LIPIcs*, pages 28:1–28:19. Schloss Dagstuhl - Leibniz-Zentrum f"ur Informatik, 2021. 1, 2.2.2, 3.3, 3.4

[20] Matti Järvisalo, Jeremias Berg, Ruben Martins, and Andreas Niskanen. Maxsat evaluation 2023. `https://maxsat-evaluations.github.io/2023/`, 2023. University of Helsinki, Finland and Carnegie Mellon University, USA. 1, 2.2.2

[21] Miran Kim and Kristin Lauter. Private genome analysis through homomorphic encryption. In *BMC medical informatics and decision making*, volume 15, pages 1–12. Springer, 2015. 1

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012. 1, 6.1, 7

[23] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *iEEE Access*, 10:30039–30054, 2022. 1

[24] Yongwoo Lee, Seonyeong Heo, Seonyoung Cheon, Shinnung Jeong, Changsu Kim, Eunkyung Kim, Dongyoon Lee, and Hanjun Kim. Hecate: Performance-aware scale optimization for homomorphic encryption compiler. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–204, 2022. doi: 10.1109/CGO53902.2022.9741265. 1, 6.1

[25] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*, pages 1–23. Springer, 2010. 1

[26] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*, pages 1–23. Springer, 2010. 2.1

[27] Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-wbo: A modular maxsat solver. In *Theory and Applications of Satisfiability Testing–SAT 2014: 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings 17*, pages 438–445. Springer, 2014. 2.2.2

[28] Marie Paindavoine and Bastien Vialla. Minimizing the number of bootstrappings in fully homomorphic encryption. In *International Conference on Selected Areas in Cryptography*, pages 25–43. Springer, 2015. 1, 3.1, 3.2

[29] Microsoft Research. Microsoft seal (release 4.0). `https://github.com/microsoft/SEAL`, 2020. URL `https://github.com/microsoft/SEAL`. Available from https://github.com/microsoft/SEAL. 1

[30] Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. Partial weighted maxsat for optimal planning. In *PRICAI 2010: Trends in Artificial Intelligence: 11th Pacific Rim International Conference on Artificial Intelligence, Daegu, Korea, August 30–September 2, 2010. Proceedings 11*, pages 231–243. Springer, 2010. 2.2.1

[31] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 1, 6.1, 7

[32] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. Sok: Fully homomorphic encryption compilers. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1092–1108. IEEE, 2021. 1

[33] Tommy White, Charles Gouert, Chengmo Yang, and Nektarios Georgios Tsoutsos. Fhe-

booster: Accelerating fully homomorphic execution with fine-tuned bootstrapping scheduling. In *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 293–303. IEEE, 2023. 1, 3.1

[34] Runhua Xu, Nathalie Baracaldo, and James Joshi. Privacy-preserving machine learning: Methods, challenges and directions, 2021. URL `https://arxiv.org/abs/2108.04417`. 1