Learning Search Control Knowledge to Improve Plan Quality

María Alicia Pérez

July 1995
CMU-CS-95-175

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

Thesis Committee:
Jaime Carbonell, Chair
Tom Mitchell
Reid Simmons
Manuela Veloso
Martha Pollack, University of Pittsburgh

This research was supported in part by a scholarship of the Ministerio de Educación y Ciencia of Spain and in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330.

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory, the U. S. Government, or the Spanish Government.

Keywords: Artificial intelligence, planning, problem solving, machine learning, PRODIGY, plan quality, evaluation of plans, planning performance, search control knowledge.

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

LEARNING SEARCH CONTROL KNOWLEDGE TO IMPROVE PLAN QUALITY

ALICIA PEREZ

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

THESIS COMMITTEE CHAIR — DATE: 7/7/95

DEPARTMENT HEAD — DATE: 7/7/95

APPROVED:

DEAN — DATE: 7/11/95

Abstract

Generating good, production-quality plans is an essential element in transforming planners from research tools into real-world applications, but one that has been frequently overlooked in research on machine learning for planning. Most work has aimed at improving the *efficiency of planning* ("speed-up learning") or at acquiring or refining the planner's action model. This thesis focuses on learning search-control knowledge to improve the *quality of the plans* produced by the planner.

Knowledge about plan quality in a domain comes in two forms: (a) a post-facto quality metric that computes the quality (e.g. execution cost) of a plan, and (b) planning-time decision-control knowledge used to guide the planner towards high-quality plans. The first kind is not operational until *after* a plan is produced, but is exactly the kind typically available, in contrast to the far more complex operational decision-time knowledge. Learning operational quality control knowledge can be seen as *translating* the domain knowledge and quality metrics into runtime decision guidance. The full automation of this mapping based on planning experience is the ultimate objective of this thesis.

Given a domain theory, a domain-specific metric of plan quality, and problems which provide planning experience, the QUALITY architecture developed in this thesis automatically acquires operational control knowledge that effectively improves the quality of the plans generated. QUALITY can (optionally) learn from human experts who suggest improvements to the plans at the operator (plan step) level. We have designed two distinct domain-independent learning mechanisms to efficiently acquire quality control knowledge. They differ in the language used to represent the learned knowledge, namely control rules and control knowledge trees, and in the kinds of quality metrics for which they are best suited.

QUALITY is fully implemented on top of the PRODIGY4.0 nonlinear planner. Its empirical evaluation has shown that the learned knowledge produces near-optimal plans (reducing before-learning plan execution costs 8% to 96%). Although the learning mechanisms and learned knowledge representations have been developed for PRODIGY4.0, the framework is general and addresses a problem that must be confronted by any planner that treats planning as a constructive decision-making process.

Acknowledgements

This thesis, and I, owe much to Jaime Carbonell. From the first day as my advisor Jaime was encouraging me and telling me that I could do this thesis, this PhD. I have learned many things from him, and many ideas in the thesis, including the relevance of the topic, originated in very fruitful and fun discussions with him. Thanks for being always so supportive and patient.

The other members of my thesis committee, Tom Mitchell, Reid Simmons, Manuela Veloso, and Martha Pollack challenged me to go out of a narrow view of the problem and shared with me their enthusiasm for the field. My discussions with them were always very enlightening and suggested new lines of work, not all of which I have been able to explore (yet). And thanks for reading and commenting on this long document.

I must thank Manuela in a special way. First of all for her friendship. Then for all the things we shared. Collaborating with Manuela was great and I learned from her much, not only professionally. Manuela has challenged me so many times... And I am extremely thankful.

The Prodigy team also played a big role in this thesis. The current members provided technical support and great feedback on talks and papers: Jim Blythe, Mei Wang, Scott Reilly, Rujith de Silva, Karen Haigh, Rob Driskill, Eugene Fink, Peter Stone, and Luiz Edival de Souza. And thanks too to the past members gone to greater things: Steve Minton, Craig Knoblock, Dan Kuokka, Yolanda Gil, Oren Etzioni, Manuela Veloso, Robert Joseph, Michael Miller, Dan Kahn, Daniel Borrajo, Santiago Rementería, Angela Ribeiro, Masa Iwamoto, Erica Melis, and Vincent Poinot. It was fun to work with Oren on DYNAMIC. Yolanda put a lot of work on the process planning domain and has been a good friend throughout these years. Collaborating with all of you was (and, I hope, will be) a very enjoyable experience. Out of CMU, Caroline Hayes provided a lot of knowledge about process planning and Devika Subramanian suggested different ways to look at the problem of learning plan quality.

I cannot think of a better place to learn and to do research than the School of Computer Science at Carnegie Mellon. Meeting the very best faculty and graduate students in computer science has been a great pleasure. I will never forget how much I enjoyed the class taught by Allen Newell, Tom Mitchell, and Jaime Carbonell on integrated architectures during my second semester at CMU. I would like to remember specially Allen Newell and Nico Haberman for sharing with us their grand vision of computer science research. Angel Jordán has always been

very supportive. Thanks also to all the facilities staff, to Sharon Burks, and to Satya for letting me use a Coda laptop during thesis writing and to the Coda group for their technical support.

The Ministerio de Educación y Ciencia of Spain supported me with a scholarship throughout most of my graduate studies. I am grateful for their offering me the encouragement to come to a top university to pursue a doctoral degree.

A PhD is a hard, long process, but many people made mine very enjoyable. First, thanks to the best officemates in the world: Puneet Kumar, Xuemei Wang, and Henry Rowley. We spent many, many hours together, and shared many good moments. You have been an extraordinary mine of knowledge about the mysteries of Unix, Postscript, networks,... anything! I have also shared offices with Angela Hickman, Ken McMillan, David Long, David Detleffs, and Mike Young. They endured my poor English and many questions about American culture in my initial years here.

During this long process many friends have come and gone. Frequently it has been like being in Spain, but in Pittsburgh. Daniel, Isabel, Alejandro, Teresa, Pepe, José Luis, Matusa, Joserra, Juan Carlos, Lourdes, Charo, Jesús, Elizabeth, Francisco, Maite, Michel, Anibal, Jose, Enrique, Pablo, Harri, Guillermo, Alfonso, Luis, Pedro, Enrique, Raquel, Javier. They all deserve credit for some part of this work, for bearing with me and my moods in the hard times, for making me laugh so much, and for showing me the incredible value of friendship. ¡Gracias! Thanks to the current members of the Spanish troop for enduring with me the last weeks of this thesis and being genuinely supportive and always available: Aurora (thanks for typing some of this), Enrique, Pascal, Nieves, Juan, Angélica, Mari Angeles, and Pedro. My friends in Spain, Julia, Ana, and Irma, were always very supportive. How I regret not having been able to spend more time with all of them!

My other Pittsburgh friends, in particular those from the Oratory (the Catholic campus chaplaincy) have helped me keep things of perspective, out of my work and out of the department. Special thanks to Drew, Joe L., Patty, Sandy, and Dave.

Sr Madeleine Gregg, fcJ and all the Sisters Faithful Companions of Jesus have accompanied me with their love, support, and prayers, reminding me of why I was doing this. A.M.D.G.

Finally, I would like to thank my family for their support through all these years. My parents have taught us with their example a sense of responsibility and hard work, and unselfish giving. But above all they have always shown us unconditional love and support. I have been very fortunate to grow up with my siblings María del Mar, José Manuel, Sonia and Nuria. The one thing I regret about this doctorate is the time we have not spent together.

M. Alicia Pérez
July, 1995

Contents

1 Introduction **1**

1.1 Machine Learning for Planning Systems 3

1.2 Measuring Plan Quality 4

1.3 The Problem 4

1.3.1 Planning Decisions and Plan Quality 5

1.4 The Thesis 9

1.4.1 Overview of the Approach 9

1.4.2 Scientific Contributions 13

1.5 A Reader's Guide to the Thesis 14

2 The Process Planning Domain **17**

2.1 What is Process Planning? 17

2.2 Plan Quality in the Process Planning Domain 19

2.3 An Implementation of Process Planning 21

2.4 An Example 26

2.5 Summary 29

3 Search Control Rule Learning **31**

3.1 The Architecture 32

3.2 The Interactive Plan Checker 34

3.2.1 Description 34

3.2.2 Examples and Further Details 39

3.3 The Control-Rule Learning Algorithm: A Top-Level View 39

3.4 Constructing A Problem Solving Trace From The Plan 42
3.5 Building Plan Trees 43
3.6 Finding Learning Opportunities 44
3.6.1 An Example 49
3.6.2 Why These Learning Opportunities 49
3.7 Learning Operator and Bindings Control Rules 50
3.8 Example Of Learning Operator And Bindings Rules 58
3.9 Learning Goal Preference Control Rules 63
3.9.1 When Are Goal Preferences Needed? An Example 63
3.9.2 How Goal Preference Rules Are Learned 68
3.10 Example Of Learning Goal Preference Control Rules 71
3.11 Learning Control Rule Priorities 76
3.11.1 The Problem: Over-General Rules And Conflicting Preferences . . . 76
3.11.2 How To Break Preference Cycles 77
3.11.3 Learning Control Rule Priorities 80
3.11.4 An Example in an Artificial Domain 82
3.11.5 Examples in the Process Planning Domain 83
3.12 Discussion 86
3.13 Experimental Results 90
3.13.1 The Setting 90
3.13.2 The Training Phase 91
3.13.3 The Test Phase 93
3.14 Summary 96

4 Learning Control Knowledge Trees **97**
4.1 Motivation 97
4.1.1 Example 1: An Artificial Domain 98
4.1.2 Example 2: A Transportation Domain 100
4.1.3 Example 3: The Process Planning Domain 105
4.1.4 Limitations of Using Control Rules to Produce Quality Plans 107

4.1.5 Should We Still Learn Control Rules? 108
4.2 A Different Approach (A Sketch) 110
4.3 A New Representation Formalism for Control Knowledge 113
4.4 Learning Control-Knowledge Trees 117
4.4.1 An Example 118
4.4.2 Building a New Control-Knowledge Tree 121
4.4.3 Updating an Existing Control-Knowledge Tree 132
4.4.4 Learn and Update Other Cktrees 133
4.5 Using Control Knowledge Trees 136
4.5.1 Overview of Control Knowledge Matching 137
4.5.2 Calling the Cktree Matcher 141
4.5.3 Cktree Matching as Traversing the Cktree 143
4.5.4 Generating and Pruning Alternatives 144
4.5.5 When to Stop Traversing the Cktree 147
4.5.6 Matching Universally Quantified Cktree Preconditions 149
4.5.7 Exploring Multiple Alternatives Efficiently 150
4.5.8 Reusing Computation Among Alternatives 152
4.5.9 Cktree Matching When There Are Interacting Goals 158
4.5.10 Using the Same Cktrees for Different Quality Metrics 168
4.5.11 Goal Ordering Control Knowledge 168
4.6 An Example in a Transportation Domain 168
4.7 Discussion 172
4.7.1 Using Cktrees versus Control Rules as a Control Knowledge Representation Formalism 173
4.7.2 Efficiency Issues in Using Cktrees 177
4.7.3 The Accuracy of the Learned Control Knowledge 180
4.7.4 Tradeoffs Between Plan Quality and Planning Efficiency 183
4.8 Experimental Results 184
4.8.1 The Performance of the Learned Cktrees 184
4.8.2 Effect of the Default Operator Choice 188

4.8.3 Comparing Learned Control Rules and Learned Cktrees Performance 189
4.8.4 Reusing Learned Cktrees across Quality Metrics 191
4.9 Summary 192

5 Related Work **193**
5.1 Learning Search-Control for Planning 193
5.2 Interacting with a Human Expert 196
5.3 Planning Approaches to Generating Good Quality Plans 198
5.3.1 Plan Quality and Goal Interactions 199
5.3.2 Decision Theoretical Planning 200
5.3.3 Domain-Dependent Approaches 202
5.3.4 Different Quality Metrics 203

6 Conclusion **205**
6.1 Summary of the Thesis 205
6.2 Future Research Directions 206
6.2.1 Improvements to the Learning Architecture 207
6.2.2 Other Quality Metrics and Other Domains 207
6.2.3 Quality and Planning Efficiency Tradeoffs 208
6.2.4 Other Planning Techniques 209

A The PRODIGY Problem Solver **225**

B The PRODIGY4.0 Process Planning Domain **229**

C Learned Quality-Enhancing Control Rules **241**

D Detailed Experimental Results **249**

List of Figures

1.1 The problem of finding good plans addressed by this thesis. 6

1.2 An example of the effect of goal ordering in plan quality. 8

1.3 The architecture of QUALITY for learning quality search-control knowledge. . 10

2.1 The *face-mill* operator. 23

2.2 (a) Dimensions and sides of a part. (b) An example of a part and tool set-up. . 24

2.3 Control rule that rejects moving the part from another machine if the part is being held by the desired machine <machine> already. 25

2.4 Control rule that expands the machining goals first. 26

2.5 A problem specification in the process planning domain. A problem is specified by the initial state and the goal statement. 27

2.6 (a) Plan obtained by PRODIGY for the problem in Figure 2.5. (b) A better plan, according to the quality metric, for the same problem. 28

3.1 Architecture of QUALITY, that learns control knowledge to improve the quality of plans 33

3.2 Interaction with the expert and checking of the plan obtained. 36

3.3 Obtaining the next operator from the expert. 37

3.4 Testing if the expert-input operator is applicable in the current state. 38

3.5 Lazy inference rule in the process planning domain. 38

3.6 Example of dialog with the interactive plan checker. 40

3.7 A different dialog for the same problem, showing the interactive plan checker in a verbose mode, and the behavior when an operator cannot be executed. . . 41

3.8 Top level procedure to learn quality-enhancing control knowledge. 41

3.9 (a) and(b) Plan trees corresponding to two solutions of different quality for the same problem. (c) Computation of the cost of the plan trees. 45
3.10 Top-level call to the learning mechanism once the plan trees have been built: finding learning opportunities given the plan trees and exploring them to create new control knowledge. 46
3.11 Traversing the plan trees to detect learning opportunities. 48
3.12 An informal description of the explanation that underlies the algorithms presented. 50
3.13 Learning operator and bindings control rules. 51
3.14 Traversing the plan tree to propagate the relevant conditions. 53
3.15 Part of the plan tree for the better quality solution of a process planning problem used to illustrate how **propagate_conditions_up** works, and rules learned from that episode. 54
3.16 Computing a control rule precondition that justifies why g_A had cost 0. 55
3.17 Operationalizing why g_A was a subgoal and had cost 0. 55
3.18 Computing constraints on the type and value of the relevant bindings. 56
3.19 Storing in a data structure the information needed to build an operator and/or bindings control rule. 57
3.20 Building control rules. 57
3.21 Template to create an operator preference control rule. 58
3.22 Template to create a bindings preference control rule. 59
3.23 Beginning of the problem solving trace that obtained the plan of cost 28 (plan (a) of Figure 2.6). 59
3.24 Beginning of the problem solving trace to obtain the better quality plan (plan (b) of Figure 2.6). 60
3.25 Plan trees obtained from the problem solving traces for plans (a) and (b) of Figure 2.6. 61
3.26 Structure built by **create_rule_struct** (Figure 3.19) at the end of the propagation process. 62
3.27 Operator and bindings preference control rules learned from the problem in Figure 2.5. 63
3.28 An artificial domain used to illustrate the need of quality-enhancing goal preference control rules. 64
3.29 Example problem in the domain of Figure 3.28. 65

3.30 Two solutions for the problem in Figure 3.29 with the corresponding traces. . 65
3.31 Partial order corresponding to the better quality solution of Figure 3.30. . . . 66
3.32 Plan trees corresponding to the two solutions in Figure 3.30. 67
3.33 Control rule that makes the correct goal ordering decision in the problem of Figure 3.29. 68
3.34 Algorithm for learning goal preference control rules. 69
3.35 (a) An operator that adds and deletes two instantiations of the same predicate. (b) Partial plan tree where the operator appears. 70
3.36 Template to create a goal preference control rule. 71
3.37 Example problem in the process planning domain to illustrate the learning of goal control rules. 72
3.38 (a) Plan obtained by PRODIGY guided by the current control knowledge. (b) A better plan, according to the quality metric, input by a human expert. Note that both plans have the same length but different quality. 73
3.39 Beginning of the problem solving trace that obtained plan (a) of Figure 3.38. . 73
3.40 Beginning of the problem solving trace to obtain plan (b) of Figure 3.38. . . . 74
3.41 Plan trees obtained from the problem solving traces for plans (a) and (b) of Figure 3.38. 75
3.42 Goal preference control rule learned from the problem in Figure 3.37. 76
3.43 Hierarchy of heuristics to decide among conflicting preferences. 77
3.44 Example to illustrate the use of the proximity heuristic. 79
3.45 Learning preferences among control rules. 81
3.46 An artificial domain used to illustrate conflicting preferences among the learned control rules and how learning helps to break them. 82
3.47 (a) and (b): Two problems in the artificial domain of Figure 3.46 and their respective solutions. (c) and (d): Plan trees constructed from the improved solution traces. (e) and (f): Control rules learned respectively from problems (a) and (b). 84
3.48 A third problem in the artificial domain and PRODIGY4.0's problem solving trace for solving it. At `n22` the control rules learned from the previous problems give conflicting preferences. 85
3.49 Two plans of different quality for the problem in Figure 3.48. 85
3.50 Two bindings control rules learned in the process planning domain and the priority learned among them. 86

3.51 Two goal control rules learned in the process planning domain and the priority learned among them. 87

4.1 Operators in a train and van transportation domain. 101

4.2 An example problem in the transportation domain. 102

4.3 Two plans for the problem in Figure 4.2. 102

4.4 Two plans for a variant of the problem in Figure 4.2 in which *two* objects need to be transported between Monroeville and New Hampton. 103

4.5 Four examples of quality metrics in the transportation domain and their influence in the planner's decision to obtain better plans. 104

4.6 Three control rules that prefer a different alternative for drilling a hole depending on the subparts of the set-up currently available. 106

4.7 A rule learned from a spot hole problem. The top part of the figure shows an informal description of the rule. The bottom part shows the actual rule. 109

4.8 A problem in the process planning domain. The goal is to drill a spot hole on `part5`. The spot drillbit is initially set on the available milling machine. . . . 111

4.9 (a) The plan trees for two solutions to the problem of drilling a spot hole when the appropriate *tool is set on the available milling machine.* (b) The quality metric used in the example. 111

4.10 A new problem in the process planning domain, also to drill a spot hole on `part5`. In this case the part is ready on the milling machine `mm4`. 112

4.11 A sketch of the process of reusing past experience to generate a good plan for the current problem. 113

4.12 Partial view of the control knowledge tree learned from the plan trees for the problem in Figure 4.9. 114

4.13 The data structures used to store each of the types of cktree nodes. 116

4.14 Top-level call to the cktree learning mechanism. 117

4.15 (a) A problem in the processs planning domain. (b) Two solutions for that problem. The plan on the left is the plan initially obtained by the planner. The plan on the right is the improved plan suggested by a human expert. (c) Quality metric used in this example (higher values indicate lower quality). 119

4.16 Planning decisions forced in order to obtain the improved, user-given plan of Figure 4.15 (b). Step 1 of Figure 4.14 chooses the first of those decisions. . . 120

4.17 Plan trees corresponding to the plans of Figure 4.15(b). Some nodes have been omitted for clarity purposes. 120

4.18 Building a new cktree. 121

4.19 Building a cktree operator node. 123

4.20 Building a cktree binding node. 124

4.21 The first two steps of cktree construction in the example: (a) The relevant part of $plantree_A$ (cf Figure 4.17). (b) The first goal, operator, and binding cktree nodes built. 125

4.22 Building a cktree goal node. 126

4.23 Building cktree goal nodes in the example. Pointers from a variable to the nodes that use it are stored in a hash table. 126

4.24 Keeping track of achievement links. This function is called when building a cktree goal node $prec_{ck}$ for a plan tree node $prec_q$ that achieves other goals. . . 128

4.25 Keeping track of achievement links. This function is called when building a cktree goal node $prec_{ck}$ for a plan tree node $prec_q$ that was achieved by other node. 129

4.26 Keeping track of deletion links. This function is called when building a cktree goal node $prec_{ck}$ for a plan tree node $prec_q$ that was deleted by other node. . . 130

4.27 Keeping track of side effects. This function is called when building a binding node b_{ck} corresponding to a plan tree node b_q that had side effects. 130

4.28 The part of the cktree built in the example from $plantree_B$ (top of Figure 4.17) showing the achievement and deletion links created. 131

4.29 Updating an existing cktree. Note the similarity with Figure 4.18. 133

4.30 (a) Initial state and goal of a new example problem. (b) The plan tree corresponding to the lower quality solution. 134

4.31 The cktree of Figure 4.28 updated with the new planning episode described in Figure 4.30. 135

4.32 Keeping track of achievement links that point to other trees. 135

4.33 Learning and updating other cktrees. 136

4.34 A plan tree for a plan to solve goals g_1 and g_2. 137

4.35 Two cktrees for goals g_1 and g_2 learned from the plan tree in Figure 4.34. . . . 137

4.36 The control knowledge tree learned from the plan trees for the problem in Figures 4.8 and 4.9. 138

4.37 Using the cktree previously learned to solve a new problem in which the *the part is set on the milling machine* and *the tool is set on the drill press*. 139

4.38 Using the cktree previously learned to solve a new problem in which the *the part and the tool are set on the drill press.* 140
4.39 Control rules that invoke the control knowledge stored in the cktrees. 142
4.40 A control rule that invokes the cktree matcher. This rule was built automatically when the cktree for `is-tapped` was learned. 142
4.41 Definition of meta-predicate `current-goal-and-pref-op`. 143
4.42 The basic cktree matching function. The quality metric is used in Step 28. . . 145
4.43 Generating instantiations for a cktree binding node. 146
4.44 Maintaining the achieved and deleted goals. 148
4.45 Estimating the cost of achieving a universally quantified precondition. 149
4.46 Matching the available alternatives. 151
4.47 Marking the cktree nodes whose estimated cost needs to be recomputed. . . . 153
4.48 The `has-hole` cktree of Figure 4.28. This figure shows the contents of the hash table of pointers from the variables to the nodes that use them. 154
4.49 Using the cktree to estimate the cost of the first alternative. 155
4.50 Reusing previous estimates to estimate the cost of the second alternative. . . . 156
4.51 Two plan trees to solve goals g_1 and g_2. 159
4.52 Two cktrees for goals g_1 and g_2 learned from previous planning experience. . . 160
4.53 Illustration of the cktree matching process, that is, the cost estimates found and the parts of the cktree explored, for op_1^A, the first operator alternative for g_1. . 160
4.54 Illustration of the cktree matching process for op_1^B, the second operator alternative for g_1. 161
4.55 The definition of **current-goal-and-pref-op** revisited to explore the cktrees of other goals. 162
4.56 Updating the estimate for an alternative by considering other goals. 162
4.57 Traversing other cktrees to update the estimate for the current alternative. . . . 164
4.58 Checking whether the goal corresponding to a node will actually become a goal in the current problem. 164
4.59 Traversing the same cktree again with an instantiation corresponding to a different goal. 165
4.60 Using the `has-spot` cktree of Figure 4.36 as control knowledge for the problem of drilling two spot holes on the same side of a part. 166

4.61 Using the `has-spot` cktree to estimate the cost of the second alternative, *drill-in-milling-machine*. 166
4.62 Initial solution and problem solving trace obtained by the planner for the problem in Figure 4.2. 169
4.63 The solution provided by the human expert and the trace generated in order to obtain that plan. 170
4.64 Control knowledge tree learned for the problem in Figure 4.2. 171
4.65 A goal preference control rule for the transportation domain. 172
4.66 Analyzing the effect of operator ordering. 189

5.1 A taxonomy of quality metrics. 204

A.1 A skeleton of PRODIGY4.0's nonlinear planning algorithm. 226
A.2 The learning modules in the PRODIGY architecture (from [Veloso *et al.*, 1995]). 228

B.1 The type hierarchy for the process planning domain. 230

List of Tables

2.1 A quality metric for the quality of the plans in the process planning domain. . 27

3.1 The quality metric used in the experiments described. 91
3.2 Summary of the training phase. The numbers shown were computed for the 19 training problems in which learning was actually invoked. 92
3.3 Experimental results for the training phase. 93
3.4 Improvement on the quality of the plans obtained for 180 randomly-generated problems in the process planning domain. 94
3.5 Effect of the learned control knowledge in the planning CPU time and in the number of nodes searched. 95

4.1 Operators in an example artificial domain. 98
4.2 An example quality metric for the transportation domain. 101
4.3 Quality of different plans for two problems (moving one and two packages) in the transportation domain. 102
4.4 Quality of plans for four problems in the process planning domain. 105
4.5 Comparative quality of the plans to drill one and two spot holes depending of the machine used. 167
4.6 Cost of two alternative plans for the problem of Figure 4.2 for several quality metrics. 171
4.7 Cost of two alternative plans for the 2-package problem for several quality metrics. The metrics differ on the cost of the *drive-van* operator. 172
4.8 The quality metric used in the experiments described in this section. 184
4.9 Summary of the training phase. The numbers shown were computed for the 13 training problems in which learning was actually invoked. 186
4.10 Experimental results for the training phase. 187

4.11 Improvement on the quality of the plans for 180 randomly-generated problems obtained by using the learned cktrees. 187

4.12 Effect of the learned control knowledge in the planning CPU time and in the number of nodes searched. 188

4.13 An empirical comparison of the two learning approaches: learning control rules and learning control trees. 190

4.14 An experiment to factor out the effect of the learned goal-preference control rules from the effect of the cktrees. 191

4.15 Effects in quality of reusing cktrees learned for different metrics. 191

Chapter 1

Introduction

The title of this thesis, *Learning Search Control Knowledge to Improve Plan Quality*, captures three basic points that have served as motivation for our line of research. The first one is the *importance of plan quality* in planning systems. The second one is the use of *search control knowledge to generate high quality plans*. The third one is the use of *machine learning* as the vehicle to automatically acquire quality-enhancing search control knowledge.

Much research on artificial intelligence (AI) planning so far has concentrated on methods for constructing sound and complete planners that find a *satisficing* solution, and on how to find such solution in an efficient way.[1] The definition of the *quality* of the planner's solution implicit in past planning work is a rather impoverished one: a plan is *good* if it achieves the specified goal(s). As planners are applied to more interesting, realistic tasks, plan quality becomes a crucial factor. Chien *et al* [1994] report that one of the issues hindering the efforts to field planning applications is the ability to represent and reason about plan quality. Hayes [1995b] points that if a "knowledge-based system produces solutions of lower sophistication and quality than the user can produce on his or her own, the user may consider the system to be a hindrance and try to work around it instead of with it". Generating production-quality plans is an essential element in transforming planners from research tools into real-world applications. This thesis addresses the problem of *producing high quality plans*.

Controlling search is a central issue in many AI systems and in particular in classical planners. Unguided search can be prohibitely slow, even in toy application domains. Domain-specific search-control knowledge is required to capture heuristics or strategies. This search-control knowledge serves a double purpose: to increase the efficiency of the planner's search, so that plans are found faster (i.e. more efficiently, by pruning the search space), and to improve the quality of the plans that are found. There is usually more than one plan for a problem, but only

[1] For a good but somewhat outdated survey on planning techniques see [Hendler *et al.*, 1990]. Most recent work on planning can be found in [Hendler, 1992, Hammond, 1994].

the first one that is found will be returned. By directing the problem solver's attention along a particular path, control knowledge can express preferences for plans that are qualitatively better (e.g., more reliable, less costly to execute, etc.). This thesis explores the *use and automated acquisition of search-control knowledge to generate better plans*. This is different from other efforts that improve the quality of the plans by post-facto modification [Karinthi *et al.*, 1992, Foulser *et al.*, 1992].

In spite of advances in knowledge acquisition techniques and tools, acquiring knowledge, and in particular acquiring control knowledge, is still a major bottleneck in building complex planning domains. Previous research has shown the effectiveness of a variety of machine learning methods to capture problem solving heuristics expressed in a number of representation formalisms. In particular, most of the research to date in the application of machine learning to planning systems has focused on *planning efficiency*, that is, on acquiring problem solving strategies that control search in order to make problem solving more efficient. This area of research has been termed "speed-up learning" [Mitchell *et al.*, 1986, Minton *et al.*, 1989, Tadepalli, 1989, Etzioni, 1990, Pérez and Etzioni, 1992, Knoblock, 1994, Veloso, 1994, Gratch *et al.*, 1993]. This thesis looks instead at the application of machine learning to acquire strategies that lead a planner towards improving *plan quality* and describes an architecture to *learn quality-enhancing search control knowledge* from a combination of planning experience and interaction with a human expert.

This thesis builds on the PRODIGY4.0 planner [Veloso *et al.*, 1995, Carbonell *et al.*, 1992]. PRODIGY4.0 is a means-ends analysis nonlinear planner. PRODIGY2.0 [Minton *et al.*, 1989], PRODIGY4.0's precursor, was designed as a testbed for learning in the context of planning. The PRODIGY architecture[2] provides an expressive language for representing independently both domain knowledge and search-control knowledge. The planner is given a specification of the domain and can become proficient in it by forming (*learning*) its own control knowledge and/or refining its domain knowledge through the analysis of the domain and of its problem-solving experience. The PRODIGY architecture is well suited as a vehicle for our investigation because it has clear explicit decision points that permit the infusion of automatically or manually acquired control knowledge to improve plan quality. (Appendix A describes PRODIGY4.0 in the context of the PRODIGY architecture and overviews the learning modules.)

[2]Throughout this document PRODIGY4.0 refers to the most recent nonlinear planner of the PRODIGY architecture. PRODIGY refers to the whole architecture, which includes the planner and learning modules described in Appendix A. In some cases PRODIGY is also used when describing features shared by PRODIGY4.0 and the previous PRODIGY planners (PRODIGY2.0 and NOLIMIT).

1.1 Machine Learning for Planning Systems

Although both AI planning and machine learning have been the subject of much research recently, not many projects have integrated both areas. However this interaction can be very beneficial. There are three main types of goals for learners in the context of problem solving systems, namely: domain goals, planning efficiency goals, and plan quality or plan efficiency goals, all of which we define below. In the case of intelligent autonomous agents the learner's goals may be a combination of goals of these three types.

- Learning driven by domain goals: Learning is prompted by a lack of knowledge on the part of the planner about the domain in which problems are being solved. The planner's representation of the domain (available actions and preconditions and effects of each action) may be incomplete or inaccurate. This lack of knowledge makes the planner fail to solve certain problems, and a procedure is needed to acquire more knowledge or correct the existing one. Research in knowledge engineering has addressed this problem at different levels of automation. In the case of planning systems several research efforts have focused in fully automating this process [Shen, 1989, Gil, 1992, Huffman *et al.*, 1993, Wang, 1995, desJardins, 1994]. The type of learning performed by all these systems is primarily inductive and goes beyond the reformulation of the planner's initial knowledge.

- Learning driven by planning efficiency goals, or *speed-up* learning: In this case the goal of learning is to improve the efficiency of the planning process. One can view such systems as searching the space of knowledge representations in order to find more effective ways of expressing the knowledge that the system already implicitly has. After learning, if no resource bound existed at problem solving time, the planner would be able to solve, more efficiently, the same problems it was able to solve before learning. However in practice resource bounds do exist, and the effect of the learned knowledge is to increase the *solvability horizon* [Veloso, 1994, Iba, 1993]: many problems that could not be solved within a particular resource bound are solved using the learned knowledge within the given bound or an even smaller one. Several techniques have been used in this framework, including learning search control knowledge [Mitchell *et al.*, 1986, Minton, 1988, Tadepalli, 1990, Etzioni, 1990, Pérez and Etzioni, 1992, Leckie and Zukerman, 1993, Borrajo and Veloso, 1994a, Katukam and Kambhampati, 1994], macro-operators [Fikes *et al.*, 1972, Korf, 1985, Cheng and Carbonell, 1986, Segre *et al.*, 1993], chunking [Laird *et al.*, 1986], abstraction hierarchies [Knoblock, 1994, Christensen, 1990], and problem-solving cases [Veloso, 1994].

- Learning driven by plan quality, or plan efficiency, goals: The goal of learning plan quality is to acquire heuristics that guide the planner at generation time to produce plans

of good quality automatically. Not much research has addressed this kind of performance goal [Ruby and Kibler, 1990, Iwamoto, 1994, Borrajo and Veloso, 1994b].

Most of the research up to date has concentrated on the first two types of goals. This thesis concentrates on the third one.

1.2 Measuring Plan Quality

Plan quality metrics can be classified in three large groups ([Pérez and Carbonell, 1993] contains a detailed taxonomy, that we reproduce in Section 5.3.4):

- Execution cost. Some of the factors that affect a plan's execution cost can be computed by summing the costs of all the steps or operators in the plan, that is $C_{total} = \Sigma c_i$ where C_{total} is the total cost of executing the plan and c_i is the cost for each operator in the plan. c_i can be the operator execution time, the cost of the resources used by the step, or 1 if the measure is simply the length of the plan or total number of actions. Several factors that influence a plan's execution cost are the execution time, the material resources, or the agent skill requirements (which refers to the extent to which an agent can perform an action; plans with less agent skill requirements are typically less expensive).
- Plan robustness or ability to respond well under changing or uncertain conditions [Blythe, 1994, Kushmerick *et al.*, 1994].
- Other factors that capture the satisfaction of the client with the plan itself (for example the accuracy of the result, the comfort it provides to the user, or marketing concerns [Kibler, 1993]). In some cases these are hard to quantify.

This thesis addresses execution cost as the plan evaluation metric, and the automated acquisition of control knowledge is driven by the desire to minimize execution cost in future planning problems.

1.3 The Problem

Knowledge about plan quality in a domain D comes in two forms: (a) a post-facto quality metric $Q_D(P)$ that computes the quality (e.g. the execution cost) of plan P, and (b) planning-time decision-control knowledge used to guide the planner towards producing higher-quality plans. The first kind of knowledge, $Q_D(P)$, is non-operational; it cannot be brought into play until

after a plan is produced. Yet, cost functions is exactly the kind of quality knowledge typically available, in contrast to the far more complex operational decision-time knowledge. Hence, automatically acquiring the second kind of knowledge from the first is a very useful, if quite difficult endeavor. In essence, learning operational (planning-time) quality control knowledge can be seen as a *translation* problem of domain knowledge D and quality metric Q_D into runtime decision control guidance

$$\textit{Quality: } D \times Q_D \rightarrow \textit{Decision-Control}_{D,Q_D}$$

And the *full automation of the quality mapping problem* is the ultimate objective of this thesis. In practice, the mapping is learned incrementally with experience.

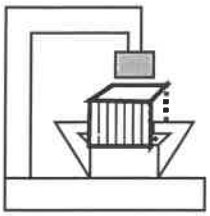
1.3.1 Planning Decisions and Plan Quality

Planning is a constructive, incremental decision-making process. Each particular planner faces multiple decisions when generating a complete plan to solve a given problem. The types of those decisions vary from planner to planner, but they generally include choosing actions (operators) to achieve goals, choosing objects in the world with which to instantiate those operators, and choosing orderings on how the problem goals are achieved. The decisions made at those points have an effect on both planning *efficiency* and plan *quality*. A few planners, in particular PRODIGY, allow planning-time decision-control knowledge to guide those decisions. Such control knowledge can be *hand-coded* or automatically acquired (*learned*) from planning experience. The problem addressed by this thesis is how to *learn* control knowledge to guide the planner to generate high-quality plans.

In this section we present two examples to illustrate how control knowledge is relevant to plan quality and how it is distinct from other kinds of control knowledge relevant to planning efficiency. Figure 1.1 summarizes the problem of finding good plans with an example in a process planning domain. Figure 1.1(a) describes the example problem as it is posed to the planner: in the initial state a square block of raw material is set on the table of a milling machine (*milling-machine3*); the goal is to have a part of height 2 with a hole in one of its sides (side1). Three distinct metrics of plan quality in this domain are displayed in the table of Figure 1.1(b). In this example plan quality corresponds to plan execution cost and the metrics assign a fixed cost to each operator (lower values are higher quality). However those costs in general may depend on the particular operator instantiation (bindings), as shown in other examples throughout the thesis. The total quality (cost) of a plan P is computed by adding the cost of the plan operators. Figure 1.1(c) presents two plans for the problem in (a). **Plan 1** uses a single machine, the milling machine on which the part is set, to both reduce the part height (using the *face-mill* operator) and drill a hole (using a *drill* operator). **Plan 2** starts by cutting the part on the milling machine and then chooses the drill press to make the hole. In order to do that the part and the only holding device available must be moved to the drill press. Figure 1.1(d) shows the quality values for each of the plans and the quality metrics. Values in

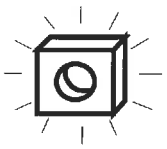
bold face indicate the cost of the better plan in each case: Plan 1 is better under metrics Q_1 and Q_3. Plan 2 is better under metric Q_2.

Initial state



Milling-machine3

Goal



has-hole side1
size-of height 2

(a)

Three distinct quality metrics

	q_1	q_2	q_3
Milling machine operators	5	7	7
Drill Press operators	5	1	1
Part set-up operators	1	1	2
Tool set-up operators	1	1	1

Plan quality $Q_i(P) = \sum_{op_j \in P} q_i(op_j)$

(b)

Two plans for the problem

Plan 1	Plan 2
face-mill switch tool drill in milling machine	face-mill release part move holding device to drill press move part to drill press hold part put-tool drill in drill press

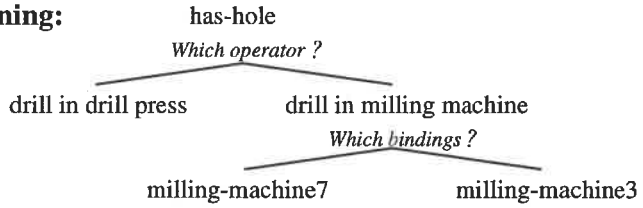
(c)

The quality of the plans

	Q_1	Q_2	Q_3
Plan 1	**11**	15	**15**
Plan 2	15	**13**	17

(d)

During planning:



(e)

Figure 1.1: The problem of finding good plans addressed by this thesis.

Figure 1.1 (e) shows the decisions that the planner confronts at problem solving time when given the example problem. It must make the appropriate decisions regarding operator (drill in the milling machine or drill in the drill press) and operator instantiation, or bindings,[3] (e.g. which of the available milling machines) in order to obtain the better plan. Those decisions are different depending on the chosen quality metric (since the quality values of the plans obtained

[3]We use the terms *bindings* and operator *instantiation* indistinctly, unless otherwise indicated by the context. We also use *planning* (or *plan*) and *problem solving* (or *solution*) interchangeably.

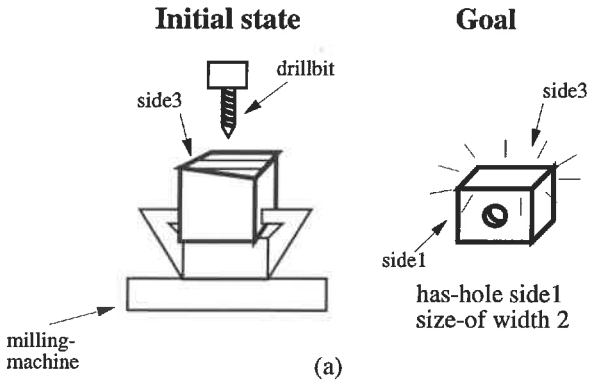
by making the decisions are different). The example shown is a simple one. There may be other goals to achieve, and thus decisions to make about which goal to work on next, and also interactions among goals. There may be other alternative operations to achieve the problem goals, and many resources (tools, machines, different types of raw materials) that must be selected by the planner.

Thus, the problem we address in this thesis is how to automatically acquire search-control knowledge that will guide the planner to make the decisions leading to the better plans, for each particular domain-specific quality metric. Note that such control knowledge is orthogonal to planning efficiency control knowledge for early pruning of choices that are guaranteed to lead to failure paths. (Such control knowledge has been the target of previous research [Minton, 1988, Etzioni, 1990, Pérez and Etzioni, 1992, Katukam and Kambhampati, 1994]). In the case of quality control knowledge the alternatives decided upon may all lead to success, that is, to plans for the problem, but those plans have different quality.

In addition to operator and operator instantiation choices, the decision about which goal to work on next in the context of the PRODIGY4.0 planner is relevant to plan quality.

Planning goals rarely occur in isolation and the interactions between conjunctive goals have an effect in the quality of the plans that solve them [Wilensky, 1983, Pollack, 1991, Nau, 1993]. In [Pérez and Veloso, 1993] we argued for a distinction between *explicit goal interactions* and *quality goal interactions*. Explicit goal interactions are represented as part of the domain knowledge in terms of preconditions and effects of the operators. In the simpler case, given two goals g_1 and g_2 achieved respectively by operators op_1 and op_2, if op_1 deletes g_2 then goals g_1 and g_2 interact because there is a strict ordering between op_1 and op_2. Goal interactions of this type have been extensively analyzed in the planning literature [Chapman, 1987, Barrett and Weld, 1994]. In least-commitment planners threat-solving mechanisms or critics take care of these goal interactions by establishing ordering constraints among the conflicting goals. In the case of PRODIGY goal preference control knowledge is automatically acquired to deal effectively (in the sense of problem solving effort) with this kind of goal interactions [Minton, 1988, Etzioni, 1990, Pérez and Etzioni, 1992, Veloso, 1994, Borrajo and Veloso, 1994a, de Silva, 1995].

On the other hand, quality goal interactions are not directly related to successes and failures. As a particular problem may be solved by many different plans, quality goal interactions may arise as the result of the particular problem solving search path explored. For example, in a process planning domain, when two identical machines are available to achieve two goals, these goals may interact, if the problem solver chooses to use just one machine to achieve both goals, as it will have to wait for the machine to be idle. If the problem solver uses the two machines instead of just one, then the goals do not interact in this particular plan [Veloso, 1994]. Similar examples occur in many domains. These interactions are related to plan quality. Whether one alternative is better than the other depends on the particular quality metric used for the domain.



A quality metric

	q_1
drill, face-mill	2
put-tool, remove-tool	1
hold-part, release-part	4

Plan quality $Q_1(P) = \sum_{op_j \in P} q_1(op_j)$

(b)

Two plans for the problem

Plan 1	Plan 2
release-part side3 hold-part side1 up drill hole remove-tool drillbit put-tool milling-cutter release-part side1 hold-part side3 up face-mill	remove-tool drillbit put-tool milling-cutter face-mill remove-tool milling-cutter put-tool drillbit release-part side3 hold-part side1 drill hole

(c)

The quality of the plans

	Q_1
Plan 1	22
Plan 2	16

(d)

Figure 1.2: An example of the effect of goal ordering in plan quality.

Figure 1.2 illustrates another case of goal interactions related to plan quality. The goal of the problem shown in (a) is to have a part of width 2 with a hole on side1. Assume that in order to reduce a part's width the part must be face-milled with its side3 facing up (i.e. facing the tool). In the initial state the drill bit to make the hole is ready (a precondition of drilling a hole), but the part is set with its side 3 facing up (a precondition of face-milling the part's width). Both goals (part's width and hole) interact because in order to achieve each one, the planner must delete a precondition of the other one already present in the initial state. For example, in order to face-mill the part, the milling cutter must be in the machine's tool holder, and achieving that requires removing the drill bit. Figure 1.2(c) shows two plans for the problem, corresponding to two different orders of achieving the top-level goals. Figure 1.2(b) shows a quality metric for the domain. As in our first example, quality refers to plan execution cost (lower values indicate higher quality). With this metric, switching the machine tool is cheaper, as it is done automatically in semi-automated machining centers, than turning the part around, which requires a human operator. Although the two plans shown for this problem have the same length, they have different quality according to such metric (Figure 1.2(d)), because the

cost of reachieving the preconditions deleted from the initial state in each plan is different.[4]

This example illustrates several points. First, plan length is not necessarily a good measure of plan quality. Second, goal achievement ordering is another planning decision relevant to plan quality, in addition to operator and operator instantiation decisions (shown in the first example). Third, how the goal interactions should be solved to obtain a good plan may depend on the metric of plan quality for the domain. The control knowledge to guide the planner to solve these quality-related interactions is harder to learn automatically than in the case of explicit goal interactions, because the domain theory D, i.e. the set of operators, does not encode these quality criteria.

In these two examples we have shown how plan quality is affected by operator, operator instantiation, and goal achievement ordering decisions. Any system that treats planning as search is faced with decisions, or commitments, to make during search. The type of decisions varies between planning algorithms, in particular between state-space based planners (such as PRODIGY4.0) and plan-space based planners. For example, SNLP's descendants [McAllester and Rosenblitt, 1991, Penberthy and Weld, 1992] must choose how to solve threats and open conditions, by adding new causal links (ordering and binding constraints) to the existing plan steps, or by adding new steps. Making the correct decisions that will lead the planner to good plans is an issue in any planner, and one that has not received much attention in the literature.

1.4 The Thesis

This thesis addresses the quality mapping problem. Given a general-purpose state-space planner, a domain theory, a domain-specific metric of the quality of the plans, and problems in that domain which provide planning experience, the techniques described here automatically acquire operational, planning-time search-control knowledge that effectively improves the quality of the plans generated by a planning system. In addition, the system (optionally) benefits from the interaction with a human expert in the application domain who suggests improvements to the plans at the operator (plan step) level. The learning techniques we have developed are independent of the application domain. We have fully implemented them in the QUALITY architecture and tested it in several domains. This section gives an overview of QUALITY.

1.4.1 Overview of the Approach

Figure 1.3 shows the architecture of QUALITY. QUALITY is implemented on top of PRODIGY4.0, the most recent nonlinear planner of the PRODIGY architecture. QUALITY is given a domain

[4]This example is elaborated in Section 3.10.

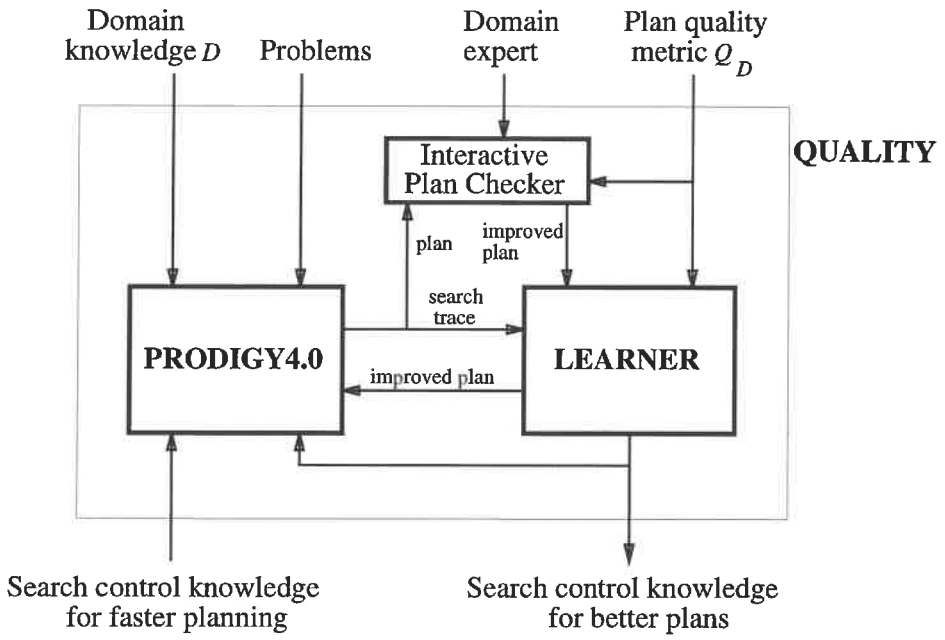


Figure 1.3: The architecture of QUALITY for learning quality search-control knowledge.

theory D (operators, inference rules, and a hierarchy of the types of objects in the domain) and a domain-specific function Q_D that evaluates the quality of the plans produced. It is also given problems to solve in that domain; solving those problems provides QUALITY with planning experience in the form of a search trace, i.e. the sequence of decisions that the planner made during a problem-solving episode.[5] QUALITY analyzes problem-solving experience by comparing the search trace obtained by solving a problem given the current control knowledge, and another search trace corresponding to a better plan for the same problem (*better* according to the quality metric) obtained as explained below. QUALITY analyzes the differences between the sequence of decisions that the planner made initially and the ones that should have been made to generate the better quality plan. The learner interprets these differences as learning opportunities and identifies and generalizes the conditions under which the individual planning choices will lead to the desired final plan. In this way QUALITY compiles knowledge to control the decision making process in new similar planning situations to generate plans of better quality.

In order to obtain the search trace corresponding to the better plan QUALITY can either function autonomously or interact with a human expert. In its autonomous mode, once PRODIGY4.0 has come up with the initial plan, QUALITY asks it to further explore the search space finding plans of increasing quality as determined by the available quality metric Q_D, until the space

[5] A design principle of the PRODIGY architecture is to make that problem solving information available at any time to allow introspection by a variety of learning methods.

is exhausted or some typically large resource bound is met. The best plan found within that bound and its corresponding search trace are passed to the learner. If the best plan is different from the initial plan learning is triggered.

Because of the large search spaces in complex domains, finding good enough plans from which to learn can be computationally very expensive. Optimization by exhaustive search is exponential in the length of the optimal plan (which may not be the shortest plan). On the other hand, human expertise is available in many domains, and can be advantageously used to help the system find useful strategies to obtain good plans. In its interactive mode, QUALITY asks a human for a better plan and then calls PRODIGY4.0 to produce a search trace that leads to that plan. QUALITY assumes that the expert's model of plan quality and the quality metric Q_D are consistent. In particular if the expert's plan is worse than the initial one, the expert's plan is rejected. The interaction with the expert is the task of the *Interactive Plan Checker* in the figure. This interaction occurs at the level of plan steps, i.e. concrete actions in the plan, which correspond to instantiated operators, and not at the level of the full range of problem-solving time decisions. Thus the expert needs to be familiar with the available operators, their parameters, preconditions, and effects. QUALITY's assumption (and design goal) is that the human is an expert in the application domain but can remain oblivious to the planner's algorithm and control knowledge representation language. This relieves him/her of understanding PRODIGY4.0's search procedure and control knowledge encoding, a very important feature to interact productively with domain experts who are not knowledge engineers.

When we allow the system to learn through interaction with a human, we must consider the gap between the expert's view of the world and efficient search control knowledge that captures such expertise. Ideally the expert should be able to provide advice using terms about the specific application domain and to ignore the details of the internal representation and the planning algorithm. On the other hand, the planner's control knowledge typically refers to problem solving states (e.g. operators that have been expanded, goals pending exploration) in which the control decision applies. This kind of control knowledge is hard to give by such an expert. However s/he can easily provide a plan for the problem and, even better, critique and improve the plan obtained by the planner by suggesting additions, deletions, or modifications of plan steps. This advice, if consistent with the quality metric, is then *operationalized* by the learning system [Mostow, 1983], that is, translated into knowledge usable efficiently during problem solving. Thus, the goal of learning problem solving expertise can be seen as translating a non-operational domain theory into an operational one [Tadepalli, 1990]. In the previous section we referred to it as the *quality mapping problem.* In our case the domain theory consists of the description of the domain (planning operators, inference rules, and type hierarchy) and the domain-specific quality metric. The goal of learning is to translate it into operational planning-time search-control knowledge based on the planner's experience and the advice of a human expert.

We have developed two different domain-independent learning mechanisms within QUALITY to efficiently acquire quality control knowledge. They differ in the language used to represent the learned knowledge, in the algorithms themselves, and in the kinds of quality metrics for which they are best suited:

- *Learning search-control rules.*
  The first mechanism learns control knowledge in the form of control rules, in particular of PRODIGY's prefer control rules [Minton *et al.*, 1989, Carbonell *et al.*, 1992]. These are productions (*if-then* rules) that indicate a preferred choice of an operator to achieve the current goal, choice of bindings or instantiation for the chosen operator, or choice of the next goal to work on among the set of goals still pending. These decisions correspond to some of PRODIGY's decision points. The two examples in the previous section illustrated how those choices are relevant to obtaining good plans. Chapter 3 describes the domain-independent algorithms that learn quality-enhancing domain-specific control-rules.

  Previous algorithms developed in the context of the PRODIGY architecture learned control rules for PRODIGY2.0, the initial, linear planner of PRODIGY, with the goal of *planning efficiency* [Minton, 1988, Etzioni, 1990, Pérez and Etzioni, 1992]. QUALITY focuses on *plan quality*. The system described in [Iwamoto, 1994] learns control rules for optimization of certain classes of problems, primarily in LSI circuit layout. HAMLET [Borrajo and Veloso, 1994b] uses a combination of induction and bounded explanation to learn control rules that increase planning efficiency and improve plan quality (in particular reduce plan length). These two approaches have been developed independently of and concurrently with QUALITY and are described in Chapter 5.

- *Learning control knowledge trees.*
  In the second learning mechanism within QUALITY the learned control knowledge is represented using a formalism that we call *control knowledge trees*. The motivation for this new representation is that, in general, complex quality metrics require reasoning about tradeoffs and taking a global view of the plan to make a set of globally optimal choices. Acquiring control rules that apply at individual decision points may prove insufficient. Instead, a more globally-scoped method is required. Control knowledge trees provide a more global view of the planning decisions and are used, together with the quality metric, to estimate the quality of each available alternative at a given planning decision point. The quality metric is parameterized and therefore the learned control knowledge trees are robust to changes in the metric. Currently they provide guidance for operator and bindings decisions, but not for goal ordering decisions. We have designed algorithms to automatically build control knowledge trees from planning experience and to use them at planning time. These algorithms and the motivation for the new representation are the subject of Chapter 4.

The learned quality-enhancing *control rules* provide effective guidance when the quality metric does not require reasoning about complex global tradeoffs. They are highly operational and

are efficiently used at planning time. In our empirical evaluation we have found that they do not reduce planning efficiency but increase it in many cases (when the higher-quality plans are also shorter in length). On the negative side, as the learned rules are domain and quality metric specific, if the metric changes they are invalidated and must be relearned. The performance in improving plan quality of the learned *control knowledge trees* is equivalent to that of control rules for simpler non-interacting situations, and superior for more complex interactions and tradeoff situations. However using control knowledge trees is computationally more expensive and may reduce planning efficiency. In addition control knowledge trees do not provide goal ordering control knowledge. In some of our experiments we have used them together with the goal preference rules learned by the first method achieving a synergistic effect. An important advantage of control knowledge trees is their robustness to changes in the quality metric.

We have fully implemented all the algorithms described and evaluated their performance in a complex process-planning domain, in which they lead to significant plan quality improvements. An additional small transportation domain is also used to test the performance of control knowledge trees. Each of Chapters 3 and 4 finishes with the results of the empirical evaluation of the algorithms described and a discussion of their characteristics and limitations.

1.4.2 Scientific Contributions

The scientific contributions of this dissertation include:

- Focus on *plan quality* instead of just planning efficiency, because generating quality plans is an essential element in transforming planners from research tools into real-world applications.
- A demonstration that search-control knowledge can guide a planner's decisions during problem solving towards better plans according to externally-defined, domain-specific, quality metrics. The thesis analyzes which planning decisions are relevant to plan quality and how the control knowledge to guide them is represented for a particular planning algorithm.
- The design of a domain-independent algorithm for automatically acquiring domain-dependent quality-enhancing search-control rules from problem solving experience. The learned rules indicate preferred goal, operator, and operator instantiation alternatives and take advantage of the rich control knowledge representation language in the PRODIGY planner.
- A new formalism for representing search control knowledge, control knowledge trees which provides operator and operator instantiation guidance. They allow the explicit use of the quality metric during planning, as the metric is parameterized, and can be reused

across different quality metrics. Control knowledge trees are automatically built from planning episodes by QUALITY and used in subsequent planning.

- A demonstration of how the learning algorithms additionally benefit from the (optional) interaction with a human user who is an expert in the application domain but remains ignorant of the planner's algorithm and control-knowledge representation language.

- Full implementation and empirical demonstration of all the algorithms mentioned. The thesis presents results on their quality improvement performance, as well as in their effect in problem solving time, across different quality metrics. The results show that planning efficiency not only does not degrade considerably by the use of the learned knowledge, but can increase in some cases. The learned knowledge provides significant improvements in plan quality and can often achieve near-optimal performance across different quality metrics.

1.5 A Reader's Guide to the Thesis

This section describes the organization of the thesis. Chapter 1 has motivated this thesis work in the context of research in planning and machine learning. It has introduced the problem, generating good quality plans, stated the overall approach of the thesis to the problem, and enumerated the scientific contributions of the thesis. The algorithms described and implemented throughout the thesis are independent of the application domain, but are illustrated with examples from two domains: a process planning domain, and a small transportation domain. Chapter 2 describes the first domain in detail, discussing the characteristics of its implementation as a large, complex domain in PRODIGY4.0, and the impact of the planner's decision in process plan quality.

The core of the thesis is divided in two major parts, which correspond to two approaches developed for learning quality-enhancing search-control knowledge. Chapter 3 describes the algorithms we have developed for learning search-control rules that indicate operator, bindings, and goal preferences. The chapter includes a discussion of the characteristics and suitability of the approach and an empirical analysis of its performance.

Chapter 4 introduces control knowledge trees as a new formalism to represent control knowledge for generating good quality plans. First, the new formalism is motivated and compared with search-control rules. Then algorithms are described in detail for automatically acquiring control knowledge trees from experience and using them at planning time. This chapter also ends with a discussion of the approach and an empirical analysis, which includes a comparison with learning control rules.

Chapter 5 compares and contrasts this thesis work with other work related to representing and reasoning about plan quality in planning systems, and to learning search-control knowledge and problem-solving expertise.

The final chapter restates the contributions of this thesis and discusses some future research directions.

Chapter 2

The Process Planning Domain

This chapter describes the process planning domain which is used throughout the thesis to demonstrate our methods for learning control knowledge to improve plan quality and to provide empirical evaluation. First we introduce the task of process planning as one of the stages of production manufacturing and analyze the importance of plan quality in process planning. Then we describe its implementation as a complex domain for the PRODIGY planner and introduce an example problem in this domain.[1]

2.1 What is Process Planning?

Process planning is one of the intermediate steps of production manufacturing [Doyle, 1985, Hayes, 1990]. The first stage in preparation for manufacturing is engineering, which entails building a model that satisfies a set of specifications, selecting the proper materials, ascertaining the proportions and desired physical properties, and configuring parts into larger assemblies. In the next step process plans are delineated. This includes listing the steps or operations, i.e. the process plans, and designating the machines, equipment, and tools needed and performance expected. A process plan, for instance, may require cutting metal stock, machining it into a desired shape, drilling holes for bolt-assembly, and polishing its surface. On the basis of the process plans, operation routines are planned in detail. This phase is called production planning. The last phase is one of scheduling multiple process plans on available machines and allocating time, resources, and human operators. The parts are then manufactured according to the production plans and the master schedule.

Current research on automation of these manufacturing processes includes Computer-Aided Design (CAD) aids, assembly automation, process planning tools, and factory scheduling

[1] Parts of this section are taken from [Gil and Pérez, 1994].

systems. Interest in these types of automation is rapidly increasing due to the need to lower manufacturing costs, a growing scarcity of experts, and a desire to make customized products widely available [Gil, 1991]. Customization requires "on-the-fly" process planning for each differentiatied product version. The process planning task in particular is complex because there are alternative processes for an operation, alternative parameters for each process, and many interactions between processes. Also, the nature of the planning goals is diverse; some deal with materials, sizes, finishes, and features, such as different types of holes and cuts; others relate to indirect issues such as tolerances for measures or tool conservation. Finally, good process plans should minimize resource consumption and execution time. All these factors make process planning an interesting application domain for AI planning techniques.

Most of the research on process planning has focused on domain-dependent algorithms (see [Chang and Wysk, 1985] for an overview) and pursued several approaches, including generative, variant, and hierarchical techniques [Nau and Chang, 1985, Descotte and Latombe, 1985, Gil and Pérez, 1994]. *Generative* approaches combine elementary process planning operations into larger sequences in order to produce the final plan. *Variant* approaches retrieve complete plans from a plan library and adapt them to the current problem specification. Other systems use *hierarchical planning* techniques where generic plans are pre-specified but must be specialized, instantiated, or interwoven. Some of these systems use AI techniques [Hayes, 1990, Descotte and Latombe, 1985, Nau, 1987] including general-purpose problem solvers coupled with special-purpose systems [Kambhampati *et al.*, 1993].

Research in the PRODIGY framework has put together expressive general-purpose planners with a variety of machine learning techniques to improve the planner's performance. The approaches to process planning mentioned above have direct correlates in PRODIGY's domain-independent planning and learning techniques. In our work we have focused on an implementation in which the planner finds solutions by reasoning from first principles in a generative manner, and search control rules guide problem solving decisions. Derivational analogy [Veloso, 1994, Veloso and Carbonell, 1993] reuses planning episodes learned from past experience, adapting them to the current problem. This technique is a generalization of the variant approach. ALPINE automatically generates abstraction hierarchies which are used by PRODIGY's hierarchical planner [Knoblock, 1994, Blythe and Veloso, 1992], which is comparable to other hierarchical planners but does not require pre-specified generic plans.

Of all the processes involved in process planning, we have concentrated on the machining, joining, and finishing operations. Machining refers to the art of creating parts, usually metal, by carving raw material with power tools such as bandsaws, lathes, milling machines, and drill presses, and using processes such as drilling, milling, turning, etc [Hayes, 1990]. Joining and assembly processes include soldering, welding and bolting. Finishing processes change the surface properties of a part, including cleaning it or removing burrs [Gil, 1991]. Computer numeric controlled machines (also called CNC machining centers) are used to perform many of these processes. These machines have a movable work table, a mechanical tool changer,

and a rotating magazine of tools [Hayes, 1990]. Section 2.3 describes our model of process planning in PRODIGY4.0.

2.2 Plan Quality in the Process Planning Domain

Plan quality is crucial in process planning to minimize both resource consumption and execution time [Doyle, 1969, Descotte and Latombe, 1985]. Hayes [Hayes, 1995b] enumerates some quality measures in process planning:

- the number of major steps (i.e. set-ups) as a rough measure of plan quality,
- the total time to complete the plan, including both machining time and set-up time,
- the feasibility of a plan upon execution (i.e. will the plan succeed?),
- the total dollar cost of the plan,
- the probability that the plan will produce the specified accuracy, and
- the reliability of the operations.

Hayes also points several difficulties in measuring plan quality: many factors are hard to measure without a huge body of empirical data; quality may be defined differently from shop to shop; and it may also be defined differently for different jobs. As Zhang and Lu [Zhang and Lu, 1992] rightly point out the ultimate objective is to develop the least-cost plan while maintaining satisfactory productivity. Their cost analysis includes a number of cost factors both variable (activity, tooling, set-ups, inventory) and fixed (depreciation and maintenance of the equipment).[2]

Set-ups are very important in process planning [Hayes, 1990]. A plan can be seen as a sequence of set-up steps, where each of those steps groups several operations to prepare the part, machine, and tools, and one or more machining operation. Set-ups are often the most time-expensive part of a plan. The number of set-ups roughly measures the cost of executing a plan, and it is commonly used by machinists as a heuristic to construct efficient plans. Sharing parts of the set-ups among operations on one or more parts usually reduces the total cost of the process plan.

Plan length is usually not an accurate metric of plan quality because different operators have different costs. For example, a tool can be switched automatically whereas resetting a part requires human assistance [Hayes, 1990]. Therefore plans that share part set-ups are cheaper than plans that share tools, even though those plans might have the same length. Figure 1.2 provides an example of such plans.

Planning is a constructive, incremental decision-making process. Each particular planner faces multiple decisions when generating a complete plan to solve a given problem. The

[2]For more detailed descriptions of the economics of process planning see for example [Zhang and Lu, 1992], and [Doyle, 1969], chapter 20.

types of those decisions vary from planner to planner, but they generally include choosing actions (operators) to achieve goals, choosing objects in the world with which to instantiate those operators, and choosing orderings on how the problem goals are achieved. All types of decisions that the planner makes at problem solving time may influence the quality of the final plan. In PRODIGY4.0 those decisions correspond to choosing a goal ordering, choosing an operator to achieve a goal, choosing values of all the parameters to instantiate the operator, and deciding when to apply an operator (instead of subgoaling on other pending operators) and which operator to apply. (See Appendix A for the details of PRODIGY4.0 planning algorithm and decision points.) Some examples of the influence of these decisions in plan quality follow:

- Goal ordering: suppose the problem posed to the planner is to have a part with two holes, one opening into another. This can be encoded as the conjunction of two goals, one for each hole. The planner has to start deciding on which hole to work first, i.e. which of the two goals try to achieve first. The following advice may be used to guide the planner's decision:

  > If a hole H_1 opens into another hole H_2, then one is recommended machining H_2 before H_1 in order to avoid the risk of damaging the drill. [Descotte and Latombe, 1985]

  This advice can be translated into a search control rule. The antecedent preconditions match when one of the holes actually opens into another. The consequent recommends with which hole to start planning. It is interesting to realize that the expert advice may apply only in some circumstances. If machining the holes in the opposite order is faster, the right decision could have been different, and the rule should only fire when reducing the risk of damaging the drill is more important than minimizing the time spent on the operations. Therefore different control rules, or control rule sets, may encode different strategies, that may require tradeoffs.

- Operator preferences: suppose the planner's goal is to reduce the size of a part. Some of the candidate operators to achieve this goal are *shape*, *shape-with-planer*, and *mill*. The following expert advice may be useful to decide which operator to try first, and can be translated into one or more control rules whose consequent proposes the appropriate operator:

  > In most shaping and planning operations, cutting is done in one direction only. The return stroke represents lost time. Thus these processes are slower than milling and broaching, which cut continuously. On the other hand, shaping and planning use single-point tools that are less expensive, are easier to sharpen, and are conducive to quicker set-ups than the multiple-point tools of milling and broaching. This makes shaping or planning often economical to machine one or a few pieces of a kind. ([Doyle, 1969], p. 597).

- Binding preferences: the choice of the correct machine, tools, and orientation for a given machining operator may save the cost of resetting the part:

  > It may be advantageous to execute several cuts on the same machine with the same fixing to reduce the time spent setting up the work on the machines [Descotte and Latombe, 1985].

- Applying an applicable operator versus subgoaling: PRODIGY4.0's default search strategy applies the applicable operators in strictly the opposite order in which they were chosen for expansion. In this domain it is useful to allow application of operators in a different order when all their preconditions become true. For example, suppose that there exist several goals to drill holes on one part, and some of the holes are in the same side and have the same diameter. If the planner has appropriately chosen the same machine and tool for those holes (at operator and binding decision time), once the preconditions of the first chosen hole are true, the other holes may be drilled even though the default strategy might suggest working on other goals.[3]

All the factors relevant to plan quality mentioned in this section are largely independent of the planning algorithm. Any planning architecture will need a control strategy that guides the planner geared toward plans of good quality according to some metric. This is the motivation of our interest on the process planning domain as a vehicle to explore the representation of plan quality metrics and the generation of good quality plans. The PRODIGY architecture is well suited as a vehicle for the investigation because it has clear explicit decision points (discussed above) that permit the infusion of automatically or manually acquired control knowledge to improve plan quality.

2.3 An Implementation of Process Planning

A domain in PRODIGY4.0 (and in most AI planning systems) is specified as a set of operators and inference rules. Operators represent the types of actions available for inclusion in the plan. Inference rules are used to compute the deductive closure of the set of predicates true in the current state.[4] There is also a domain ontology, that is, a hierarchy of the classes of objects in the domain. Figure B.1 of Appendix B shows the type hierarchy for the process planning domain. Appendix B also lists the process planning operators and inferences rules in our implementation. Building a complex domain has the difficulties typical of knowledge acquisition [Marcus, 1990,

[3] PRODIGY4.0's *permute-application-order* [Carbonell *et al.*, 1992] and SAVTA [Stone *et al.*, 1994] domain-independent search strategies achieve exactly this effect.

[4] Appendix A further explains the distinction and describes PRODIGY4.0's planning algorithm and knowledge representation.

Joseph, 1992, desJardins, 1994]. In our case an expert job-shop machinist assisted in the construction of the domain, and helped with the descriptions of the machine shop and real part specifications. Although our model is far from comprehensive and is limited in many ways, we have attempted to capture many of the complexities of the process planning task in order to develop an interesting test bench for our planning and learning research. The domain was acquired and implemented for the PRODIGY2.0 linear planner by Yolanda Gil and is described in more detail in [Gil, 1991].

PRODIGY generates plans to produce parts. It is given a request that specifies the material of the part, its shape (rectangular or cylindrical), its size along each dimension, its surface quality (roughness) and surface finish (metal coatings and polishing), and its features (holes that can be reamed, tapped, counterbored, etc). These specifications (or a subset of them) form the goal state. A description of a shop with machines, tools, and metal stock (i.e. proto-parts) forms the initial state of a problem. Rectangular parts in this implementation have six sides, and the location of a feature is determined with x and y coordinates on a given side. The planning operators represent the machining operations themselves, as well as steps to prepare the part and tool set-up, to secure the part with a holding device in a certain orientation, to clean the part and remove metal burrs from its surface, and to install an appropriate tool in the machine.

Consider, for example, an operator for face milling a part. Figure 2.1 shows the corresponding operator. We need to represent that (a) the size of the part will change along a dimension corresponding to the part side facing the machining direction, (b) the part must be held by a holding device in such a way that the desired dimension can be machined, and (c) a milling cutter needs to be in the machine's tool holder. The new size of the part must be smaller than the current size. Also, any surface properties of the side being machined will disappear, and the part will have dirt and burrs. These facts are represented as the effects and preconditions of the operator.

The domain implementation makes use of PRODIGY's ability to represent infinite types (types with infinitely many instances) and to do Lisp function calls for encapsulated calculations. Infinite types are used to represent numeric quantities, such as part sizes, hole depths, diameters, and angles. Functions can be used to generate values for variables with infinite types (for example, by inferring them from the values of other variables) and to perform numeric calculations. In the *face-mill* operator, the function `smaller` is used to constraint the legal values of part sizes allowed by the operator, and it represents the constraint that the part size never increases after milling.

Inference rules are used to specify the availability of machines, parts, tools, tool holders, and holding devices. Figure 3.5 shows one of such rules. Inference rules are also used to determine which sides should be used to hold a part to machine it in a given dimension.

In the examples discussed throughout the thesis we use parts with rectangular shape. Cylindrical parts are also allowed. Figure 2.2 (a) clarifies the relationship among the part's dimensions

```
(Operator FACE-MILL
 (params <machine> <part> <cutter> <hold-dev>
         <side> <side-pair> <dim> <old-size> <new-size>)
 (preconds
   ((<machine> MILLING-MACHINE)
    (<cutter> MILLING-CUTTER)
    (<hold-dev> (or 4-JAW-CHUCK VISE COLLET-CHUCK TOE-CLAMP))
    (<part> PART)
    (<dim> DIMENSION)
    (<side> SIDE)                ;side facing up
    (<side-pair> SIDE-PAIR)      ;pair of sides facing the holding device
    (<old-size> (and SIZE (gen-from-pred (size-of <part> <dim> <old-size>))))
    (<new-size> (and SIZE (smaller <new-size> <old-size>))))
  (and (shape-of <part> rectangular)
       (side-up-for-machining <dim> <side>)
       (sides-for-holding-device <side> <side-pair>)
       (holding-tool <machine> <cutter>)
       (holding <machine> <hold-dev> <part> <side> <side-pair>)))
 (effects ((<surface-coating> SURFACE-COATING)
           (<surface-finish> SURFACE-FINISH))
  ((del (is-clean <part>))
   (add (has-burrs <part>))
   (del (surface-coating-side <part> <side> <surface-coating>))
   (del (surface-finish-side <part> <side> <surface-finish>))
   (add (surface-finish-side <part> <side> rough-mill))
   (add (size-of <part> <dim> <new-size>))
   (del (size-of <part> <dim> <old-size>)))))
```

Figure 2.1: The *face-mill* operator. Operators in PRODIGY4.0 are defined by their preconditions and effects. Preconditions and effects contain variables (part, tool, machine, etc.) whose types are specified in the operator and appear in capital letters in the figure. The values of variables may be constrained by functions, as in the case of `<new-size>`, the new size of the part, which has to be `smaller` than the old size. The preconditions of an operator must be true in the state for the operator to be applicable. They are represented as a logical expression that may contain conjunctions, disjunctions, negations, and quantification. The preconditions of *face-mill* require that the part and the tool are appropriately set on the milling machine. The effects of the operator are the predicates to be added (e.g. the part will have a new size and will have burrs after milling) and deleted (e.g. the part will not be clean) from the state when the operator applies. Milling the part removes its surface properties, no matter what they are. Therefore `<surface-finish>` and `<surface-coating>` act as universally quantified variables. Some operators in this domain have also conditional effects that are to be performed depending on particular state conditions.

(width, length, and height) and the numbering of the part's sides for rectangular parts. The sides of a part are uniquely identified, and the orientation of a part held by a holding device on a machine table is well defined by the side facing up and the pair of opposite sides that are touching the holding device. In the *face-mill* operator (Figure 2.1) they correspond to the instantiations of variables `<side>` and `<side-pair>`. Most operations machine the side facing up (side-milling is an exception). When the part is being held, the sides facing the holding device are not available for machining. When the planner chooses an operator to achieve a goal,

it must instantiate it, that is, bind the operator variables to objects in the problem. Choosing the bindings for a machining operator includes determining the part orientation so the desired side can be machined. There are usually several orientations that allow the machining operation. The choice of orientation, holding device, and tool is important as it may influence the number of set-ups in the plan: if the proper values are chosen, several machining operations may be performed with the same set-up. Figure 2.2 (b) shows the set-up of a part and tool on a drill press.

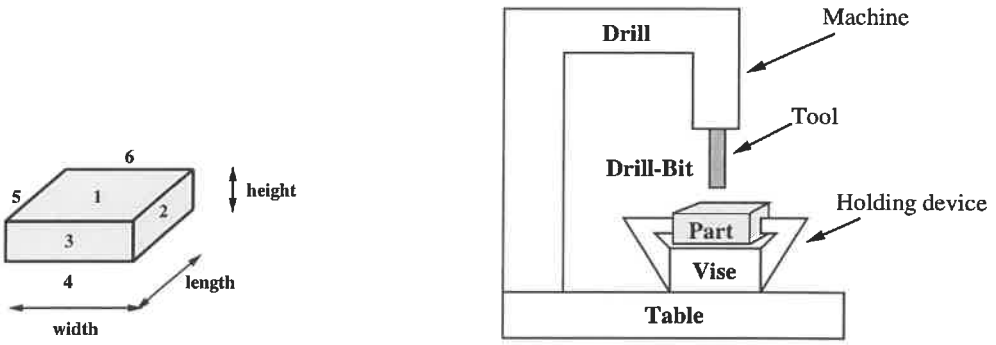


Figure 2.2: (a) Dimensions and sides of a part. The numbers indicate how the part sides are identified and are used in the examples throughout the thesis. (b) An example of a part and tool set-up (from [Joseph, 1992]). The part is being held by the holding device, which is a vise in the example of the figure. The part and holding device are sitting on the table of a machine. The machine is a drill-press in this example. The machine tool-holder is holding a tool to drill a hole (a drill-bit).

Some qualitative and quantitative measures of the complexity of this domain follow:

- The precondition expression of some operators and inference rules includes negated predicates, disjunctions, and universal quantification. Some of the preconditions correspond to predicates derived by inference rules.
- The effects of many operators are not reversible. In addition, some effects are context-dependent and are encoded in PRODIGY as conditional effects.
- There are 117 rules, that include 73 operators and 44 inference rules. Of the 73 operators, 38 correspond to machining operations, and 35 to set-ups.
- The average number of parameters for an operator is 7, the average number of preconditions is 5, and the average number of effects (adds and deletes) is 3, although the most complex operators may contain considerably more.

- There are 41 different predicates, of which 7 are static (i.e., they do not change during problem solving). Eleven Lisp domain-specific functions are defined as part of the domain and used in the operators to perform numerical computations and constrain variable values.

- There are 85 different types and subtypes of objects in the type hierarchy, 5 of which are infinite types.

- In order to increase the search efficiency 18 control rules are used (see below). This number does not include the quality-enhancing rules learned automatically by the algorithms described in this thesis.

- The length of many solutions is over one hundred steps (including operators and inference rules).

- The initial state that represents the machine shop may include more than 500 facts.

Some hand-coded control rules guide the search in order to avoid failure search paths and thus improve planning efficiency. For example, when the current goal is to set a part free on the machine table, and the part is being held by that same machine, the planner should not try to subgoal on bringing the part over from a different machine table, but rather to release the part from the holding device. That is the purpose of the operator rejection rule in Figure 2.3. Appendix B lists the set of hand-coded control rules for planning efficiency. Control rules that make the planner more efficient could be learned by one of PRODIGY's learning mechanisms [Minton, 1988, Etzioni, 1990, Pérez and Etzioni, 1992, Borrajo and Veloso, 1994a] and thus are not the focus of our research.

```
(control-rule PUT-ON-MACHINE-TABLE-IF-HOLDING
 (if (and (current-goal (on-table <machine> <part>))
          (or-metapred
           (known (holding <machine> <holding-device> <part> <s> <s1> <s2>))
           (known (holding-weakly <machine> <holding-device> <part> <s> <s1> <s2>)))))
 (then reject operator PUT-ON-MACHINE-TABLE))
```

Figure 2.3: Control rule that rejects moving the part from another machine if the part is being held by the desired machine <machine> already.

This domain, and also other domains used along the thesis, include a control rule that expands certain goals first before starting work on their subgoals. Figure 2.4 shows this rule. The rule indicates that machining goals should be expanded before working on goals that setup the work for the machining operations. Having the set-up subgoals available is useful to make informed choices of operator and bindings for the remaining machining goals.

```
(control-rule EXPAND-MAIN-GOALS-FIRST
  (if (and
       (candidate-goal <goal>)
       (goal-instance-of <goal>
        has-hole has-spot is-tapped is-countersinked is-counterbored is-reamed
        size-of shape-of surface-finish-quality-side surface-coating-side surface-coating)
       (candidate-goal <other-goal>)
       (~ (goal-instance-of <other-goal>
        has-hole has-spot is-tapped is-countersinked is-counterbored is-reamed
        size-of shape-of surface-finish-quality-side surface-coating-side surface-coating))))
  (then reject goal <other-goal>))
```

Figure 2.4: Control rule that expands the machining goals first.

2.4 An Example

We now introduce a simple example problem in the process planning domain. This problem is used in later sections to illustrate the learning algorithms. Figure 2.5 shows its initial state and goal statement. The goal is to have a part of height 2 with a spot hole in its side 1 at coordinates 1.375×0.25. The rest of the part features, such as its material, width, or length remain unspecified. An aluminum part `part5` of height 3, length 5, and width 3 is available in the shop, i.e. in the initial state. There are also some tools, including a spot drill and a plain mill (a type of milling cutter), and two machines, namely a milling machine and a drill press. Milling machines can be used both to reduce the size of a part, and to drill holes and spot holes on it. Drill presses can drill holes and spot holes, and terminate the holes in different ways (tapped, counterbored, etc). The planner uses inference rules to infer from the state that the machines, tools, and part are initially free, and that the part available has rectangular shape. These rules do not change the world state but compute the deductive closure of the facts already in the state.

Table 2.1 shows the plan quality metric used in the example. It represents the quality of a plan as its execution cost, computed by adding the cost of the individual operators in the plan. Higher values of the metric represent worse quality. Each operator has a cost, which in this example is independent of how the operator variables are instantiated.[5] PRODIGY4.0 uses inference rules for planning as if they were operators, i.e. for achieving subgoals, but they do not correspond to actual process planning operators. As we mentioned before, their application only computes the deductive closure of the state. They are part of the plan but the quality metric assigns them cost 0 and for clarity purposes we will not display them as part of the plans in our examples.

The plan quality metric of this example focuses on the cost of setting up the work on the job-shop machines, and was motivated by the fact that machinists frequently use the number of major steps, i.e. set-ups, as a rough measure of the quality of the plan [Hayes, 1995b], as

[5]Section 4.1.2 describes a more complex quality metric that assigns values to the operators depending on the instantiations of the operator variables.

```
(objects
;;machines
 (object-is milling-machine1 MILLING-MACHINE)
 (object-is drill1 DRILL)

;;holding devices
 (object-is vise1 VISE)

;;parts and holes
 (object-is part5 PART)
 (object-is hole1 HOLE)

;;tools
 (object-is spot-drill1 SPOT-DRILL)
 (object-is twist-drill6 TWIST-DRILL)
 (object-is plain-mill1 PLAIN-MILL)
 (object-is end-mill3 END-MILL)
 (object-is brush7 BRUSH)
 (object-is soluble-oil SOLUBLE-OIL)
 (object-is mineral-oil MINERAL-OIL))

        (state (and (diameter-of-drill-bit twist-drill6 9/64)
                    (material-of part5 ALUMINUM)
                    (size-of part5 LENGTH 5)
                    (size-of part5 HEIGHT 3)
                    (size-of part5 WIDTH 3)))

(goal ((<part> PART)) (and (size-of <part> HEIGHT 2)
                           (has-spot <part> hole1 side1 1.375 0.25)))
```

Figure 2.5: A problem specification in the process planning domain. A problem is specified by the initial state and the goal statement.

Type	Cost	Operators
Machining operators	2	drill-with-spot-drill, drill-with-twist-drill, drill-with-high-helix-drill, tap, countersink, counterbore, ream, drill-with-spot-drill-in-milling-machine, drill-with-twist-drill-in-milling-machine, face-mill, side-mill
Machine and holding device set-up operators	2	put-holding-device-in-drill, put-holding-device-in-milling-machine, remove-holding-device-from-machine, put-on-machine-table, remove-from-machine-table, hold-with-vise, release-from-holding-device
Tool operators	1	put-tool-on-milling-machine, put-tool-in-drill-spindle, remove-tool-from-machine
Cleaning operators	2	clean, remove-burrs
Oil operators	1	add-soluble-oil, add-mineral-oil, add-any-cutting-fluid

Table 2.1: A quality metric for the quality of the plans in the process planning domain. The operators have been separated in groups for clarity purposes. Next to each group is the cost of each of the operators in the group.

discussed in Section 2.2. This metric assigns lower costs to the operators for moving tools than to the operators for preparing and moving parts. The reason is that in computer numeric

controlled (CNC) machining centers tools are switched automatically while holding the parts requires human assistance and therefore is more expensive [Hayes, 1990].

A plan	A better plan
put-tool-drill drill1 spot-drill1	**put-tool-mm** milling-mach1 spot-drill1
put-holding-device-drill drill1 vise1	**put-holding-dev-milling-mach** milling-mach1 vise1
clean part5	**clean** part5
put-on-machine-table drill1 part5	**put-on-machine-table** milling-mach1 part5
hold drill1 vise1 part5 side1 side2-side5	**hold** milling-mach1 vise1 part5 side1 side3-side6
drill-in-drill-press drill1 spot-drill1 vise1 part5 hole1 side1 side2-side5	**drill-in-milling-mach** milling-mach1 spot-drill1 vise1 part5 hole1 side1 side3-side6
	remove-tool milling-mach1 spot-drill1
put-tool-milling-mach milling-mach1 plain-mill1	**put-tool-milling-mach** milling-mach1 plain-mill1
release drill1 vise1 part5	
remove-holding-device drill1 vise1	
put-holding-dev-milling-mach milling-mach1 vise1	
remove-burrs part5 brush7	
clean part5	
put-on-machine-table milling-mach1 part5	
hold milling-mach1 vise1 part5 side1 side3-side6	
face-mill milling-mach1 part5 plain-mill1 vise1 side1 side3-side6 height 2	**face-mill** milling-mach1 part5 plain-mill1 vise1 side1 side3-side6 height 2
cost = 28	**cost = 15**
(a)	(b)

Figure 2.6: (a) Plan obtained by PRODIGY for the problem in Figure 2.5. (b) A better plan, according to the quality metric, for the same problem.

Figure 2.6 shows two solutions for the problem described above. The crucial difference between the two plans is the type of drilling operation. Plan (a) uses the drill press to drill the spot hole. Step `<drill-with-spot-drill drill1 spot-drill1 vise1 part5 hole1 side1 side2-side5>` indicates that `hole1` will be drilled in `part5` on the drill press `drill1` using as a tool a drill-bit `spot-drill1`, while the part is being held with its side 1 up and sides 2 and 5 facing the holding device, `vise1` (cf. Figure 2.2). Plan (b) uses the milling machine to drill the spot hole, and the drill and mill operations share the same set-up (machine, holding device, and orientation), so that the part does not have to be released and held again. The ability to share the set-up is due to the choice of operator to drill the hole (*drill-in-milling-machine*) and the choice of the same instantiations for the machine, holding device, and orientation in both the drill and mill operations. Because of this set-up sharing, plan (b) is a better plan according to the metric described.

2.5 Summary

This chapter has described the process planning domain that will be used as an example throughout the thesis to exemplify the quality-enhancing control-knowledge learning algorithms and evaluate them empirically. The task of process planning is to generate plans to manufacture parts given the specifications of the part and the manufacturing environment. In order to generate production-quality plans, issues about plan quality must be considered. The chapter has given examples of how those issues affect all the types of planning decisions. In particular reducing set-up costs is commonly used by human machinists as a rough heuristic to construct efficient plans. All these characteristics make of process planning and interesting domain for AI planners.

Process planning has been implemented as a complex domain for the PRODIGY4.0 nonlinear planner. The implementation includes a large number of operators, which describe machining and set-up actions, inference rules, and classes of objects organized in the domain ontology. This chapter has described the implementation and given an example of a planning problem and a quality metric in this domain

Chapter 3

Search Control Rule Learning

The goal of learning plan quality is to acquire heuristics that guide the planner at generation time to produce plans of good quality automatically. In this chapter we present one strategy for addressing this learning problem, and a more complex strategy is presented in Chapter 4. Section 3.1 presents a high level view of the learning system. Section 3.2 describes the Interactive Plan Checker that (optionally) allows a human domain expert to interact with the learner at the operator level to suggest variations of the plan produced that improve its quality. [1] Sections 3.3 to 3.10 describe in detail the learning process, namely finding learning opportunities, building operator and bindings quality-enhancing control rules, the need for goal preference rules, and how they are learned. Section 3.11 explains how the learned rules may be overgeneral and lead to incorrect decisions, and QUALITY's way of incrementally refining the learned knowledge by adding new rules and priorities among them. The limitations of this algorithm and of the use of control rules to represent quality-enhancing control knowledge are discussed in Section 3.12. These limitations motivate the more complex method discussed in Chapter 4. Section 3.13 presents the results of a series of empirical evaluations that demonstrate the effectiveness of the automatically-acquired plan-quality control rules.

The learning methods described in this chapter are independent of the application domain. Throughout this chapter we will illustrate them with examples from the process planning domain described in Chapter 2 and from some artificial domains. Section 3.12 discusses some of the characteristics of the domains and of quality metrics for which our algorithms for learning quality-enhancing control rules have a good performance.

[1] Although the Interactive Plan Checker and the overall architecture are described in this chapter devoted to control rule learning, they are also used by the control knowledge trees learning algorithm described in Chapter 4.

3.1 The Architecture

Knowledge about plan quality in a domain D comes in two forms: (a) a post-facto quality metric $Q_D(P)$ that computes the quality (e.g. the execution cost) of plan P, and (b) planning-time decision-control knowledge used to guide the planner towards producing high-quality plans. The first kind of knowledge, cost functions $Q_D(P)$, is non-operational for plan generation; it cannot be brought into play until *after* a plan is produced. Yet, cost functions is exactly the kind of quality knowledge typically available, in contrast to the far more complex operational decision-time knowledge. Hence, automatically acquiring the second kind of knowledge from the first is a very useful, if quite difficult endeavor. In essence, learning operational (planning-time) quality control knowledge can be seen as a *translation* problem of domain knowledge D and quality metric Q_D into runtime decision control guidance:

$$\textit{Quality: } D \times Q_D \rightarrow \textit{Decision-Control}_{D,Q_D}$$

And the *full automation of this quality mapping problem* is the ultimate objective of this thesis. In practice, the mapping is learned incrementally with experience.

Figure 3.1 shows the architecture of QUALITY, the core of this thesis, which learns precisely that mapping. QUALITY is *given* a domain theory D (operators and inference rules) and a domain-dependent metric that evaluates the quality of the plans produced $Q_D(P)$. It is also given problems to solve in that domain. QUALITY analyzes the planning episodes by comparing the search trace[2] for the plan obtained given the current control knowledge, and another search trace corresponding to a better plan (*better* according to the quality metric). The latter search trace is obtained by asking a human expert for a better plan and then producing a search trace that leads to that plan. Alternatively, the system can function autonomously by letting the planner search further until a better plan is found[3]. The system then analyzes the differences between the sequence of decisions that the planner made initially and the ones that should have been made to generate the plan of better quality. The learner interprets these differences as learning opportunities and identifies the conditions under which the individual planning choices will lead to the desired final plan. In this way QUALITY compiles knowledge to control the decision making process in new similar planning situations to generate plans of better quality.

Several points are worth mentioning:

- Learning is driven by the existence of a better plan and a failure of the current control knowledge to produce it. The learner can actively find the trace corresponding to a better plan either by letting the planner search further, or by asking a human expert for a better plan and then producing a search trace that leads to that plan.

[2] By *search trace* we mean the sequence of decisions made by the planner when solving the problem.

[3] Because of the large search spaces in complex domains, this can be computationally very expensive in practice (cf Section 3.12).

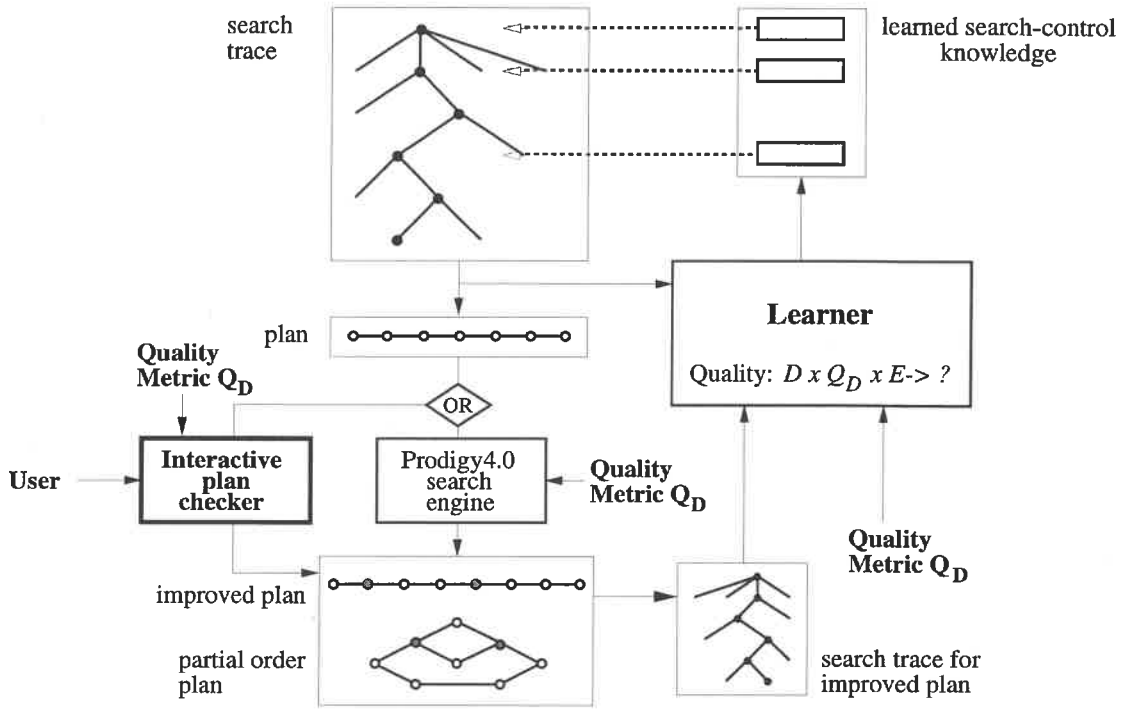


Figure 3.1: Architecture of QUALITY, that learns control knowledge to improve the quality of plans. QUALITY automates the quality mapping problem of translating the non-operational quality metric Q_D into operational quality control knowledge that can be used at planning time.

- The glass-box approach followed in PRODIGY's design, i.e. the ability to introspect fully into its decision cycle and thus modify it at will, allows the learner to notice gaps in the control knowledge and reason about how to fill these gaps [Pérez, 1994]. These gaps correspond to learning opportunities and arise by finding wrong problem solving decisions, where search deviated towards the worse plan. These suboptimal decisions are due either to a lack of control knowledge or to wrong or conflicting preferences suggested by the current control rules. Only the opportunities that appear earlier in the search trace (the sequence of the decisions made by the planner) are explored by the learning mechanism. What is learned depends on the learning opportunity: if no control knowledge was available that would suggest the desired alternative, a new control rule is learned; if conflicting preferences were suggested by several control rules, a priority between the rules is learned. If the present plan cannot be improved, nothing is learned.

- The quality mapping problem $Quality(D, Q_D)$ addressed by QUALITY entails a change of representation from the knowledge about quality encoded in the domain-dependent plan quality metric Q_D into knowledge operational at planning time. As QUALITY learns incrementally with experience, the mapping is more precisely defined as:

$$\textit{Quality: } D \times Q_D \times E \rightarrow \textit{Decision-Control}_{D,Q_D}$$

where E is the actual planning experience including the human expert improvements that bias the learning.

- Operational knowledge is needed because the plan and search tree are only partially available when a decision has to be made. The translated knowledge is expressed as search-control knowledge in terms of the problem solving state and meta-state, such as which operators and bindings have been chosen to achieve the goals, or which are the candidate goals to expand.

- We do not claim that the learned control knowledge will necessarily guide the planner to find optimal plans, but that the quality of the plans will incrementally improve with experience, as the planner sees new interesting problems in the domain.[4]

The sections that follow describe in detail the different parts of the architecture.

3.2 The Interactive Plan Checker

QUALITY can interact with a human expert so that the expert suggests variations to a given plan that may produce a plan of better quality. This interaction is performed by a module, the *interactive plan checker*, whose purpose is to capture the expert knowledge about how to generate better plans in order to learn quality-enhancing control knowledge. The interactive plan checker obtains a better plan from the domain expert and tests whether that plan is correct and actually of better quality. The interaction with the expert is at the level of plan steps, i.e. concrete actions to perform, which correspond to instantiated operators, and not at the level of the full range of planning time decisions. This relieves the expert of understanding PRODIGY4.0's search procedure, a very important feature to interact productively with domain experts who are not knowledge engineers. Section 3.2.1 describes the interactive plan checker. Section 3.2.2 presents an example and additional details.

3.2.1 Description

The interactive plan checker offers to the expert the initial plan obtained by the planner with the current knowledge as a guide to build a better plan. The expert can add, remove, or modify the steps of the initial plan. A plan step is an instantiated operator, and it is represented with the operator name and the values (*bindings*) with which the operator variables are instantiated. The

[4]However the methodology used to evaluate QUALITY's performance in the experimental results sections separates the training phase (in which learning occurrs) from the test phase (in which learning is inhibited).

initial plan is presented in menu form and the expert suggests in sequence the steps that make up a better plan. At each point the expert may choose any step from the initial plan, or provide a new step. The step is checked for applicability in the current state. If the preconditions of the operator are true in the state, the step execution is simulated and its effects are added to or deleted from the state, effectively modifying it. Note that the plan is built forward, and the order in which the steps are provided is important, as the state is updated with each step and the effects of one step may preclude the applicability of the preconditions of the next. At any time the expert may ask to see the current state. When the expert finishes inputting the plan steps, the checker verifies that the plan provided satisfies the problem goal. Note that by construction both the initial plan and the expert plan are correct [5] but they differ in their quality. Finally the quality metric is applied to the plan provided and its value shown to the expert, who may then decide to abandon the plan and restart the process to provide another plan.

The interaction of the expert with the interactive plan checker, and thus with QUALITY, is only at the level of plan steps, that is, of instantiated operators. Operators represent actions in the world. On the other hand, inference rules usually do not correspond semantically to actions in the world and are used only to compute the deductive closure of the current state [Carbonell *et al.*, 1992]. The plan checker does not require that the user specifies inference rules as part of the plan (neither are they shown as part of the initial plan), but it fires the relevant ones as needed. The features of the interactive plan checker that we have described allow the domain expert to remain oblivious to the planning algorithm, in particular to PRODIGY4.0's decision points, and also to the planner's control-knowledge representation language. The expert only needs to know which are the available operators, their parameters (variables), and their preconditions and effects. This is a reasonable assumption if the expert has provided the knowledge to build the domain. An additional assumption is that the expert's model of plan quality is consistent with the quality metric.

Figure 3.2 describes the interactive plan checker in detail. In Step 5 the expert may pick an operator from the old plan, which is presented in menu form, or propose a totally new operator. This process is shown in Figure 3.3. At each point s/he can ask to see the current state (Steps 4-5). If a new operator is proposed (Steps 7-14 of Figure 3.3) the plan checker verifies that the input operator name is valid and that the bindings satisfy the type specifications of the variables indicated in the operator schema. If some bindings are not specified, or they have an illegal values, the system tries to compute them from the variable's type specification. For example, the type specification of operator *drill-with-twist-drill* indicates that the values of the variables that represent the diameter of the drill-bit and the diameter of the hole produced can be inferred given the choice of tool. If more than one value is possible, the checker asks the user, possibly suggesting a default value.

Then the operator precondition expression is tested in the current state to determine if the

[5] A plan is *valid* if the preconditions of every operator are satisfied before the operator execution. A plan is *correct* if it is valid and it satisfies the goal statement [Veloso *et al.*, 1995].

Interactive_plan_checker
Input: *initial_plan*. *Goal, initial_state,* and *current_state* are global variables.
Output: a sequence *new_plan_all_steps* of instantiated operators and inference rules that constitute a correct plan.

```
1.  show_operators(initial_plan)
2.  current_state ← initial_state
3.  new_plan ← ∅
4.  new_plan_all_steps ← ∅
5.  op ← input_operator                                   ;; See Figure 3.3
6.  if op = 0
    then                                                  ;; user has completed plan
7.       if satisfied_goal_in_state?
8.         ∧ eval_quality(new_plan) > eval_quality(initial_plan)
9.         ∧ input_user_satisfied?(new_plan)              ;; User wants further improvements
10.      then return(new_plan_all_steps)
11.      else goto 2
    else
12.     failed_prec ← test_applicable(op)                 ;; See Figure 3.4
13.     if failed_prec
14.     then inform_user(op, failed_prec)                 ;;op can't be executed
15.          goto 5
16.     else current_state ← apply_op(op)                 ;;execute op
17.          current_state ← fire_eager_inference_rules
18.          new_plan ← append(new_plan, op)
19.          new_plan_all_steps ← append(new_plan_all_steps, op)
20.          goto 5
```

Figure 3.2: Interaction with the expert and checking of the plan obtained. **Eval_quality** is the plan quality metric. **Apply_op** updates the current state with the effects of the instantiated operator. In this and the next figures bold face indicates a procedure call and italics are used to represent variables.

operator is applicable. Figure 3.4 shows this process. If the precondition is not satisfied, i.e. the operator is not applicable, the checker notifies the expert which precondition conjunct failed to be true (Steps 12-15 of Figure 3.2). If the operator precondition expression is satisfied, the operator is executed and the state updated (Steps 16-20). When the new plan is completed, if the goal is not satisfied in the final state (Step 7), or the plan has worse quality than the previous plan according to the plan quality metric (Step 8), the plan checker informs so and

input_operator
Input: *initial_plan*
Output: an instantiated operator (or 0 to indicate end of plan)

```
1.  user_input ← input_next_step
2.  case user_input
3.    0:      return(0)                                   ;; User indicates end of plan
4.    !:      show(current_state)                         ;; User asks to see current state
5.            goto 2
6.    a_valid_op_number: return(nth(user_input, initial_plan)  ;; Use operator from initial plan
7.    *:      op_and_args ← input_op_and_bindings          ;; Use a new operator
8.            if invalid_op_name(op_and_args)
9.            then goto 7
10.           invalid_bnds ← check_op_type_specification(op_and_args)
11.           if invalid_bnds
12.           then op_and_args ← ask_for_individual_bindings_offer_defaults(invalid_bnds)
13.                goto 10
14.           else return(op_and_args)
```

Figure 3.3: Obtaining the next operator from the expert.

prompts the expert for another plan, as the focus is on learning control knowledge that actually improves plan quality. The plan checker also allows the expert to further improve the quality of the current plan (Step 9) as in some problems it is easier for him/her to incrementally suggest improvements that may have been overlooked in the first revision.

As we mentioned, the interaction with the user is at the level of operators, which represent actions in the world. In domains with inference rules, they usually do not correspond semantically to actions in the world and are used only to compute the deductive closure of the current state [Carbonell *et al.*, 1992]. (Appendix A describes PRODIGY4.0's inference rules and Figure 3.5 gives an example.) Therefore the plan checker does not require that the user specifies them as part of the plan, but it fires the rules needed in a similar way to the planner. Each time the state changes after executing one step, the *eager* inference rules are fired (Step 17 of Figure 3.2). The *lazy* inference rules fire only on demand, when a precondition of an operator is not true in the state, and they may fire on a chain (Steps 2-7 of Figure 3.4). A truth maintenance system keeps track of the rules that are fired, and when an operator is applied and the state changes, the effects of the inference rules whose preconditions are no longer true are undone.

This process terminates when the expert is satisfied with the quality of the current plan. Actually the expert may terminate the dialog even if full satisfaction is not achieved (e.g. the expert runs out of time or patience...). Still, partial quality improvement may be obtained by learning using the current plan, as learning is incremental and robust, rather than all-or-nothing.

test_applicable(*op*)
Input: *op*; *current_state* is a global variable
Output: the first unsatisfied literal in the precondition expression

```
1. unsatisfied_literal ← process_and_test(precond_exp(op))
2. if unsatisfied_literal
   then
3.      lazy_rule ← find_relevant_applicable_lazy_inference_rule(unsatisfied_literal)
4.      if lazy_rule                          ;; lazy_rule adds unsatisfied_literal
        then
5.          current_state ← apply_op(lazy_rule)
6.          new_plan_all_steps ← append(new_plan_all_steps, lazy_rule)   ;; See Figure 3.2
        else
7.          return(unsatisfied_literal)
8. else return(nil)                           ;; op is applicable
```

Figure 3.4: Testing if the expert-input operator is applicable in the current state. **Process_and_test** processes the operator precondition expression handling the rich PRODIGY4.0 syntax (conjunction, disjunction, quantification) and tests its truth in the current state. **Find_relevant_applicable_lazy_inference_rule** determines whether the unsatisfied precondition can be added by a lazy inference rule. Therefore lazy inference rules fire only on demand.

```
(Inference-Rule HOLDING-DEVICE-EMPTY
 (mode lazy)
 (params <machine> <holding-device>)
 (preconds
  ((<machine> Machine) (<holding-device> Holding-device))
  (forall
   ((<part> Part)
    (<side> Side)
    (<side-pair> (and Side-Pair (gen-from-pred (sides-for-holding-device <side> <side-pair>)))))
   (and (~ (holding-weakly <machine> <holding-device> <part> <side> <side-pair>))
        (~ (holding <machine> <holding-device> <part> <side> <side-pair>)))))
 (effects ()
  ((add (is-empty-holding-device <holding-device> <machine>)))))
```

Figure 3.5: Lazy inference rule in the process planning domain. If the holding device is not holding any part, then it is empty.

In the end the interactive plan checker returns a list of steps (instantiated operators and inference rules) that form a *correct* plan [Veloso *et al.*, 1995], i.e. one that executed sequentially starting on the initial state will end in a state that satisfies the goal statement.

3.2.2 Examples and Further Details

Figure 3.6 presents an example of the dialog between the interactive plan checker and the human expert for the problem in Section 2.4.[6] The expert was shown the initial plan that PRODIGY4.0 found guided by the current control knowledge. That plan is in Figure 2.6(a). The expert detected ways in which this plan could be improved and suggested them in the dialog. The underlined text corresponds to the expert's input. The expert starts by proposing an operator that is not in the initial plan. S/he only inputs the operator name and is offered default binding values for the machine and the tool. Other plan steps are input by indicating their number in the initial plan. The resulting plan corresponds to plan (b) in Figure 2.6.

Figure 3.7 presents a second example of dialog for the same problem using the checker's verbose mode. The example illustrates an operator that was suggested but was not applicable in the current state. The expert suggested starting by drilling the spot-hole. However the tool had not been set yet on the machine spindle and the operator could not be applied because one of its preconditions was false. Therefore the expert inputs a step to situate the tool in place. Note that although the precondition `(is-available-tool-holder milling-machine1)` was false in the state, it could be added deductively by an inference rule. The system found the rule, `tool-holder-available`, and fired it. The verbose mode illustrates as well the effects of applying the operator in the current state. Adding `(holding-tool milling-machine1 spot-drill1)` to the state forced the TMS to retract from the state facts like `(is-available-tool-holder milling-machine1)` that were added by inference rules supported by the operator effect. When the operator was eventually applied, all the eager inference rules were fired, mimicking PRODIGY4.0's behavior at operator application time.

3.3 The Control-Rule Learning Algorithm: A Top-Level View

Figure 3.8 shows QUALITY's basic procedure to learn quality-enhancing control knowledge, in the case that a human expert provides a better plan. The procedure assumes that the expert and the system share the quality metric Q_D. Steps 2, 3 and 4 correspond to the interactive plan checking module, described in Section 3.2, that asks the expert for a better solution S_e and checks for its correctness. Step 6 constructs a problem solving trace from the expert solution and obtains decision points where control knowledge is needed, which in turn become learning opportunities. Step 8 corresponds to the actual learning phase. It compares the plan trees obtained from the problem solving traces in Step 7, explains why one solution was better than the other, and builds new control knowledge. The sections that follow describe these steps in

[6]Developing the user interface was not a goal of this thesis. Thus the interface shown in the figure leaves much room for improvement. A more user-friendly interface, possibly integrated with PRODIGY4.0's graphical user interface, is under development.

```
This is the initial solution obtained by the planner:
  1. <put-in-drill-spindle drill1 spot-drill1>
  2. <put-holding-device-in-drill1 drill1 vise1>
  3. <clean part5>
  4. <put-on-machine-table drill1 part5>
  5. <hold-with-vise drill1 vise1 part5 side1 side2-side5>
  6. <drill-with-spot-drill1 drill1 spot-drill1 vise1 part5 hole1 side1 side2-side5 1.375 0.25>
  7. <put-tool-on-milling-machine milling-machine1 plain-mill1>
  8. <release-from-holding-device drill1 vise1 part5 side1 side2-side5>
  9. <remove-holding-device-from-machine drill1 vise1>
  10. <put-holding-device-in-milling-machine milling-machine1 vise1>
  11. <remove-burrs part5 brush7>
  12. <clean part5>
  13. <put-on-machine-table milling-machine1 part5>
  14. <hold-with-vise milling-machine1 vise1 part5 side1 side3-side6>
  15. <face-mill milling-machine1 part5 plain-mill1 vise1 side1 side3-side6 height 3 2>
compute-cost = 28

[setting the initial state]
Enter each operator as an operator number (if it was in the previous plan)
for a new operator, or !  to see the current state.  Terminate with 0.
Op number> *
Input instantiated operator (no parens):  put-tool-on-milling-machine

Variable <MACHINE> has no value.  Its type specification is MILLING-MACHINE.
Input a value [MILLING-MACHINE1]:

Variable <ATTACHMENT> has no value.  Its type specification is (OR MILLING-CUTTER DRILL-BIT).
Input a value [END-MILL1]:  spot-drill1
.  #<PUT-TOOL-ON-MILLING-MACHINE [<MACHINE> MILLING-MACHINE1] [<ATTACHMENT> SPOT-DRILL1]>

Op number> 10
10.  #<PUT-HOLDING-DEVICE-IN-MILLING-MACHINE [<MACHINE> MILLING-MACHINE1] [<HOLDING-DEVICE> VISE1]>

Op number> 12
12.  #<CLEAN [<PART> PART5]>

Op number> 13
13.  #<PUT-ON-MACHINE-TABLE [<MACHINE> MILLING-MACHINE1] [<PART> PART5] [<ANOTHER-MACHINE> ()]>

Op number> 14
14.  #<HOLD-WITH-VISE [<HOLDING-DEVICE> VISE1] [<MACHINE> MILLING-MACHINE1] [<PART> PART5] [<SIDE> SIDE1] [<SIDE-PAIR> ...

Op number> *
Input instantiated operator (no parens):  drill-with-spot-drill-in-milling-machine milling-machine1 spot-drill1 vise1 part5
hole1 side1 side3-side6

Variable <LOC-X> has no value.  Its type specification is (AND HOLE-LOCATION (X-LOCATION-OF #<P-O: PART5 part> <LOC-X>)).
Input a value [NIL]: 1.375
Variable <LOC-Y> has no value.  Its type specification is (AND HOLE-LOCATION (Y-LOCATION-OF #<P-O: PART5 part> <LOC-Y>)).
Input a value [NIL]: 0.25
.  #<DRILL-WITH-SPOT-DRILL-IN-MILLING-MACHINE [<MACHINE> MILLING-MACHINE1] [<DRILL-BIT> SPOT-DRILL1] [<HOLDING-DEVICE> ...
Op number> *
Input instantiated operator (no parens):  remove-tool-from-machine milling-machine1 spot-drill1
.  #<REMOVE-TOOL-FROM-MACHINE [<MACHINE> MILLING-MACHINE1] [<TOOL> SPOT-DRILL1]>

Op number> 7
7.  #<PUT-TOOL-ON-MILLING-MACHINE [<MACHINE> MILLING-MACHINE1] [<ATTACHMENT> PLAIN-MILL1]>

Op number> 15
15.  #<FACE-MILL [<MACHINE> MILLING-MACHINE1] [<MILLING-CUTTER>
PLAIN-MILL1] [<HOLDING-DEVICE> VISE1] [<PART> PART5] [<DIM> HEIGHT] ...
Op number> 0

Solution:
  1. <put-tool-on-milling-machine milling-machine1 spot-drill1>
  2. <put-holding-device-in-milling-machine milling-machine1 vise1>
  3. <clean part5>
  4. <put-on-machine-table milling-machine1 part5>
  5. <hold-with-vise milling-machine1 vise1 part5 side1 side3-side6>
  6. <drill-with-spot-drill-in-milling-machine milling-machine1 spot-drill1 vise1 part5 hole1 side1 side3-side6 1.375 0.25>
  7. <remove-tool-from-machine milling-machine1 spot-drill1>
  8. <put-tool-on-milling-machine milling-machine1 plain-mill1>
  9. <face-mill milling-machine1 part5 plain-mill1 vise1 side1 side3-side6 height 3 2>
compute-cost = 15
Do you want to try another solution?  no
```

Figure 3.6: Example of dialog with the interactive plan checker, slightly edited for presentation purposes. The underlined text corresponds to the expert's input. The plan being interactively improved is the solution plan to the problem introduced in Section 2.4.

```
[setting the initial state]
Op number> *
Input instantiated operator (no parens):  drill-with-spot-drill-in-milling-machine milling-machine1 spot-drill1 vise1 part5
hole1 side1 side3-side6 1.375 0.25

Checking step #<DRILL-WITH-SPOT-DRILL-IN-MILLING-MACHINE [<MACHINE> MILLING-MACHINE1] [<DRILL-BIT> SPOT-DRILL1] ...

Precondition #<HOLDING-TOOL MILLING-MACHINE1 SPOT-DRILL1> is false in state.

 Step #<DRILL-WITH-SPOT-DRILL-IN-MILLING-MACHINE [<MACHINE> MILLING-MACHINE1] [<DRILL-BIT> SPOT-DRILL1] [<HOLDING-DEVICE> ...

Op number> *
Input instantiated operator (no parens):  put-tool-on-milling-machine milling-machine1 spot-drill1

Checking step #<PUT-TOOL-ON-MILLING-MACHINE [<MACHINE> MILLING-MACHINE1] [<ATTACHMENT> SPOT-DRILL1]>.

Precondition #<IS-AVAILABLE-TOOL-HOLDER MILLING-MACHINE1> is false in state.
Testing lazy rule TOOL-HOLDER-AVAILABLE... Fired.
Adding #<IS-AVAILABLE-TOOL-HOLDER MILLING-MACHINE1> to the state.
Testing lazy rule TOOL-AVAILABLE... Fired.
Adding #<IS-AVAILABLE-TOOL SPOT-DRILL1> to the state.
Applying operator:
Adding #<HOLDING-TOOL MILLING-MACHINE1 SPOT-DRILL1> to the state.
Processing dependents...
Deleting #<IS-AVAILABLE-TOOL SPOT-DRILL1> from the state.
Deleting #<IS-AVAILABLE-TOOL-HOLDER MILLING-MACHINE1> from the state.
   Step #<PUT-TOOL-ON-MILLING-MACHINE [<MACHINE> MILLING-MACHINE1] [<ATTACHMENT> SPOT-DRILL1]> executed.

Testing eager rule PART-NOT-AVAILABLE-AND-HOLDING-DEVICE-NOT-EMPTY-AND-MACHINE-NOT-AVAILABLE.
Testing eager rule TABLE-AND-HOLDING-DEVICE-NOT-AVAILABLE.
Testing eager rule TOOL-AND-TOOL-HOLDER-NOT-AVAILABLE... Fired.
Testing eager rule MACHINE-NOT-AVAILABLE.
```

Figure 3.7: A different dialog for the same problem, showing the interactive plan checker in a verbose mode, and the behavior when an operator cannot be executed.

---

1. Run PRODIGY4.0 with the current set of control rules and obtain a solution S_p.
2. Show S_p to the expert.
   Expert provides new solution S_e possibly using S_p as a guide.
3. Test S_e. If it solves the problem, continue. Else go back to step 2.
4. Apply the plan quality metric to S_e.
   If it is better than S_p, continue. Else go back to step 2.
5. Compute the partial order $\mathcal{P}$ for S_e identifying the goal dependencies between plan steps.
6. Construct a problem solving trace corresponding to a solution S'_e that satisfies $\mathcal{P}$.
   This determines the set of decision points in the problem solving trace
   where control knowledge is missing.
7. Build the plan trees T'_e and T_p, corresponding respectively to the search trees for S'_e and S_p.
8. Compare T'_e and T_p explaining why S'_e is better than S_p, and build control rules.

Figure 3.8: Top level procedure to learn quality-enhancing control knowledge.

detail. As it was mentioned before, the subject of this chapter is learning quality-enhancing control knowledge represented as control rules. Chapter 4 describes a different way to represent such control knowledge and how it is learned. The only difference in the top-level algorithm of Figure 3.8 will be Step 8.

3.4 Constructing A Problem Solving Trace From The Plan

Once the expert has provided a correct and good plan, the planner is called in order to generate a *problem solving trace*, i.e. a sequence of decisions which if taken by PRODIGY would produce that plan. The goal of this process is to obtain a set of decision points where the default search heuristic, or the currently available control knowledge, needs to be overriden in order to guide PRODIGY4.0 towards the expert's good quality solution. Those decision points become the learning opportunities for our algorithm.

To force the planner to generate a given solution, the system uses PRODIGY4.0's signal and interrupt handling mechanism [Carbonell *et al.*, 1992]. The signal mechanism provides a way to run user-provided code at regular intervals, as often as once per node, during problem solving. To generate the given solution, PRODIGY4.0 starts searching for a plan using the default control heuristics and the available control knowledge. At each decision node a function called by the signal mechanism checks that the current alternative can be part of the desired solution. If the check fails, an interrupt is generated and PRODIGY4.0 is forced to backtrack. For example, at a bindings node the interrupt mechanism checks whether the instantiated operator is part of the expert's plan, and if not it tries a different set of bindings. To allow plans in which an operator with the same bindings occurs multiple times, the mechanism keeps track of the operators in the plan that have already been expanded, and in some cases determines whether the expansion order will eventually lead to the desired plan. PRODIGY is not actually told what to do – it is just told to try again at the appropriate divergence point from the expert's better solution. Once this expanded search produces the correct alternative, the node stores the alternative tried initially, that is, that suggested by the current control knowledge, and the desired one. Later on the learner will determine what control knowledge would be needed so that the correct alternative is explored first in future planning under similar conditions.

To determine the backtracking point when the alternative is rejected because it is not part of the desired plan, some dependency-directed heuristics are used in addition to PRODIGY4.0's default chronological backtracking. For example, if the order of operator application so far is not consistent with the expert's solution, backtracking may try a different goal ordering at an earlier node in the search tree. Due to PRODIGY4.0's nonlinear behavior and flexibility in the use of search strategies [Stone *et al.*, 1994], in particular different goal ordering and operator application ordering strategies, there may be several ways to generate a plan. The backtracking heuristics used by the algorithm described in this section prefer backtracking to earlier points on the search trace, and trying different goal orderings.

In some cases the algorithm does not require that the solution obtained be exactly the same as the one provided by the expert. PRODIGY4.0 includes an algorithm to extract a partially ordered graph from the totally ordered plan [Veloso, 1994, Veloso *et al.*, 1990] to capture the ordering constraints among the steps in the plan. For certain plan quality metrics, such as those additive on the cost of the operators in the plan, all the linearizations of the partial order of a plan are

equally acceptable because they have the same cost. The algorithm that constructs the search trace from the expert's plan is content with obtaining a trace for any of its linearizations. If n is the number of operators in the plan, p is the average number of preconditions, d is the average number of delete effects, and a is the average number of add effects of an operator, then the algorithm that generates the partial order runs in $O((p+d+a)n^2)$ [Veloso, 1994]. Empirical evidence shows that the partial-order generator runs in negligible time compared to the search time to generate the input totally ordered plan.

3.5 Building Plan Trees

At this point, two problem solving traces are available. One was obtained using the current control knowledge. The second corresponds to the generation of the better plan. QUALITY builds a *plan tree* from each of them. The nodes of a plan tree are the goals, operators, and bindings, or instantiated operators, considered by PRODIGY4.0 during problem solving in the successful search path that lead to that solution. The plan trees are built by translating the nodes in the successful path of the search trace into nodes in the plan tree. Abandoned paths that lead to backtracking are ignored. In the plan tree, a goal is linked to the operator considered to achieve the goal, the operator is linked in turn to its particular instantiation (bindings) chosen and the bindings are linked to the subgoals corresponding to the instantiated operator's preconditions.[7] Leaf nodes correspond to subgoals that were true when PRODIGY4.0 tried to achieve them.

The fact that a subgoal is true when PRODIGY4.0 tries to achieve it depends on which operators have been applied at that point. This information about the order of operator application is available from the problem solving trace but cannot be represented by just the parent child relationship between the plan tree nodes. For this reason the plan trees contain additional information in the form of achievement and deletion links.[8] Each leaf node records, in a `how-achieved` field, how the subgoal in the node was achieved. The content of that field is one of the following:

- The constant `:initial-state` if the subgoal was achieved because it was true in the initial state.
- A link to another goal node in the plan tree that corresponds to the same subgoal if it was achieved first by subgoaling and is still true.

[7] PRODIGY4.0 uses inference rules for planning as if they were operators, i.e. for achieving subgoals. When we mention operators in this context we refer to both operators and inference rules.

[8] If a subgoal is shared by two or more operators as a common precondition of them, it appears multiple times in the plan tree. For that reason we call them trees, as a node can only have one parent. Exceptions are nodes connected by achievement and deletion links.

- A link to a binding node whose application achieved the subgoal as a side effect.[9]

The goal nodes may also store `how-deleted` links to binding nodes whose application deleted the node subgoal as a side effect. Both the achievement and deletion links are bidirectional: a bindings node may contain `applied` links to the goal nodes corresponding to its side effects; a goal node may contain links `achieves-too` to other goal nodes corresponding to the same subgoal.

Figure 3.9 shows the plan trees corresponding to two plans for the same problem. The problem has two top-level goals, g_1 and g_2. The two plans are different in that op_2 and op_2' were chosen respectively to achieve goal g_2. op_2 and op_2' have different subgoals. Note that in plan tree (b) subgoal g_{12} is shared between the subtrees for the two top-level goals g_1 and g_2, and needs to be achieved only once. This information is stored in the form of a `how-achieved` link as shown in the figure. The reciprocal `achieves-too` link is not shown. On the other hand, in the plan tree (a) the subgoals for the two top-level goals are different (g_{12} and g_{22}) and more steps are needed to satisfy both of them. Recapitulating, the two plans differ in the operator and bindings chosen to achieve g_2. Those decisions are the ones for which the guidance provided by control knowledge is needed. They correspond to learning opportunities.

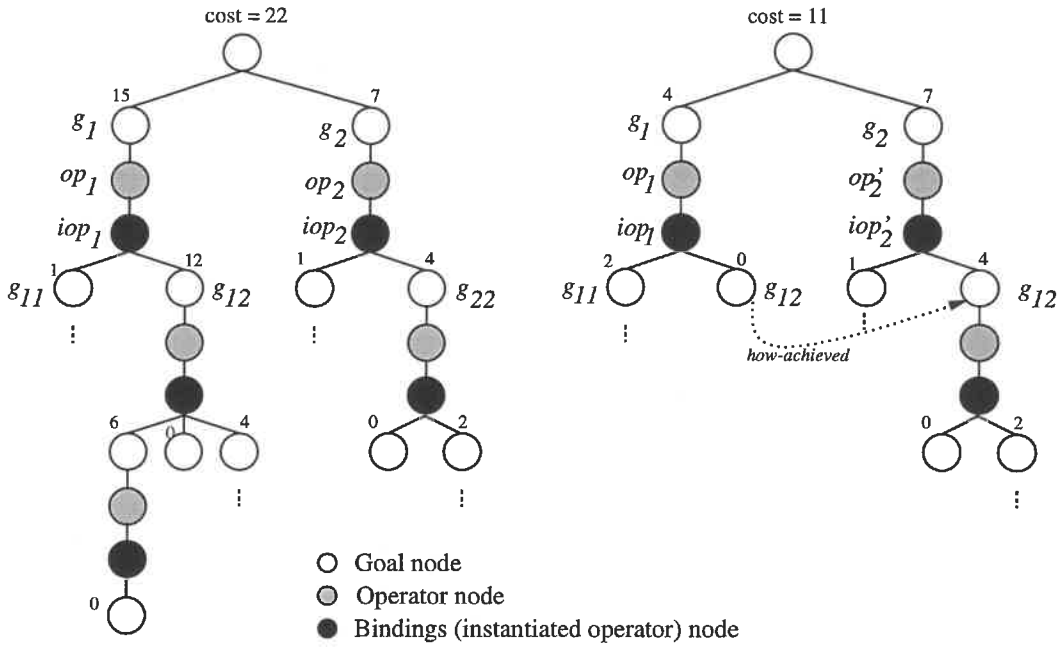
Assigning cost to the plan tree nodes: After the plan trees have been built, the plan quality metric is used to assign *costs* to their nodes, starting with the leaves and propagating them back up to the root. The box in Figure 3.9 summarizes how the cost computation is done. The leaf nodes have cost 0. A bindings (instantiated operator) node has the cost given by the quality metric, plus the sum of the costs of achieving its preconditions, i.e the costs of its children subgoals. Operator and bindings nodes have the same cost.[10] A goal node has the cost of the operator used to achieve it, i.e. of its child. Note that if a goal had to be reachieved, it has more than one child operator. Then the cost of achieving the goal is the sum of the costs of the children operators. Finally the root cost will be the cost of the plan. The plans for the plan trees in Figure 3.9 have different costs and plan tree (a) corresponds to the plan of worse quality.

3.6 Finding Learning Opportunities

The plan trees corresponding to the two traces are now available for the next step in the learning algorithm (Step 8 in Figure 3.8), namely generating the new control knowledge. The rationale of the algorithm is to explain why one solution is better than the other and transform this explanation into appropriate control knowledge that results in the generation of the better

[9] If the subgoal is a primary effect of the binding node, it means that the parent of the binding node is that subgoal, which is precisely the previous case.

[10] Inference rules have cost 0 as their application does not correspond to the performance of any actions, but rather the internal computation of the state's deductive closure.



(a) $Plantree_B$ (b) $Plantree_A$

e_{iop}: cost of instantiated operator iop, given by the plan quality metric
$c(n)$:cost associated with node n

$\forall$ goal node n_g, $c(n_g) = 0$ if n_g is a leaf node
$c(n_{op})$ if n_{op} is a child of n_g
$\forall$ operator node n_{op}, $c(n_{op}) = c(n_{iop})$ if n_{iop} is a child of n_{op}
$\forall$ bindings node n_{iop}, $c(n_{iop}) = e_{iop} + \sum_{n_g \in children(n_{iop})} c(n_g)$

(c)

Figure 3.9: (a) and(b) Plan trees corresponding to two solutions of different quality for the same problem. A number next to a node indicates the cost of the subtree rooted at that node. For clarity some nodes are not displayed. The cumulative cost of each subtree is propagated up to its ancestors. In the sections that follow the plan tree on the right, which corresponds to the better solution, will be called $plantree_A$. The plan tree on the left will be called $plantree_B$. (c) Computation of the cost of the plan trees.

solution in similar future situations. Figure 3.10 describes the top level view of the process. **Learn** is given the two plan trees and produces a set of control rules. Variable $plantree_A$ corresponds to the improved solution, and $plantree_B$ to the more costly solution, i.e. the one

obtained by the planner with the current control knowledge. In Step 2 the plan trees are compared in order to find the *learning opportunities*, i.e. reasons why the cost of the solution in $plantree_A$ is smaller than that of $plantree_B$. Then these opportunities are explored in Steps 7-11, possibly leading to the generation of control rules.

learn($plantree_A$,$plantree_B$)

1. *rule_structs* ← ∅
2. *learning_opportunities* ←
   analyze_goal(**root**($plantree_A$), **root**($plantree_B$)) ;; Figure 3.11
3. for each *learning_opportunity* ∈ *learning_opportunities*
4. g_A ← learning_opportunity.A(*learning_opportunity*)
5. g_B ← learning_opportunity.B(*learning_opportunity*)
6. *decision* ← learning_opportunity.decision(*learning_opportunity*)
7. case **dec_type**(*decision*)
8. :operator_or_bnds ;; Figure 3.13
9. *rule_structs* ← append(*rule_structs*, **learn_op_and_bnds_dec**(g_A,*decision*))
10. :goal ;; Figure 3.34
11. *rule_structs* ← append(*rule_structs*, **learn_goal_dec**(g_A,g_B,*decision*))
12. **create_rules**(*rule_structs*) ;; Figure 3.20

Figure 3.10: Top-level call to the learning mechanism once the plan trees have been built: finding learning opportunities given the plan trees and exploring them to create new control knowledge.

A learning opportunity is composed of a goal node g_A in $plantree_A$ which has cost 0, and a corresponding goal node g_B in $plantree_B$ with cost greater than 0. The different costs of g_A and g_B support the difference in cost between the two solutions. The algorithm does not exploit as learning opportunities cases in which $cost(g_A) < cost(g_B)$ due only to a difference in cost of the operators that achieve g_A and g_B. Section 3.6.2 discusses this limitation. A learning opportunity also includes a decision point in the problem solving process where the decision caused the planner to eventually expand those goal nodes and achieve them with such costs. Step 2 does this credit assignment process. It determines such pairs of goal nodes with cost divergence in the plan trees and analyzes them finding the relevant decision point for each of them. The process of constructing the problem solving trace from the good plan (described in Section 3.4) determined a set of decision points where the planner's decision needed to be overriden to lead to that plan. The credit assignment process prefers the earliest of those points relevant to the two goal nodes g_A and g_B. Earliest refers to the order in the sequence of planner's decision, not in the final linearized plan. After the learning opportunities are determined Steps 7-11 in Figure 3.10 utilize the type of the decision, operator and bindings choice, or a goal ordering, to select a particular learning mechanism.

To determine (Step 2) the learning opportunities, i.e. the goal nodes where there are cost

divergences, **analyze_goal**, in Figure 3.11, simultaneously traverses the plan trees in preorder, starting at their roots. At each call of **analyze_goal** the cost of two goal nodes, $q\_goal\_node_A$ and $q\_goal\_node_B$, is compared. The cost of a goal node is 0 (a) if that goal was true in the initial state of the problem, (b) if it was added by an operator chosen to achieve another occurrence of the same goal in a different subtree, or (c) if it was added as a side effect of an operator relevant to another goal. Note that this information is stored in the plan tree as links between the nodes (see Section 3.5).

Two costs are associated with nodes of a plan tree:

- **q_node_cost**(*plantree_node*): the cost associated with a plan tree node. It corresponds to the cost of the subtree rooted at that node. Section 3.5 explained how this cost is obtained.
- **op_cost**(*plantree_operator_node*): the cost that the plan quality metric assigns to the operator expanded in *plantree_operator_node*. In general it is applicable at bindings nodes (i.e. the quality metric applies to instantiated operators).

Analyze_goal stops traversing the plan trees when the cost of achieving $q\_goal\_node_A$ is 0 and the cost of achieving $q\_goal\_node_B$ is greater than 0 (Steps 5-11). A learning opportunity is detected in this case, and **relevant_decision** associates with $q\_goal\_node_A$ and $q\_goal\_node_B$ a problem solving decision that lead to their expandsion as subgoals. This is the only case where the learner is able to exploit a learning opportunity.

Analyze_goal also stops traversing the plan trees when the cost of $q\_goal\_node_A$, which corresponds to the good quality solution, is greater than that of $q\_goal\_node_B$, a node in the plan tree for the lesser quality solution (Steps 12-14). Obviously, this cost divergence cannot explain why solution A is better than solution B, and the learner ignores it and stops exploring those subtrees.[11] Instead, it tries to find a different branch of the plan tree to explain the cost divergence. **Analyze_goal** also stops when the cost improvement is due the difference of the cost of the individual operators chosen (**child**($q\_goal\_node_A$) and **child**($q\_goal\_node_B$) respectively (Step 13) and not to the difference of cost of achieving those operators preconditions. The learning algorithm is not able to deal with this case and nothing is learned.

In the only remaining case (Steps 15-22), the cost of the subtree for $q\_goal\_node_A$ is smaller than the cost of the subtree for $q\_goal\_node_B$. This means that further exploration (or plan tree traversal) may detect an explanation of the better quality of A. Therefore **analyze_goal** is called recursively with the preconditions of the operators **child**($q\_goal\_node_A$) and **child**($q\_goal\_node_B$) used to achieve $q\_goal\_node_A$ and $q\_goal\_node_B$ respectively. **Comparable_preconds_p** matches pairs of preconditions of those operators that have a comparable cost.

[11] Ignoring this condition might cause inaccurate overgeneralization if in future similar episodes the difference in cost between $q\_goal\_node_A$ and $q\_goal\_node_B$ could be larger than the savings obtained by taking the suggested guidance.

analyze_goal($q\_goal\_node_A, q\_goal\_node_B$) ;; traverses plantrees in preorder

1. $op\_cost_A \leftarrow$ **op_cost**(**child**($q\_goal\_node_A$)) ;; cost according to quality metric
2. $op\_cost_B \leftarrow$ **op_cost**(**child**($q\_goal\_node_B$))
3. $subtree\_cost_A \leftarrow$ **q_node_cost**($q\_goal\_node_A$) ;; cost of the subtree rooted at node
4. $subtree\_cost_B \leftarrow$ **q_node_cost**($q\_goal\_node_B$)
5. if $subtree\_cost_A = 0 \wedge subtree\_cost_B > 0$
   then
6. $learning\_opportunity \leftarrow$;; this is a learning opportunity
7. make_learning_opportunity
8. :decision **relevant_decision**($q\_goal\_node_A, q\_goal\_node_B$)
9. :A $q\_goal\_node_A$
10. :B $q\_goal\_node_B$
11. return({$learning\_opportunity$})
12. else if $subtree\_cost_A \geq subtree\_cost_B$;; if subgoal is cheaper in expensive plan tree
13. $\vee\ (op\_cost_B - op\_cost_A) \geq (subtree\_cost_B - subtree\_cost_A)$
    ;; or if improvement is due only to different operator cost
    then
14. return(nil) ;; don't explore further
    else
15. $learning\_opportunities \leftarrow \emptyset$
16. for each <$subgoal\_qnode_A$,$subgoal\_qnode_B$>
17. such that $subgoal\_qnode_A \in$ **children**(**child**($q\_goal\_node_A$))
18. $\wedge\ subgoal\_qnode_B \in$ **children**(**child**($q\_goal\_node_B$))
19. $\wedge$ **comparable_preconds_p**($subgoal\_qnode_A$,$subgoal\_qnode_B$)
20. $learning\_opportunities \leftarrow$
21. append($learning\_opportunities$, **analyze_goal**($subgoal\_qnode_A$,$subgoal\_qnode_B$))
22. return($learning\_opportunities$)

Figure 3.11: Traversing the plan trees to detect learning opportunities. *Q_goal_node* refers to a plan tree goal node. **Q_node_cost**(*node*) is the cost associated with *node* in the plan tree. **Op_cost**(*operator_node*) is the cost that the plan quality metric assigns to the operator expanded in *operator_node*.

Note that if the operators are different their preconditions may be so, and the algorithm is not able to analyze them. If the operators differ on their bindings, or have similar preconditions, **analyze_goal** is able to proceed down the plan trees. This analysis of related preconditions is done in a domain independent way. For example, in the case of the two alternative operators whose effect is drilling a hole in a part, namely drilling in the milling machine and drilling in the drill press, their preconditions are very similar, as they require that both the part and an

appropriate tool have been placed on the corresponding machine.

3.6.1 An Example

We now illustrate this process with the plan trees in Figure 3.9. The plan tree on the right corresponds to the better quality solution, $plantree_A$ in the algorithms sketched above. **Analyze_goal** is first called with the roots of both plan trees and traverses them recursively in preorder. The cost of the goal nodes corresponding to g_1 are first compared. As the cost is smaller in $plantree_A$, the traversal proceeds on their subtrees (Steps 15-22 of **analyze_goal**). The cost of g_{11} is higher in $plantree_A$ than in $plantree_B$. This corresponds to Step 12 and is one of the cases that the learner does not explore further, as it does not support in explaining why solution A is better than solution B.

Analyze_goal proceeds traversing the plan trees. g_{12} has cost 0 in $plantree_A$ and cost 12 in $plantree_B$. There are links in $plantree_A$ between the two occurrences of g_{12}, which indicate that the cost is 0 because it had already been achieved as a precondition of iop'_2. According to the algorithm (Steps 5-11) a learning opportunity is detected. **Relevant_decision** assigns credit for the 0 cost of g_{12} to the choice that the planner made of op'_2 over op_2 to achieve top level goal g_2. The learning opportunity built is $<g_{12_A}, g_{12_B}, dec\_point_{op_2,op'_2}>$ where g_{12_A} is the plan tree node for g_{12} with cost 0 in $plantree_A$, g_{12_B} is the plan tree node for g_{12} with cost greater than 0 in $plantree_B$, and $dec\_point_{op_2,op'_2}$ is the problem solving decision point, where op_2 should be preferred over op'_2.

The plan tree traversal continues now comparing the cost of g_2 in both trees. As they are found equal (Step 12), **analyze_goal** terminates returning the only learning opportunity detected.

3.6.2 Why These Learning Opportunities

Figure 3.12 presents a high level, informal description of the explanation: the learner finds an explanation of why solution A is of better quality than solution B by finding instances of nodes in the plan trees that satisfy the description in the figure and then expressing it in *operational* terms. This is a key point of the learning process. The description is operational if it can be used at problem solving time, i.e. at the decision point captured by the learning opportunity. These operationalized descriptions will be the core of the left-hand sides of the newly learned rules and will be tested at future search decision points to guide the planner towards the better solution. The next section describes how the rules are created.

The learning algorithm exploits only a limited class of learning opportunities: those cases in which $cost(g_A) = 0$ and $cost(g_B) > 0$. The algorithm cannot learn from other cases in which $cost(g_A) < cost(g_B)$, due only to a different cost of the operators used to achieve g_A and g_B

Solution A is better (cheaper) than solution B because
$q\_goal\_node_B$ is a subgoal with cost greater than 0
$\wedge$ $q\_goal\_node_A$ is a subgoal and has cost 0
because it was:
a) true in the initial state, or
b) the precondition of some operator, and was achieved as such, or
c) achieved as a side effect of an operator chosen for another goal

Figure 3.12: An informal description of the explanation that underlies the algorithms presented.

(Step 13). Additionally the algorithm cannot learn when those operators have the same cost but it cannot find comparable preconditions (Step 19). These cases limit the kinds of quality metrics for which the algorithm is suited to those in which the improvements in quality, i.e. savings in plan cost, are due to sharing the work among different parts of the plan. At planning time this translates in sharing subgoals among plan operators by making appropriate operator and bindings choices. These sources of improvement are common in many domains. Process planning is a good example because sharing of subgoals maps into sharing of parts or subparts of a set-up, and the number of set-ups is frequently related to the quality of the plan.

3.7 Learning Operator and Bindings Control Rules

The last section described how the plan trees are traversed and compared in order to find learning opportunities. This and the coming sections present the algorithms that exploit those learning opportunities in order to build quality-enhancing search control rules. Although the examples used to illustrate these algorithms are mostly taken from the process planning domain, the algorithms are *domain-independent*.

Each learning opportunity has associated a decision point where the learner detected a gap in the current control knowledge. This is the time to reason about how to fill those gaps, and the learner, in Steps 7-12 of Figure 3.10, selects a particular mechanism to explore each of them. The choice depends on the kind of control knowledge that is needed at the decision point, namely to learn operator and bindings control rules, or to learn goal-ordering control rules. Section 3.9 describes the procedure to learn goal-preference control rules. This section describes the procedure to learn operator and binding control rules. Note that PRODIGY4.0 distinguishes clearly between choosing an operator schema relevant to a goal and choosing a way to instantiate that schema among several possible instantiations. Therefore there are distinct types of control rules for these distinct decisions. However, conceptually, and for the sake of plan quality, both decisions are sufficiently related to merit simultaneous consideration. The cost of a plan step and its influence on the cost of the rest of the plan depends equally on

the operator schema choice and on the bindings choice.

Figure 3.13 describes the top level function of the mechanism to learn operator and bindings control rules. This function, **learn_op_and_bnds_dec** is called for each learning opportunity <g_A, g_B, *dec_point*> with the goal node g_A of $plantree_A$, and the decision point. The function will determine which conditions are available at the decision point to explain both *why* g_A *became a subgoal* and *why it was achieved with cost 0.* This is done by traversing the tree up, and can be seen as propagating those facts up the plan tree to the decision point. Figure 3.12 showed the reasons why the cost of g_A may be 0, while the cost of the corresponding goal node g_B in $plantree_B$ was greater than 0. Steps 1-13 of **learn_op_and_bnds_dec** explore them finding the starting points to traverse the plan tree as follows:

- g_A had cost 0 because it was true in the initial state (Steps 2-5): **learn_op_and_bnds_dec** will go up the plan tree starting on g_A to justify why g_A was expanded as a subgoal. Every decision in the path up from g_A where an operator was chosen is responsible for g_A becoming a subgoal and supports this explanation. The bindings that appear in g_A are relevant to this explanation, and are also propagated up. Every decision up the tree that chose one of those bindings among other alternatives is justified in this way. The

---

learn_op_and_bnds_dec(g_A, *dec_point*) ;;g_A is a q-goal-node

1. case type(**how_achieved**(g_A))
2. :initial_state
3. $g'_A \leftarrow g_A$
4. *rel_bnds* $\leftarrow$ **all_bnds**(g_A)
5. *type* $\leftarrow$:initial_state
6. :subgoaled
7. $g'_A \leftarrow$ **goal_to_propagate_up**(g_A,**goal_how_achieved**(g_A),*dec_point*)
8. *rel_bnds* $\leftarrow$ **all_bnds**(g_A)
9. *type* $\leftarrow$:subgoaled
10. :side_effect
11. $g'_A \leftarrow$ **q_goal_that_caused_side_effect**(g_A)
12. *rel_bnds* $\leftarrow$ **bnds_that_caused_side_effect**(g_A)
13. *type* $\leftarrow$:subgoaled
14. *rule_structs* $\leftarrow$
    propagate_conditions_up(g'_A, *rel_bnds*, *dec_point*, g'_A, *type*, $\emptyset$) ;; Figure 3.14
15. return(*rule_structs*)

Figure 3.13: Learning operator and bindings control rules.

propagation terminates upon reaching *dec_point* and its result is the set of features of the meta-state at *dec_point* that lead g_A have cost 0, as explained below.

- g_A had cost 0 because it was also a precondition of another operator and was achieved by subgoaling (Steps 6-9): this is detected by following the links that indicate how g_A was achieved. **Goal_to_propagate_up** finds the goal node at which to start propagating up: either g_A or the goal node where it was first achieved. It chooses the node whose expansion was due to the choice made at the decision point. All the bindings that appear in the goal node are relevant, as in the first case.

  Figure 3.9 can be used to illustrate this process. In the previous section we explained how the learning opportunity $<g_{12_A}, g_{12_B}, dec\_point_{op_2,op_2'}>$ was found. g_{12} in the plan tree to the right was achieved by subgoaling when it was expanded as a child precondition of iop_2'. **Goal_to_propagate_up** returns that child g_{12} because its expansion was indirectly due to the choice at the decision point $dec\_point_{op_2,op_2'}$.

- g_A had cost 0 because it was achieved as a side effect of an operator *op* relevant to another goal g_A' (Steps 10-13): **q_goal_that_caused_side_effect** returns g_A'. The relevant bindings are those bindings of *op* that appear in g_A' and caused the side effect g_A.

Once **learn_op_and_bnds_dec** determines the goal g_A' and the relevant bindings to start the traversal up the plan tree, **propagate_conditions_up** is called in Step 14. Figure 3.14 describes it. The goal is to find the features of the meta-state at *dec_point* that are relevant to g_A becoming a subgoal with cost 0. A propagation step (Steps 11-14) consists of mapping the relevant bindings for g with the variable names of the operator *op* that introduced it (**propagate_backwards**), pruning the bindings introduced by that operator, i.e. those that do not come from the goal the operator is relevant for, and propagating them up in the recursive call. There may be other goal nodes dep_g achieved by achieving g and therefore with cost 0. As the algorithm is explaining why g_A has cost 0 each dep_g is also used to propagate conditions up (Steps 15-16), because its 0 cost depends on both dep_g and g becoming subgoals.

To illustrate this mechanism with an example, Figure 3.15 shows part of the plan tree for the better quality solution of a process planning problem. Assume `part5` is on the milling machine table, i.e. `(on-table part5 mm1)` is true in the state and was achieved with cost 0, and **propagate_conditions_up** is called with it as the initial value of g. Assume as well that the *dec_point* corresponds to the choice of *drill-with-spot-drill-in-milling-machine* over a different operator. Both the part and the machine are relevant arguments of `on-table` for it to be true in the state. Therefore the bindings for `<mach>` and `<part>` are propagated up (marked in gray in the figure). All the bindings for the operator `hold-with-vise` came from the right-hand side, i.e. from its parent goal `holding` (Step 12); the operator did not introduce any new bindings. Of all the arguments of holding, only the machine and the part are relevant for `(on-table`

propagate_conditions_up (g, *rel_bnds*, *dec_point*, g'_A, *type*, *rule_structs*)

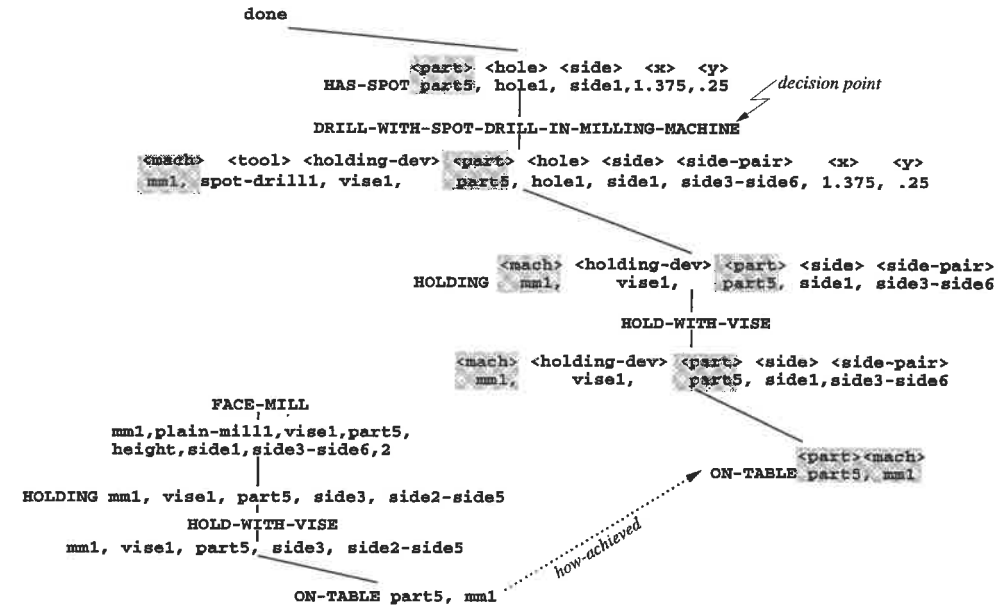
1. if g = *root_of_plantree*
2. return (*rule_structs*) ;;not in dec path
3. else if **parent**(g) = *dec_point*
4. $goal\_precond \leftarrow$ **prepare_goal_precond**($p, type, dec\_point$) ;; Figure 3.16
5. *rule_structs* $\leftarrow$;; Figure 3.19
   cons(**create_rule_struct**($goal\_precond$,*dec_point*,*rel_bnds*), *rule_structs*)
6. return (*rule_structs*)

   else
7. *op* $\leftarrow$ **parent_op**(g)
8. if **decision_was_made_p**(*op*)
   then
9. $goal\_precond \leftarrow$ **prepare_goal_precond**($p, type, op$) ;; Figure 3.16
10. *rule_structs* $\leftarrow$
    cons (**create_rule_struct**($goal\_precond$, *op*, *rel_bnds*), *rule_structs*)
11. $\sigma \leftarrow$ **propagate_backwards**(g, *op*)
12. *introduced_bnds* $\leftarrow$ **op_vars_and_bnds**(*op*) $\setminus$ **rhs-bnds**(*op*,**parent_goal**(*op*))
13. $g'_A \leftarrow$ **apply_substitution**(σ, g'_A)
14. *rel_bnds* $\leftarrow$ **apply_substitution**(σ, *rel_bnds* $\setminus$ *introduced_bnds*)
15. for each $dep_g \in$ **goal_achieves_too**(g)
16. *rule_structs* $\leftarrow$
    propagate_conditions_up(dep_g, *rel_bnds*, *dec_point*, g'_A, *type*, *rule_structs*)
17. $g \leftarrow$ **parent_goal**(*op*)
18. goto 1

Figure 3.14: Traversing the plan tree to propagate the relevant conditions.

`part5 mm1)` becoming a subgoal (Step 14). They are propagated up. At that point, the choice of drill-with-spot-drill-in-milling-machine corresponds to a decision point. The bindings for `<part>` came from the right-hand side of the operator, i.e from its parent goal `(has-spot part5 hole1...)`. However the binding for the machine was introduced by the operator and the learned control knowledge will guide the choice of that binding. In addition, as `(on-table part5 mm1)` has an achievement link to the subtree for another goal, the propagation continues on that path too (Steps 15-16).

The propagation stops when it reaches the root of the plan tree, or the *dec_point*. There may be also other decision points where a decision was made to force PRODIGY4.0 into the good plan. **Decision_was_made_p** in Step 8 detects whether the current node of the plan tree corresponds to one of those operator or binding decisions. Each of these points becomes a candidate for



(a)

```
(control-rule prefer-drill-with-spot-drill-in-milling-machine9
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (known (on-table <machine> <part>))
          (type-of-object <machine> milling-machine)))
 (then prefer operator drill-with-spot-drill-in-milling-machine
                       drill-with-spot-drill))

(control-rule prefer-bnds-drill-with-spot-drill-in-milling-machine10
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (current-operator drill-with-spot-drill-in-milling-machine)
          (known (on-table <machine-2> <part>)) (diff <machine-2> <machine-1>)))
 (then prefer bindings ((<machine> . <machine-2>)) ((<machine> . <machine-1>))))
```

(b)

Figure 3.15: (a) Part of the plan tree for the better quality solution of a process planning problem used to illustrate how **propagate_conditions_up** works. The grayed arguments correspond to the bindings propagated up from `on-table`. Those bindings were relevant for the subgoal to have cost 0. They are propagated up to the instantiated *drill* operator node, which introduced those variables and corresponds to a decision point. (b) An operator preference and bindings preference rules that would be learned from that episode. The next sections describe in detail how the rules are built.

learning control knowledge (Steps 3-5 and 8-10). Note that not only is *dec_point* a learning opportunity but so are other decisions made after it on the path that lead to g'_A's expansion. At each of those points, information is recorded about what conditions of the planner's meta-state are relevant to the explanation. This information includes:

- A control rule precondition expression, computed by **prepare_goal_precond** (in Fig-

prepare_goal_precond(p, $type$, dec_point)

```
1. case type
3.   :initial_state   return(<known op_precond_exp(p)>)
4.   :subgoaled       operationalize(p, dec_point)        ;; Figure 3.17
```

Figure 3.16: Computing a control rule precondition that justifies why g_A had cost 0.

ure 3.16), that captures why g_A had cost 0, namely if it was true in the state, it was a pending goal,[12] or one of its ancestors was a pending goal. The later case is determined by **operationalize**, described in Figure 3.17. In this way the explanation is made *operational* to the decision point, this meaning that it is transformed in a condition that can be tested in the problem solver's meta-state at the time when the relevant decision should be made.

- A determination of which of the variable bindings chosen at each decision point are relevant to g_A becoming a 0 cost subgoal. These relevant bindings are a subset of the arguments of g_A. In the example of Figure 3.15, the values of `<machine>` and `<part>` were relevant at the decision point so that `(on-table part5 mach1)` had cost 0.

operationalize(p, dec_point)

```
1. if current_goal_p(p, dec_point)
2.       return(nil)
3. else if pending_goal_at_dec_point(p, dec_point)
4.       return(<pending-goal op_precond_exp(p)>)
   else
5.       p ← parent_goal(op)
6.       operationalize(p, dec_point)
```

Figure 3.17: Operationalizing why g_A was a subgoal and had cost 0, i.e. representing it as an expression that can be tested in the planner's current state and meta-state at the decision point. The meta-state includes knowledge of the subgoaling links and the operators expanded. A goal is pending if its parent binding node has been expanded. The function in this figure finds out whether p was the *current* goal or a *pending* goal at the decision point for which learning was invoked. Both conditions can be checked at that point, i.e. they are operational.

[12] A *pending goal* is a goal pending to be achieved for which the planner has still not chosen a relevant operator. PRODIGY4.0's search algorithm maintains the set of pending goals at each point, and chooses the next goal to work on from that set, thus displaying nonlinear character. The meta-predicate `pending-goal` checks whether its argument is a pending goal.

static_precs(*op,rel_bnds,goal_precond*)

1. *relevant_op_static_precs* ← **relevant_static_precs_for_bnds**(**op_static_precs**(*op*),*rel_bnds*)
2. *preconds* ← *relevant_op_static_precs* ⋃ {*goal_precond*}
3. *relevant_op_vars* ← **op_vars_that_appear_in**(*rel_bnds*)
4. ⋃ **op_vars_that_appear_in**(*preconds*)
5. *type_precs* ← ∅
6. for each *rel_var* ∈ *relevant_op_vars*
7. *var_type_in_prec* ← **signature_type**(*rel_var,preconds*)
8. *var_type_in_op* ← **op_var_spec**(*rel_var, op*)
9. if *var_type_in_op* ⊂ *var_type_in_prec*
10. push(<`type-of-object` *rel_var var_type_in_op*>, *type_precs*)
11. return (append (*relevant_op_static_precs, type_precs*))

Figure 3.18: Computing constraints on the type and value of the relevant bindings. These constraints must be satisfied for g_A (captured in *goal_precond*) to become a subgoal of cost 0.

- Constraints on the type and value of those relevant bindings. These constraints are computed by **static_precs** in Figure 3.18. Each predicate in a domain requires that its arguments are of a certain type. We call this the predicate's *signature*, and it is computed at domain creation time by looking at all the operators in which the predicate appears as a precondition or effect. Given a variable in a predicate, **signature_type** in Step 7 returns the possible types it can belong to. The allowed types for a variable can be further constrained if the variable is introduced by a particular operator (Step 8). If the operator-specified type (or combination of types) is more specific than the type required by the signature of the conditions in which the variable appears, a type constraint will become part of the learned rule (Steps 9-10). Summarizing, **static_precs** computes the constraints that the variables in the relevant bindings must satisfy for g_A to become a subgoal of cost 0. These constraints are obtained by looking at the operator's type specification and static preconditions.[13]

  Back to the example of Figure 3.15, in this domain the signature of `holding` allows any type of machine as an argument, because `holding` appears as precondition or effect of all kinds of machining operations. However, for `(on-table <part> <mach>)` to be shared by the face-mill subtree and the drill subtree, the value must be of type milling machine.

All this information is stored in a data structure described in Figure 3.19, which will be later used to generate control rules. Once all the learning opportunities have been explored, the

[13]Static preconditions are those that cannot be added or deleted by operators. Therefore they are true only if they are present in the initial state, or if they are added to the initial state by inference rules supported only by other static facts. The operator static preconditions are used to prune the possible values of the operator variables.

create_rule_struct(*goal_precond,dec_point,rel_bnds*)

```
1.  make_rule_struct
2.        :current_goal     parent_goal(dec_point)
3.        :current_op       op
4.        :good             decision.good(dec_point)
5.        :bad              decision.bad(dec_point)
6.        :relevant_bnds    rel_bnds
7.        :goal_precond     goal_precond
8.        :other_preconds   static_precs(op,rel_bnds)          ;; Figure 3.18
9.        :decision_point   dec_point
10.       :rule_type        decision.type(dec_point)           ;; :operator or :bindings
```

Figure 3.19: Storing in a data structure the information needed to build an operator and/or bindings control rule.

create_rules (*rule_structs*)

```
1.  rule_structs ← merge_rule_structs_for_same_dec_point(rule_structs)
2.  for each rule_struct ∈ rule_structs
3.      case rule_struct.type(rule_struct)
4.          :operator
5.                  create_operator_rule(rule_struct)
6.                  create_bnds_rule(rule_struct)
7.          :bindings
8.                  create_bnds_rule(rule_struct)
9.          :goal
10.                 create_goal_prefer_rule(rule_struct)
```

Figure 3.20: Building control rules.

control rules are created (Step 12 of Figure 3.10). **Create_rules** is described in Figure 3.20. In Step 1 all the structures in *rule_structs* corresponding to the same decision point are merged in a single one. The merging corresponds to a conjunction of the conditions. Although each of the rule structures supports the decision, the better solution has less cost because all of those conditions happened in this particular learning episode, and each one of them may be not sufficient to make a decision in a similar problem. We take here a conservative approach, preferring to err on the side of overly-specific rules.

One or more control rules are built from each of the resulting structures. If the structure corresponds to an operator decision, it will support the choice of both the operator and its particular instantiation. The relevant bindings stored in the structure give reasons to choose

bindings for the operator. Therefore the learner creates an operator preference rule and a bindings preference rule (Steps 4-6). If the structure corresponds to a bindings decision, only a bindings preference rule is created (Steps 7-8). The operator and bindings rules are created by filling the templates in Figures 3.21 and 3.22 respectively. Note that these templates use only the results of the learning algorithms and are justified by the learning target, namely guide the planner towards better plans. It is important to note that *they are independent of the application domain.* Section 3.12 explains why preference rules are created instead of select rules. Appendix A describes the syntax of preference control rules in PRODIGY4.0 and how they are used.

create_operator_rule(*rule_struct*)

```
(control-rule prefer-op-rule_struct.good(rule_struct)-n
   (if (and (current-goal rule_struct.current_goal(rule_struct))
            rule_struct.goal_precond(rule_struct)
            rule_struct.other_preconds(rule_struct) )
   (then prefer operator rule_struct.good(rule_struct)
                         rule_struct.bad(rule_struct))
```

Figure 3.21: Template to automatically create an operator preference control rule using the information gathered by **learn_op_and_bnds_dec**. Note that this template and the one in the next figure are *independent* of the application domain. The learned rules will become dependent on the domain when the template is filled with the results of the learning algorithm.

3.8 Example Of Learning Operator And Bindings Rules

The example in Section 2.4 will be used to illustrate how operator and bindings control rules are learned. Assume that the planner obtained initially plan (a) in Figure 2.6. This solution was improved by interaction with a human expert (Figure 3.6) leading to plan (b) in Figure 2.6. Then a problem solving trace was constructed from the improved plan as explained in Section 3.4. Figures 3.23 and 3.24 show part of the problem solving traces for both solutions. The differences between the two plans correspond at problem-solving time with an operator decision (prefer drill-with-spot-drill-in-milling-machine over drill-with-spot-drill) and an instantiation decision (bindings for drill-with-spot-drill-in-milling-machine). The search traces corresponding to plans (a) and (b) are transformed in the plan trees shown in Figure 3.25.

The learner first determines the learning opportunities by traversing the plan trees using the algorithm in Figure 3.11. **Analyze_goal** does not explore the right subtrees of both plan trees because they have equal cost. It goes down the left subtrees because the difference in the

create_bnds_rule(*rule_struct*)

```
(control-rule prefer-bindings-rule_struct.good(rule_struct)-n
   (if (and (current-goal rule_struct.current_goal(rule_struct))
            (current-operator rule_struct.good(rule_struct))
            rule_struct.goal_precond(rule_struct)
            rule_struct.other_preconds(rule_struct) )
   (then prefer bindings generate_bnds_pairs(rule_struct.relevant_bnds(rule_struct))) )
```

where the result of **generate_bnds_pairs** has the form

```
((<op-var-1> . <good-var-1>) ... (<op-var-n> . <good-var-n>))
((<op-var-1> . <bad-var-1>) ... (<op-var-n> . <bad-var-n>))
```

Figure 3.22: Template to automatically create a bindings preference control rule using the information gathered by **learn_op_and_bnds_dec**. **Generate_bnds_pairs** produces two binding lists using the variable names that appear in the operator (`<op-var-i>`) and the variables that appear in the control rule learned preconditions. Only the variables that were determined as relevant, i.e. those in **rule_struct.relevant_bnds** are used, since those are the ones the rule must generate bindings for.

```
  2 n2 (done)
  4 n4 <*finish* part5>
  5   n5 (size-of part5 height 2) [1]
  7   n7 <face-mill
        milling-machine1 part5 plain-mill1 vise1 side1 side3-side6 height 3 2> [7]
Firing delete goals EXPAND-MAIN-GOALS-FIRST
  8     n8 (shape-of part5 rectangular) [1]
 10     n10 <is-rectangular part5>
Firing delete goals EXPAND-MAIN-GOALS-FIRST
 11   n11 (has-spot part5 hole1 side1 1.375 0.25)
 13   n13 <drill-with-spot-drill
         drill1 spot-drill1 vise1 part5 hole1 side1 side2-side5 1.375 0.25> [1]
 14     n14 (holding-tool drill1 spot-drill1) [3]
 16     n16 <put-in-drill-spindle drill1 spot-drill1>
...
```

Figure 3.23: Beginning of the problem solving trace that obtained the plan of cost 28 (plan (a) of Figure 2.6).

subtrees cost supports the fact that $plantree_A$ is better than $plantree_B$. Step 12 of **analyze_goal** (Figure 3.11) decides to abandon the exploration of the `holding-tool` subtrees because the cost of the subtree in $plantree_A$ is greater than the cost of the subtree in $plantree_B$. **Analyze_goal** stops when it finds that the cost of holding the part as a precondition of the face mill operator is 0 in $plantree_A$ and greater than 0 in $plantree_B$. The goal nodes corresponding to this `holding` subgoal become a learning opportunity. In Step 8 **relevant_decision** assigns credit for the 0 cost of holding the part to the problem solving decision point in which the planner chose operator

```
  2 n2 (done)
  4 n4 <*finish* part5>
  5   n5 (size-of part5 height 2) [1]
  7   n7 <face-mill
        milling-machine1 part5 plain-mill1 vise1 side1 side3-side6 height 3 2> [7]
Firing delete goals EXPAND-MAIN-GOALS-FIRST
  8     n8 (shape-of part5 rectangular) [1]
 10     n10 <is-rectangular part5>
Firing delete goals EXPAND-MAIN-GOALS-FIRST
 11   n11 (has-spot part5 hole1 side1 1.375 0.25)
Op #<OP: DRILL-WITH-SPOT-DRILL> was not in the solution proposed by the expert.
Backtracking to make new operator decision at node 12.
 13   n15 <drill-with-spot-drill-in-milling-machine
        milling-machine1 spot-drill1 vise1 part5 hole1 side1 side2-side5 1.375 0.25> [1]
Op #<DRILL-WITH-SPOT-DRILL-IN-MILLING-MACHINE
[<MACHINE> MILLING-MACHINE1] [<DRILL-BIT> SPOT-DRILL1] [<HOLDING-DEVICE> VISE1] [<PART>
PART5] [<SIDE> SIDE1] [<SIDE-PAIR> SIDE2-SIDE5] [<LOC-X> 1.375] [<LOC-Y> 0.25] [<HOLE> HOLE1]>
was not in the solution proposed by the expert.
Backtracking to make new binding decision at node 15.
 12   n14 drill-with-spot-drill-in-milling-machine
 13   n17 <drill-with-spot-drill-in-milling-machine
        milling-machine1 spot-drill1 vise1 part5 hole1 side1 side3-side6 1.375 0.25>
 14     n18 (holding-tool milling-machine1 spot-drill1) [2]
 16     n20 <put-tool-on-milling-machine milling-machine1 spot-drill1>
...
```

Figure 3.24: Beginning of the problem solving trace to obtain the better quality plan (plan (b) of Figure 2.6). This trace was constructed starting with the improvements to the plan suggested by the human expert. The operator initially chosen at node `n11` is not in that plan. PRODIGY4.0 backtracks trying a different operator instead. Similarly the instantiation of the operator at `n15` is not in the improved plan. PRODIGY4.0 backtracks again to try a different instantiation. Later, control rules will be automatically acquired which, should have they been present, would have lead to generating the better plan first.

drill-with-spot-drill-in-milling-machine over operator *drill-with-spot-drill* in order to obtain the improved plan. This decision point was stored during the construction of the problem solving trace for the better plan.

Next **learn_op_and_bnds_dec** (Figure 3.13) explores this learning opportunity. g_A is the node for the holding precondition of face-mill in $plantree_A$. g_A was also the precondition of another operator, drill-with-spot-drill-in-milling-machine, in the subtree on the right. This is detected by following the **goal_how_achieved** links in the plan tree (Step 1). That precondition was achieved by subgoaling. In Step 7 **goal_to_propagate_up** returns that subgoal (g'_A) because its expansion was due to the choice of the drill operator at the decision point. All the bindings of `(holding mm1 vise1 part5 side1 side3-side6)` are relevant so that the two subgoals are shared (Step 8). Finally in Step 14 **propagate_conditions_up** (Figure 3.14) is called with g_A and all the bindings of holding marked as relevant.

Only one propagation step is needed in this example, and the propagation terminates at node

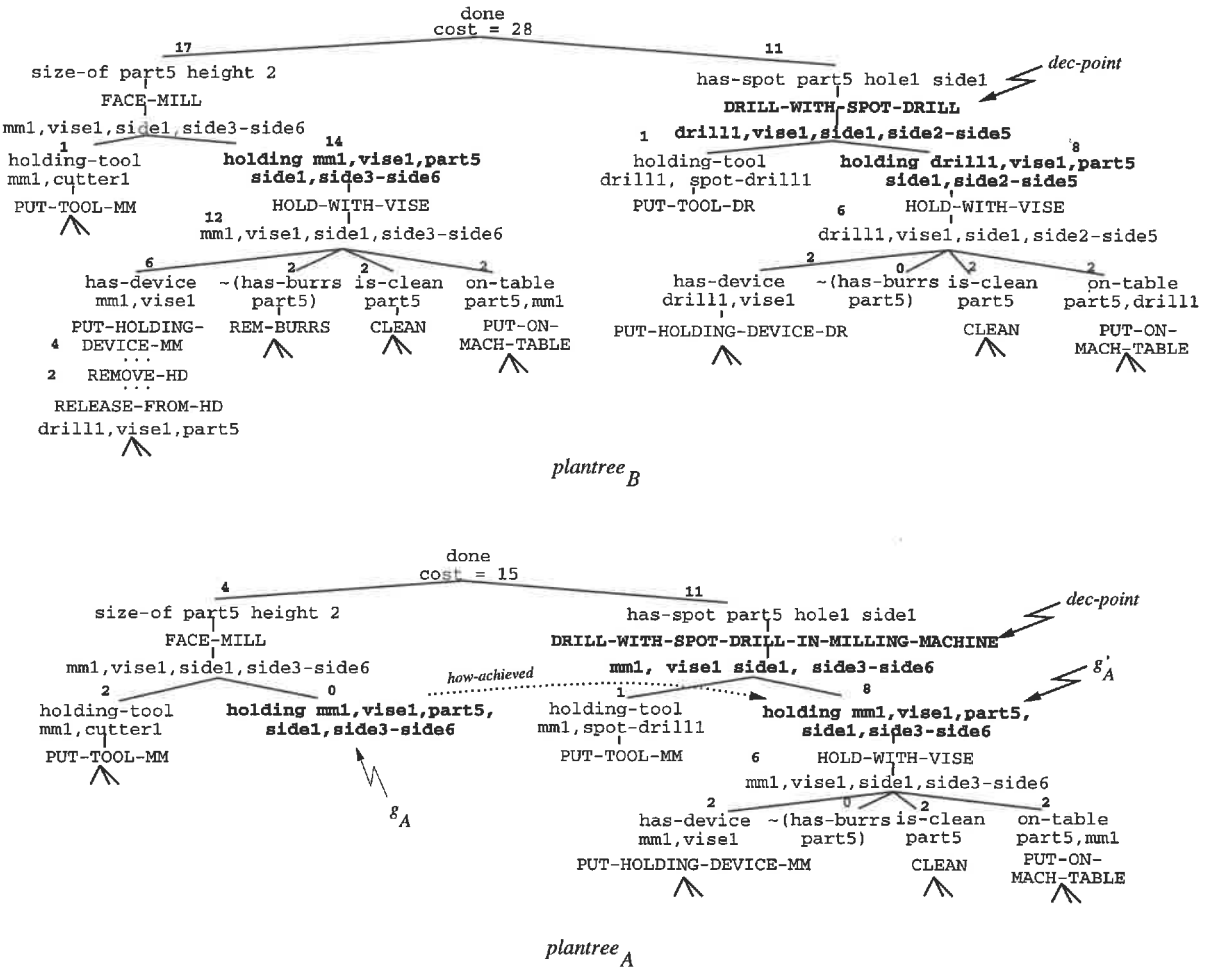


Figure 3.25: Plan trees obtained from the problem solving traces for plans (a) and (b) of Figure 2.6. The top plan tree corresponds to the worse quality solution, $plantree_B$ in the learning algorithms. The bottom plan tree corresponds to the better plan, and therefore it is $plantree_A$. Some parts of the plan trees have been omitted for clarity and space purposes.

drill-with-spot-drill-in-milling-machine because it corresponds to the decision point. Note that more steps may be needed in general, such as if `has-spot` were a subgoal of drilling a hole (of which drilling a spot hole is the first step) and the learning opportunity were to learn the appropriate operator and binding choice for drilling the hole. Back to our example, when the propagation terminates:

- the relevant bindings at that point are those of the part, machine, holding device, side,

and side pair. The tool `spot-drill1` was not propagated because it was not relevant for `holding`. Therefore is not marked as relevant.

- **prepare_goal_precond** makes operational the fact that g_A was a 0 cost subgoal: `(holding mm1 vise1 part5 side1 side3-side6)` was a pending goal at the decision point (Steps 3-4 of Figure 3.17).
- **static_precs** computes the constraints on the type and value of the relevant bindings. The type specification of the operator variables requires that the type of the machine is a milling machine. This type is a subset of the types allowed by the signature of the `holding` predicate, in which any machine is valid. Steps 7-10 of Figure 3.18 build this constraint on the type of the machine. The other relevant bindings do not originate other constraints.

Step 5 of **propagate_conditions_up** stores all this information in the structure shown in Figure 3.26. From this structure two control rules are created because the explanation supports the choice of both the operator and the bindings at the decision point.

- An operator preference control rule, that would override the default operator choice at node `n11`.
- A bindings preference control rule. The bindings for the part and side are given from the right-hand side of the operator when it is matched against the goal. Therefore they need not be specified in the bindings control rule. Given the operator choice the type of the machine will necessarily be a milling machine. Therefore the constraint on the machine type need not appear in the bindings rule.

:current_goal	has-spot part5 hole1 side1 1.375 0.25
:current_op	drill-with-spot-drill-in-milling-machine
:good	<drill-with-spot-drill-in-milling-machine milling-machine1 spot-drill1 vise1 part5 side1 side3-side6 1.375 0.25 hole1>
:bad	<drill-with-spot-drill drill1 spot-drill1 vise1 part5 side1 side2-side5 hole1 1.375 0.25>
:relevant_bnds	((<machine> milling-machine1) (<holding-device> vise1) (<part> part5) (<side> side1) (<side-pair> side3-side6))
:goal_precond	<pending-goal (holding milling-machine1 vise1 part5 side1 side3-side6)>
:other_preconds	(type-of-object <machine> milling-machine)
:decision_point	#<goal-node 11 #<has-spot part5 hole1 side1 1.375 0.25>>
:rule_type	:operator

Figure 3.26: Structure built by **create_rule_struct** (Figure 3.19) at the end of the propagation process.

Figure 3.27 shows the two control rules learned from this example. The rules indicate which operator and bindings are preferred to achieve the current goal, namely to make a spot hole in a certain part side, if a pending goal (that is, a goal yet to be achieved by the planner) is to hold that part in a milling machine in certain orientation and with certain holding device.

```
(control-rule pref-drill-with-spot-drill-in-milling-machine30
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (pending-goal (holding <mach> <holding-dev> <part> <side> <side-pair>))
          (type-of-object <mach> milling-machine)))
 (then prefer operator drill-with-spot-drill-in-milling-machine
                       drill-with-spot-drill))

(control-rule pref-bnds-drill-with-spot-drill-in-milling-machine31
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (current-operator drill-with-spot-drill-in-milling-machine)
          (pending-goal (holding <mach4> <holding-dev5> <part> <side> <side-pair-6>))
          (or (diff <mach4> <mach1>) (diff <holding-dev5> <holding-dev2>)
              (diff <side-pair-6> <side-pair-3>))))
 (then prefer bindings  ((<mach> . <mach4>)(<hd> . <holding-dev5>)(<sp> . <side-pair-6>))
                        ((<mach> . <mach1>)(<hd> . <holding-dev2>)(<sp> . <side-pair-3>))))
```

Figure 3.27: Operator and bindings preference control rules learned from the problem in Figure 2.5.

3.9 Learning Goal Preference Control Rules

The previous two sections described how operator and bindings rules are learned and illustrated it with an example. This section focuses on learning goal preference rules. Section 3.9.1 describes how goal decisions influence plan quality by introducing a small artificial domain. Section 3.9.2 presents how quality-enhancing goal rules are learned. Then an example in the process planning domain is used in Section 3.10 to step through the learning process.

3.9.1 When Are Goal Preferences Needed? An Example

Section 2.2 showed the influence of goal decisions in the quality of a plan, thus motivating the need of quality-enhancing goal preference search-control rules. To explain this point we introduce in this section a small artificial domain. Figure 3.28 shows the domain and a plan quality metric. Figure 3.29 describes a simple problem. The goal is to achieve both g_1 and g_2. This example will illustrate how two plans of the same length but different quality are obtained depending on a particular goal decision.

Given the initial state, in the process of achieving g_1 PRODIGY will necessarily delete g_{21}. g_{21} is needed to achieve g_2. On the other hand, in the process of achieving g_2 PRODIGY deletes g_{12}, which is needed to achieve g_1. Therefore either order of achieving g_1 and g_2 will cause a

```
operator op1
:preconds (and (g11)(g12))
:adds     (g1)

operator op11
:preconds (g21)
:adds     (g11)
:dels     (g21)

operator op12
:preconds (and (g121)(g122))
:adds     (g12)

operator op121
:preconds (p)
:adds     (g121)
```

```
operator op2
:preconds (and (g21)(g22))
:adds     (g2)

operator op21
:preconds (g211)
:adds     (g21)

operator op22
:preconds (and (g221)(g12))
:adds     (g22)
:dels     (g12)
:conditional-effects
       (if (p) (del (g121))
```

Quality metric:	op1	op2	op11	op21	op12	op22	op121
	2	2	1	1	6	6	6

Figure 3.28: An artificial domain used to illustrate the need of quality-enhancing goal preference control rules. The quality metric assigns a cost to each operator. The cost of a plan is the sum of the cost of each of the plan operators. Higher values mean higher cost and worse quality.

goal clobbering. Figure 3.30 presents two solutions to that problem, corresponding to the two orderings of achieving g_1 and g_2. Above each solution is its corresponding problem solving trace. [14] Initially the planner obtains the trace and solution on the left. On the bottom right-hand side of the figure is the improved solution. From it, the problem solving trace (shown above it) is obtained, as described in Section 3.4. At `n14` the interrupt mechanism detects that the application of op_{22} as the first operator in the plan is not consistent with the partial order, in Figure 3.31, for the desired solution. The interrupt signal forces the planner to backtrack to `n10` and try a different goal ordering. Note that an alternative backtracking point would have been `n13`, preferring subgoaling on g_{11} to applying op_{22}. However the heuristics we have developed to choose a backtracking point prefer the first option, as it corresponds to an earlier decision faced by the planner. `n14` becomes a decision point open to the need of new control knowledge. Two points are important in this example:

- PRODIGY4.0 has different domain independent heuristics that detect goal interactions similar to this one. However they are not enough to guide the planner to the better plan in this example, since both plans have a similar goal interaction, and their difference in

[14] The delete goal control rule that fires at node n7 is similar to that described at the end of Section 2.3.

```
State: (and (g12)  (g21)        Goal:  (and (g1)(g2))
            (g121) (g211)
            (g122) (g221))
```

Figure 3.29: Example problem in the domain of Figure 3.28.

```
  2 n2 (done)
  4 n4 <*finish*>
  5   n5 (g1) [1]
  7   n7 <op1>
Firing delete goals rule
  8   n8 (g2)
 10   n10 <op2>
 11     n11 (g22) [1]
 13     n13 <op22>
 14     n14 <OP22>
 15   n15 <OP2>
 16     n16 (g11) [1]
 18     n18 <op11>
 19     n19 <OP11>
 20     n20 (g12)
 22     n22 <op12>
 23     n23 <OP12>
 23   n24 <OP1>
 23 n25 <*FINISH*>
```

```
Plan1:  1. <op22>
        2. <op2>
        3. <op11>
        4. <op12>
        5. <op1>

        cost = 17
```

(a)

```
  2 n2 (done)
  4 n4 <*finish*>
  5   n5 (g1) [1]
  7   n7 <op1>
Firing delete goals rule
  8   n8 (g2)
 10   n10 <op2>
 11     n11 (g22) [1]
 13     n13 <op22>
 14     n14 <OP22>
backtracking to obtain better solution:
  prefer other goal after n10
 10   n10 <op2>
 11     n16 (g11)
 13     n18 <op11>
 14     n19 <OP11>
 15   n20 <OP1>
 16     n21 (g21) [1]
 18     n23 <op21>
 19     n24 <OP21>
 20     n25 (g22)
 22     n27 <op22>
 23     n28 <OP22>
 23   n29 <OP2>
 23 n30 <*FINISH*>
```

```
Plan2:  1. <op11>
        2. <op1>
        3. <op21>
        4. <op22>
        5. <op2>

        cost = 12
```

(b)

Figure 3.30: Two solutions for the problem in Figure 3.29 with the corresponding traces. (a) was obtained initially. (b) was constructed from the improved plan (as described in Section 3.4), shown at the bottom. `n14` is the decision point where new control knowledge is required and therefore became a learning opportunity. Although both plans have the same length, they do not have the same cost, since `op12` is more expensive than `op21`.

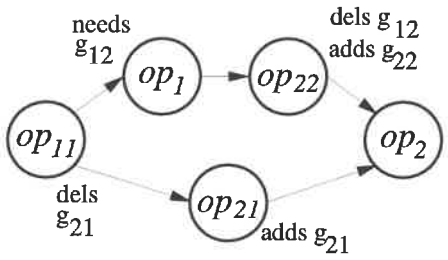


Figure 3.31: Partial order corresponding to the better quality solution of Figure 3.30. The partial order is used to generate a problem solving trace from the solution.

quality is captured only by the quality metric.

- Both solutions have the same length, and they only differ in op_{12} and op_{21}. This difference is due to the order in which the top level goals are achieved. This ordering causes different subgoals (g_{12} and g_{21}) to become subgoals, therefore introducing operators op_{12} and op_{21} respectively in the plan. The costs of op_{12} and op_{21} are different, hence the costs of the complete plans are different.

Figure 3.32 shows the plan trees built from the two solutions. The plan trees store information about how goals were achieved. For example in the first solution g_{12} was achieved by subgoaling, while in the second solution it was true in the state when needed. The plan trees also store information about deleted subgoals. For example, op_{22} deleted g_{12} in both plans.

The learner generates automatically the rule in Figure 3.33. The rational behind the rule is that when g_{22} is a candidate subgoal, and g_{12} is true in the state and is needed by some operators, PRODIGY should work towards applying those operators first, because achieving g_{22} will delete g_{12}. When PRODIGY attempts to match the rule at the decision point, `<other-goal>` gets bound to the first candidate goal. The control rule matcher works by looking for another candidate, `<pref-goal>` that is preferred over the currently first candidate `<other-goal>`. The control rule utilizes the following domain-independent meta-predicates:

- `(candidate-goal <g>)`: true if `<g>` is among the set of goals PRODIGY4.0 may choose to work on. If `<g>` is unbound, `candidate-goal` may generate values for it.
- `(known <expr>)`: tests whether `<expr>` is true in the state, and can be used as a generator of values for the variables that appear in `<expr>`.
- `(is-subgoal-of-ops <goal> <instantiated-ops>)`: tests whether `<goal>` is a precondition of one of `<instantiated-ops>`. It can be used as generator for both `<goal>` and `<instantiated-ops>`, binding them respectively to a literal and a list of instantiated operators. If `<goal>` is a list of goals, `<ops>` is the union of the corresponding instantiated operators.

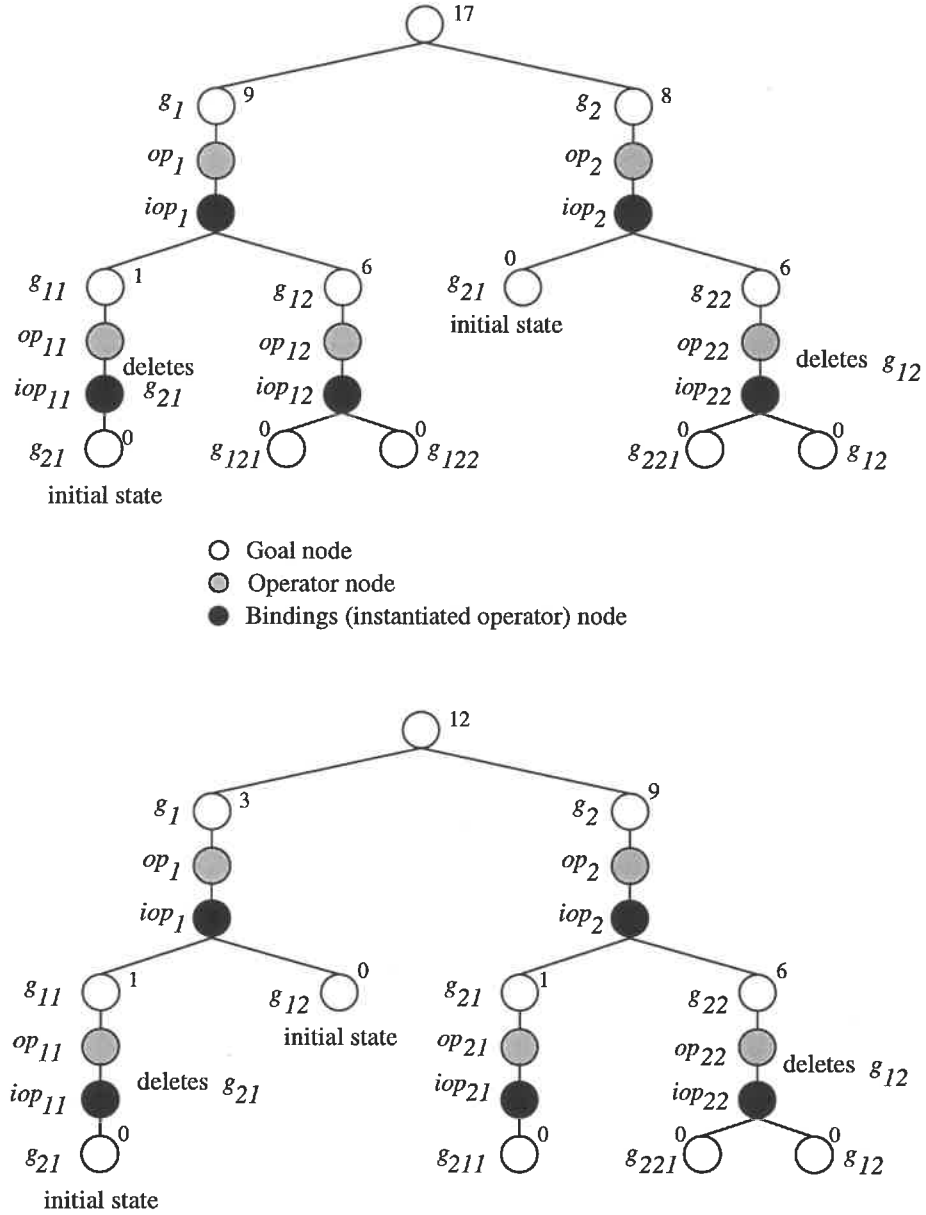


Figure 3.32: Plan trees corresponding to the two solutions in Figure 3.30.

- `(is-pending-subgoal-in-subtree <goal> <instantiated-ops>)`: tests whether `<goal>` (a) is currently a pending goal, and (b) is a subgoal in the subgoal tree below one of `<instantiated-ops>`. If `<goal>` is unbound, it can be used as a generator of bindings for `<goal>`.

- `(first-pending-subgoal-in-subtree <goal> <instantiated-ops>)`: tests whether

```
(control-rule prefer-goal-1
 (if (and (candidate-goal (g22))
          (known (g12))
          (is-subgoal-of-ops (g12) <ops>)
          (first-pending-subgoal-in-subtree <pref-goal> <ops>)
          (diff <pref-goal> <other-goal>)
          (~ (is-pending-subgoal-in-subtree <other-goal> <ops>))))
 (then prefer goal <pref-goal> <other-goal>))
```

Figure 3.33: Control rule that makes the correct goal ordering decision in the problem of Figure 3.29. The rule suggests PRODIGY to work on the operators that need g_{12} while it is true, instead of working on g_{22}, which would delete g_{12}. This goal preference control rule was automatically learned from the plan trees in Figure 3.32.

`<goal>` is the first of the pending goals at the node that belongs to the subgoal tree below one of `<instantiated-ops>`. If `<goal>` is unbound, it binds it to such subgoal.

The rule in Figure 3.33 was learned because of the given quality metric used, in particular because of the difference in cost of op_{12} and op_{21}. Should their cost have been reversed, a different rule would have been learned.[15]

3.9.2 How Goal Preference Rules Are Learned

In previous sections we saw how after the learning opportunities have been found, the learner selects a particular learning mechanism to explore each learning opportunity based on the type of decision associated with it (Steps 7-12 of Figure 3.10). Section 3.7 described how operator and bindings control rules are learned. This section describes the mechanism to learn goal preference control rules.

The top level function of this mechanism is **learn_goal_dec** (Figure 3.34). It is called for each learning opportunity <g_A, g_B, dec_point> corresponding to a wrong goal decision. Recall that g_A and g_B are nodes in $plantree_A$ and $plantree_B$ respectively corresponding to the same goal g. g_A has cost 0 because it was true when it was needed as a precondition. g_B has cost greater than 0 because it was deleted and needed to be reachieved. For example, in the plantrees of Figure 3.32 g_A and g_B correspond to the ocurrences of g_{12} as precondition of op_1 in the two plan trees. The decision point is node `n10` in the traces of Figure 3.30.

The goal of the algorithm is to find out how to protect g_B from deletion before it is used as a precondition. The solution is reordering at some point the candidate subgoals, so the goals that need g_B are achieved before the goals that cause its deletion. The candidate point for such

[15] Actually no rule would have been learned in this example because the planner would have initially obtained the better solution. The rule would have been learned if the planner's default choice had been op_{21} instead.

learn_goal_dec(g_A, g_B, dec_point)

1. $op_{deleting} \leftarrow$ **op_that_deleted**(g_B)
2. $g_{deleting} \leftarrow$ **who_needs_goal**(**parent_goal**($op_{deleting}$), *dec_point*)
3. $G_{needing} \leftarrow$ **who_needs_goal**(g_A, *dec_point*)
4. $\bigcup$ {**who_needs_goal**(g, *dec_point*): $g \in$ **goal_achieves_too**(g_A)}
5. $rel\_bnds_{G_{needing}} \leftarrow$
6. **merge_rel_bnds**({**propagate_conditions_up** (g,**all_bnds**(g),*dec_point*,g,-,$\emptyset$)
7. : $g \in G_{needing}$})
8. $rel\_bnds_{g_{deleting}} \leftarrow$
9. **merge_rel_bnds**(**propagate_conditions_up** ($g_{deleting}$,**all_bnds**(g),*dec_point*,$g_{deleting}$,-,$\emptyset$))
10. *constraints* $\leftarrow$ **extract_constraints** ($rel\_bnds_{G_{needing}}, rel\_bnds_{g_{deleting}}$)
11. *bnds* $\leftarrow$ **common_bnds** ($rel\_bnds_{G_{needing}}, rel\_bnds_{g_{deleting}}$)
12. $precond_{g_A} \leftarrow$ **generalize**($g_A, bnds$)
13. $precond_{G_{needing}} \leftarrow$ **generalize**($G_{needing}, bnds$)
14. $precond_{g_{deleting}} \leftarrow$ **generalize**($g_{deleting}, bnds$)
15. *gen_constraints* $\leftarrow$ **generalize**(*constraints, bnds*)
16. *rule_struct* $\leftarrow$;; Figure 3.36
17. **create_goal_prefer_rule**($precond_{g_A}, precond_{G_{needing}}, precond_{g_{deleting}}$, *gen_constraints*)
18. return(*rule_struct*)

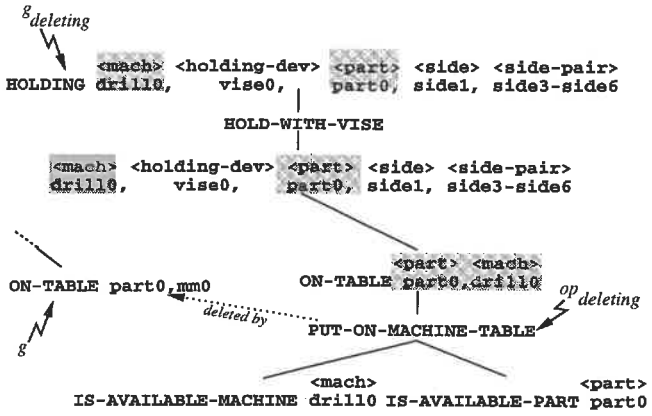
Figure 3.34: Algorithm for learning goal preference control rules.

reordering is *dec_point*, as the better quality solution was achieved by overriding PRODIGY's choice at that node. **Learn_goal_dec** starts by finding out which operator $op_{deleting}$ deleted g_B in $plantree_B$. Step 2 finds $g_{deleting}$, a subgoal candidate at the decision point for which operator $op_{deleting}$ was selected. Thus achieving $g_{deleting}$ caused g_B's deletion. Step 3 computes $G_{needing}$, the set of candidate subgoals at the decision point whose achievement needs g_A. This set also includes subgoals whose achievement needs other instances of g_A at different nodes of the plantree; those other nodes are stored in the `:achieves-too` slot of node g_A and are returned by the call to **goal_achieves_too**. To obtain the better quality plan the planner should work on $G_{needing}$ before working on $g_{deleting}$. Both $g_{deleting}$ and $G_{needing}$ are computed using **who_needs_goal**(g, dec_point), which returns g if g is a candidate at dec_point. Otherwise it returns the goal candidate at the dec_point that lead eventually to subgoaling on g. In other words, it operationalizes why g is needed, with respect to the meta-knowledge available at the decision point. It works similarly to **operationalize** in Figure 3.17.

In the artificial domain (Figure 3.32), $op_{deleting}$ is op_{22}, which deleted g_{12}. Its parent, g_{22}, is $g_{deleting}$, a candidate goal at the decision point which caused g_{12} to be deleted. g_{12} itself is a candidate as well at the decision point, and becomes the only element in $G_{needing}$.

```
(Operator PUT-ON-MACHINE-TABLE
 (params <machine> <part>)
 (preconds
  ((<machine> Machine)
   (<part> Part))
  (and
   (is-available-part <part>)
   (is-available-machine <machine>)))
 (effects
  ((<other-machine> Machine))
  ((del (on-table <other-machine> <part>))
   (add (on-table <machine> <part>)))))
```

(a)



(b)

Figure 3.35: (a) An operator that adds and deletes two instantiations of the same predicate. (b) Partial plan tree where the operator is used to achieve `(on-table part0 drill0)`. A constraint is built saying that `<mach>` must be different from `mm0`. The constraint is needed so that the operator actually adds and deletes the two different instantiations of `on-table`.

Next the algorithm computes in Steps 5-9 the relevant bindings of $G_{needing}$ and $g_{deleting}$ so that g is needed by $G_{needing}$ and deleted by $g_{deleting}$. This is done in a similar way as for the operator and bindings rules in Figure 3.14. The relevant binding propagation may find constraints on the values of the relevant bindings, and these constraints will be part of the control rule precondition.[16] Figure 3.35 presents an example of how one of these constraints is found. On the left-hand side there is an operator (to move one part between two machines) that both adds and deletes two instantiations of the same predicate. That operator was chosen on the right-hand side of the figure to achieve the goal `(on-table drill0 part0)`. Its application deleted g = `(on-table mm0 part0)` as a side effect. When g is analyzed by **learn_goal_dec** the instantiation of operator put-on-machine-table becomes $op_{deleting}$. When the relevant bindings `<machine>` and `<part>` are propagated up, a constraint is added saying that `<machine>` must be different from `mm0` in order for the operator to delete g.

The relevant bindings found are used in Steps 12-14 to generalize $G_{needing}$ and $g_{deleting}$. Their generalization will be used to build the control rule precondition. **Generalize** generates consistent variable names throughout all the preconditions based on the relevant bindings, makes sure that the variables will be properly bound at control-rule matching time, and may generate constraints on the types of the variables. The types are those specified in the operators that introduced the subgoals in $G_{needing}$ and $g_{deleting}$ in a which the variable appears. Finally, a control rule is created by filling in the template in Figure 3.36.

[16] We have separated the computation of the constraints as Step 10 for clarity purposes. However they are actually computed as the propagation occurs.

create_goal_prefer_rule(g_A, $g_{deleting}$, $G_{needing}$, *constraints*)

```
(control-rule prefer-goal-n
  (if (and (candidate-goal g_deleting)
           if g_A ∈ G_needing then (known g_A)          ;; g_A needs to be protected
           {(pending-goal g) such that g ∈ G_needing \ g_A}
           (is-subgoal-of-ops G_needing <ops>)
           constraints
           (first-pending-subgoal-in-subtree <pref-goal> <ops>)
           (diff <pref-goal> <other-goal>)
           (~(is-pending-subgoal-in-subtree <other-goal> <ops>))))
  (then prefer goal <pref-goal> <other-goal>))
```

Figure 3.36: Template to create a goal preference control rule using the information gathered by **learn_goal_dec**. The template, as well as the learning algorithm, are independent of the domain. The rule will become domain-independent when the results of the learning algorithm are used to fill out the template. The meta-predicates used are described in Section 3.9.1.

3.10 Example Of Learning Goal Preference Control Rules

To illustrate how goal preference control rules are learned, we introduce here another problem from the process planning domain. Figure 3.37 shows the initial state and goal statement for this problem. The goal is to have a part of length 0.5 inches with a spot hole on its side 1 at coordinates .5 $\times$ 1.5. An aluminum part of length 5 inches is available in the shop. Given the large size reduction needed, the part must be face-milled. The part is already clean, and it is being held on the milling machine with its side 3 facing up. This orientation is appropriate for face milling the length of the part. The milling machine spindle is holding a spot drill, and several milling tools are available in the shop. Given this initial set-up roughly two plans to machine the part are possible:

- Start by drilling the spot hole. To do this, maintain the drilling tool in the machine but release the part to put side 1 facing up. After drilling the hole, hold the part in the initial orientation (side 3 facing up), replace the tool, and face mill the part.
- Start by face milling the part. To do this, maintain the part orientation and replace the drilling tool with a milling tool. After milling the part, change the part's orientation (side 1 up), put the drilling tool back, and drill the spot hole.

Figure 3.38 shows the complete plans corresponding to these alternatives. Both plans have the same length, but plan (a) has a greater cost according to the quality metric of Table 2.1. The difference in quality is due to the first two operators of each plan, since operators that move

```
(objects
;;machines
 (object-is milling-mach1 MILLING-MACHINE)
 (object-is drill1 DRILL)

;;holding devices
 (object-is vise1 VISE)

;;parts and holes
 (object-is part5 PART)
 (object-is hole1 HOLE)

;;tools
 (object-is spot-drill1 SPOT-DRILL)
 (object-is twist-drill13 TWIST-DRILL)
 (object-is high-helix-drill1 HIGH-HELIX-DRILL)
 (object-is tap14 TAP)
 (object-is counterbore2 COUNTERBORE)
 (object-is tap1 TAP)
 (object-is plain-mill1 PLAIN-MILL)
 (object-is end-mill1 END-MILL)

 (object-is brush1 BRUSH)
 (object-is soluble-oil SOLUBLE-OIL)
 (object-is mineral-oil MINERAL-OIL))


        (state (and (diameter-of-drill-bit twist-drill13 1/4)
                    (diameter-of-drill-bit high-helix-drill1 1/32)
                    (diameter-of-drill-bit tap14 1/4)
                    (size-of-drill-bit counterbore2 1/2)
                    (diameter-of-drill-bit tap1 1/32)

                    (material-of part5 ALUMINUM)
                    (size-of part5 LENGTH 5)
                    (size-of part5 WIDTH 3)
                    (size-of part5 HEIGHT 3)

                    (has-device milling-mach1 vise1)
                    (holding milling-mach1 vise1 part5 side3 side2-side5)
                    (holding-tool milling-mach1 spot-drill1)
                    (is-clean part5)))

    (goal ((<part> PART)) (and (size-of <part> LENGTH 0.5)
                               (has-spot <part> hole1 side1 1/2 1.5)))
```

Figure 3.37: Example problem in the process planning domain to illustrate the learning of goal control rules. The goal is to have a part of length 0.5 with a spot hole on side1. The part is initially set on the milling machine in the orientation required for face-milling its length. The spot-drill is initially ready in the machine tool holder.

tools are cheaper than operators that set up parts and holding devices. Plan (a) was obtained in PRODIGY's first attempt to solve this problem using its default heuristics.[17] Plan (b) was the result of the improvements on plan (a) suggested through the interaction with the human expert. Figures 3.39 and 3.40 show the beginning of the problem solving process to obtain respectively plans (a) and (b). `n13` is the first decision point where PRODIGY's default decision was overriden in order to obtain the improved plan.

Given these two solutions, the learner first builds the plan trees in Figure 3.41. The top plan tree $plantree_B$ corresponds to the worse quality plan. The bottom one $plantree_A$ corresponds to the better plan. The learner then looks for learning opportunities. The cost of holding the

[17] PRODIGY had already learned control rules to prefer the milling machine for the drilling operation.

A plan	A better plan
release milling-mach1 vise1 part5 side3 side2-side5	**remove-tool** milling-mach1 spot-drill1
hold milling-mach1 vise1 part5 side1 side2-side5	**put-tool-mm** milling-mach1 plain-mill1
drill-with-spot-drill-mm milling-mach1 spot-drill1 vise1 part5 hole1 side1 side2-side5 1/2 1.5	**face-mill** milling-mach1 part5 plain-mill1 vise1 side3 side2-side5 length 5 0.5
remove-tool milling-mach1 spot-drill1	**remove-tool** milling-mach1 plain-mill1
put-tool-mm milling-mach1 plain-mill1	**put-tool-mm** milling-mach1 spot-drill1
release milling-mach1 vise1 part5 side1 side2-side5	**release** milling-mach1 vise1 part5 side3 side2-side5
remove-burrs part5 brush1	**remove-burrs** part5 brush1
clean part5	**clean** part5
hold milling-mach1 vise1 part5 side3 side2-side5	**hold** milling-mach1 vise1 part5 side1 side2-side5
face-mill milling-mach1 part5 plain-mill1 vise1 side3 side2-side5 length 5 0.5	**drill-with-spot-drill-mm** milling-mach1 spot-drill1 vise1 part5 hole1 side1 side2-side5 1/2 1.5
cost = 18	**cost = 16**
(a)	(b)

Figure 3.38: (a) Plan obtained by PRODIGY guided by the current control knowledge. (b) A better plan, according to the quality metric, input by a human expert. Note that both plans have the same length but different quality.

```
  2 n2 (done)
  4 n4 <*finish* part5>
  5   n5 (size-of part5 length 0.5) [1]
  7   n7 <face-mill
         milling-machine1 part5 plain-mill1 vise1 side3 side2-side5 length 5 0.5> [7]
Firing delete goals EXPAND-MAIN-GOALS-FIRST
  8     n8 (shape-of part5 rectangular) [1]
 10     n10 <is-rectangular part5>
Firing delete goals EXPAND-MAIN-GOALS-FIRST
 11   n11 (has-spot part5 hole1 side1 1/2 1.5)
Firing operator pref rule
 13   n13 <drill-with-spot-drill-in-milling-machine
          milling-machine1 spot-drill1 vise1 part5 hole1 side1 side2-side5 1/2 1.5> [1]
 14     n14 (holding milling-machine1 vise1 part5 side1 side2-side5) [1]
 16     n16 <hold-with-vise milling-machine1 vise1 part5 side1 side2-side5>
 17       n17 (on-table milling-machine1 part5) [3]
 ...
```

Figure 3.39: Beginning of the problem solving trace that obtained plan (a) of Figure 3.38.

part for the milling operation is 0 in $plantree_A$ and 8 in $plantree_B$. The nodes g_A and g_B corresponding to that goal and the relevant decision point n13 become the learning opportunity. **learn_goal_dec** (Figure 3.34) is called and computes:

```
  2 n2 (done)
  4 n4 <*finish* part5>
  5   n5 (size-of part5 length 0.5) [1]
  7   n7 <face-mill
        milling-machine1 part5 plain-mill1 vise1 side3 side2-side5 length 5 0.5> [7]
Firing delete goals EXPAND-MAIN-GOALS-FIRST
  8     n8 (shape-of part5 rectangular) [1]
 10     n10 <is-rectangular part5>
Firing delete goals EXPAND-MAIN-GOALS-FIRST
 11   n11 (has-spot part5 hole1 side1 1/2 1.5)
Firing operator pref rule
 13   n13 <drill-with-spot-drill-in-milling-machine
         milling-machine1 spot-drill1 vise1 part5 hole1 side1 side2-side5 1/2 1.5> [1]
 14     n14 (holding milling-machine1 vise1 part5 side1 side2-side5) [1]
 16     n16 <hold-with-vise milling-machine1 vise1 part5 side1 side2-side5>
 17       n17 (on-table milling-machine1 part5) [3]
Op #<OP: PUT-ON-MACHINE-TABLE> was not in the solution proposed by the expert.
Backtracking to make new operator decision at node 18.

19       n21 <release-from-holding-device milling-machine1 vise1 part5 side3 side2-side5> [20]
 20       n22 <RELEASE-FROM-HOLDING-DEVICE MILLING-MACHINE1 VISE1 PART5 SIDE3 SIDE2-SIDE5>
Wrong op application sequence:  there are other ops that should have been applied
before op #<RELEASE-FROM-HOLDING-DEVICE [<MACHINE> MILLING-MACHINE1]...  (at node 22).
Backtracking to make new decision:  preferring other goal or applied op at node 16.

 16     n16 <hold-with-vise milling-machine1 vise1 part5 side1 side2-side5>
 17       n24 (is-empty-holding-device vise1 milling-machine1) [2]
...
Backtracking to make new decision:  preferring other goal or applied op at node 13.

 13   n13 <drill-with-spot-drill-in-milling-machine
         milling-machine1 spot-drill1 vise1 part5 hole1 side1 side2-side5 1/2 1.5> [1]
 14     n47 (holding-tool milling-machine1 plain-mill1)
 16     n49 <put-tool-on-milling-machine milling-machine1 plain-mill1>
...
```

Figure 3.40: Beginning of the problem solving trace to obtain plan (b) of Figure 3.38. At `n14` PRODIGY tries first to work on holding the part for the drill operation but realizes that the expert's solution will not be obtained with that ordering. Eventually PRODIGY backtracks to `n13` and tries working on switching the tool at `n47`.

- $op_{deleting}$: `<release milling-machine1 vise1 part5 side3 side2-side5>`, the operator that deleted g_B (Step 1).
- $g_{deleting}$: `(holding milling-machine1 vise1 part5 side1 side2-side5)`, the candidate goal at the decision point which lead to the expansion of $op_{deleting}$ (Step 2).
- $G_{needing}$: {`(holding milling-machine1 vise1 part5 side3 side2-side5)`}, the set of candidate goals at the decision point whose achievement needs g_A. In this example the set has only one element, g_A itself (Step 3).
- $rel\_bnds_{G_{needing}}$: all the bindings in g_A, namely the machine, holding device, part, side up and side pair, are relevant since g_A is in $G_{needing}$ (Steps 5-7).

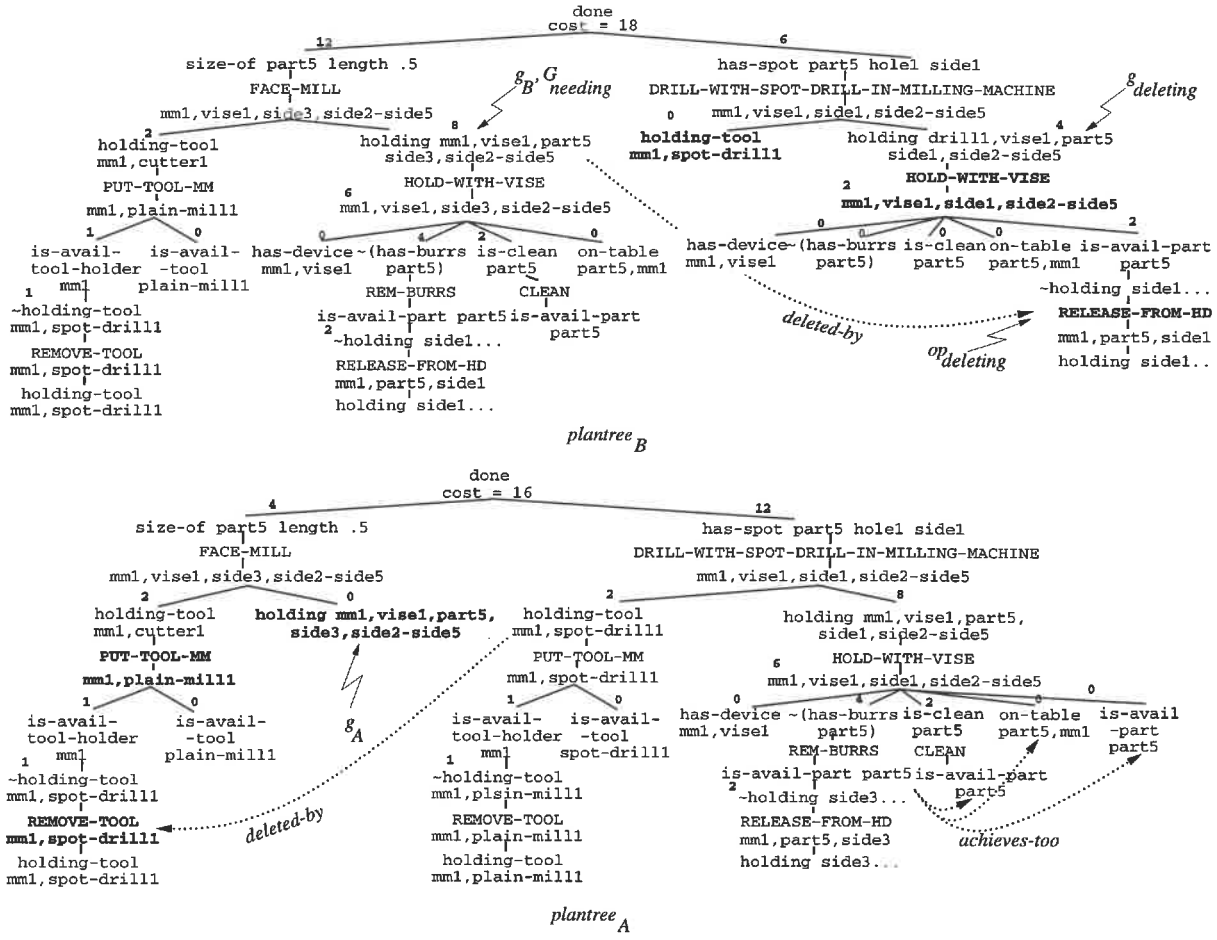


Figure 3.41: Plan trees obtained from the problem solving traces for plans (a) and (b) of Figure 3.38. The top plan tree $plantree_B$ corresponds to the worse quality plan, plan (a). The bottom one $plantree_A$ corresponds to the better plan, plan (b).

- $rel\_bnds_{g_{deleting}}$: in order that achieving $g_{deleting}$ deletes g_B, the only relevant binding is the binding for the part. This is computed by propagating the deleting effect of $op_{deleting}$ up to $g_{deleting}$ in $plantree_B$ (top of Figure 3.41). The propagation goes through goal node `(is-available-part part5)`, which reduces the relevant bindings to the binding for the part (Steps 8-9).

This information is used to fill in the template of Figure 3.36 and build the rule in Figure 3.42. The rule suggests to keep working on the subgoals of the operators that will need the `known` precondition before working on the candidate goal that matches the first precondition.

```
(control-rule prefer-goal-1
 (if (and (candidate-goal
            (holding <milling-machine15> <vise16> <part> <side17> <side2-side58>))
          (known (holding <machine-1> <holding-device-2> <part> <side-3> <side-pair-4>))
          (is-subgoal-of-ops
            (holding <machine-1> <holding-device-2> <part> <side-3> <side-pair-4>)
            <ops>)
          (first-pending-subgoal-in-subtree <pref-goal> <ops>)
          (diff <pref-goal> <other-goal>)
          (~ (is-pending-subgoal-in-subtree <other-goal> <ops>))))
 (then prefer goal <pref-goal> <other-goal>))
```

Figure 3.42: Goal preference control rule learned from the problem in Figure 3.37. The rule advises the planner to focus on the goals that need the holding set-up currently available (known precondition) instead of changing it (a `candidate-goal`).

3.11 Learning Control Rule Priorities

The algorithms presented in previous sections learn control rules from single problem solving episodes. In this section we discuss how those rules may be over general and therefore lead to incorrect decisions, and how our learner deals with this problem in a domain-independent way. Both points are illustrated with examples.

3.11.1 The Problem: Over-General Rules And Conflicting Preferences

QUALITY builds an explanation from a single example of a difference between a plan generated and a quality-improving modification of that plan as suggested by an expert. It compiles the reasons why one alternative at the decision point should be preferred over the others. The explanation is *incomplete* as it does not consider all possible hypothetical scenarios. For example, back to the example of Section 3.8, assume that in addition to reducing the part's height and making a spot hole, a third goal is to have `hole1` counterbored. Counterboring can only be performed by the drill press. Therefore using the milling machine for the spot hole may not be the best alternative, as the part has to be set up also in the drill press. However the rules in Figure 3.27 would still fire choosing the milling machine. Those rules are too general. It would be computationally too expensive to consider all the possible scenarios (goal statements and initial states) to build the complete explanation of when and why the alternative suggested by the rule is going to lead to a better plan. We consider this as an instance of the *intractable theory problem* mentioned in the literature on explanation-based learning [Mitchell *et al.*, 1986]. The consequence of using incomplete explanations is learning over-general knowledge. Some research has addressed this problem in a number of ways including introducing exceptions to the rule applicability [Tadepalli, 1989], making the domain theory more tractable by introducing simplifying assumptions [Chien, 1989, Ellman, 1988], learning more specific rules and sorting

If there is a conflict between preferences:
1. Use learned rule-priority information
   (only if preferences come from different control rules).
2. Use other heuristics:
   a. Proximity heuristic.
   b. First candidate goal heuristic (position in candidate goal list).
3. Use PRODIGY4.0's default heuristics:
   a. Left-right order of preconditions.
   b. First operator found.
   c. First instantiation found.

Figure 3.43: Hierarchy of heuristics to decide among conflicting preferences.

them in a hierarchy [Iwamoto, 1994], or inductively and incrementally refining the learned rules [Borrajo and Veloso, 1994b].

In the case of QUALITY the rules learned may fire in a context where a different rule (e.g. a more specific one) may be more appropriate in terms of producing a better quality plan. In addition, several rules learned from different previous problems may fire at a decision point and give conflicting preferences. PRODIGY4.0 must break that conflict in favor of one of the alternatives. QUALITY refines the learned knowledge incrementally upon failures. A failure occurs when the alternative preferred by the learned rules leads to a plan that can be improved. The refinement process does not modify the applicability conditions of the learned rules. Instead it adds new rules if needed and sets priorities among rules. In the counterboring example, a new rule is learned and it is given priority over the previously-learned rule.

3.11.2 How To Break Preference Cycles

Several over-general rules may fire at a given decision point suggesting conflicting alternatives.[18] A cycle in the rule preferences occur when there is a chain of rule firings $r_1, r_2, ...r_n$ such that $r_i, i = 1, ...n-1$ prefers alt_i over alt_{i-1} and r_n prefers alt_n over alt_1. PRODIGY4.0 then must break that conflict and choose one of the alternatives. We have devised a collection of heuristics to decide among the conflicting preferences suggested by the prefer control rules. Figure 3.43 shows the heuristics used and the order in which they are applied. In the first place PRODIGY4.0 uses available information about priorities among the existing preference rules. These priorities are learned automatically in a way described in the next section. They capture characteristics of the plan quality metric. The examples in Sections 3.11.4 and 3.11.5 illustrate the use of this

[18] In PRODIGY4.0 these conflicts only occur when preference control rules fire. Select and reject control rules choices determine the set of candidates over which the prefer rules will fire and choose the best candidate. From now on we will refer only to prefer control rules. Appendix A describes how prefer control rules fire.

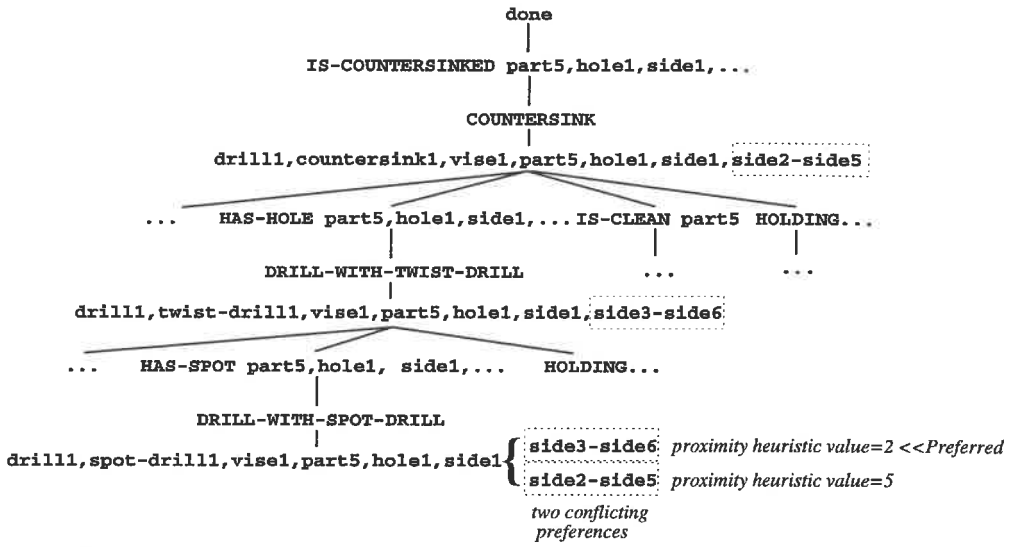
learned heuristic and its relationship with the plan quality metric. This heuristic is only useful to break conflicts among alternatives selected by different control rules. If the conflicting alternatives were the result of firing the same control rule with different instantiations, information about priorities among different rules is not useful to break the conflict.

If the conflict could not be broken using the learned control rule priorities, other heuristics are used. They prefer to keep the planner's focus of attention:

- The proximity heuristic: prefer operator and bindings alternatives that will share subgoals closer to the current focus of attention. Some of the operator and bindings preference control rules learned by the algorithms in Section 3.7 have a precondition of the form `(pending-goal <goal>)` (see Figure 3.27 for an example). If the rule fires and its suggested preference is chosen <*goal*>'s instantiation will be a subgoal whose achievement cost will be shared by several occurrences. Several of these rules, or one of them instantiated in several ways, may match at a decision point and give conflicting preferences. The proximity heuristic assigns a value to each of the rule instantiations that fire. Then it breaks the conflict by preferring the candidate selected by the rule instantiation with the lowest value. The value is computed as the length of the path (in number of PRODIGY4.0's parent-child node links) from the current node (at which the rule is firing) to the node that introduced <*goal*>. If <*goal*> and the current goal do not have a common ancestor, the value is the depth of PRODIGY4.0's search tree. This is the case for example when <*goal*> and the current goal are chosen to achieve different top-level goals.

  Figure 3.44 shows an example of the usefulness of the proximity heuristic. The figure shows PRODIGY4.0's search tree for a problem where the goal is to have a countersinked hole in the side 1 of a part. The sequence of steps to machine such a hole is to first make a spot hole, second to drill the hole, and last to countersink it. Before this last step the part needs to be cleaned, and therefore it must be released and then held again. Therefore for the plan to be good the orientation of the part for countersinking the hole needs not to be the same as the orientation for making the spot hole and hole. Assume that the planner is ready to choose an operator to drill the spot hole, and that the control rule in the figure, to choose bindings for the drill-with-spot-drill operator, has been learned from a previous episode. Its pending-goal precondition can be bound in two different ways, as there are two pending goals that match it. One is a precondition of countersink, with orientation `side2-side5`. The other is a precondition for drill-with-twist-drill, with orientation `side3-side6`. Therefore the control rule fires twice and suggests two different candidate orientations to make the spot hole. The figure shows the values assigned by the proximity heuristic to each candidate. The heuristic prefers the bindings such that the two drilling operators share their holding subgoals.

- The first candidate goal heuristic: if the conflict is about a goal decision, prefer the conflicting goal that appears earliest in the list of candidate goals. Although PRODIGY4.0




```
(control-rule prefer-bnds-drill-with-spot-drill7
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (current-operator drill-with-spot-drill)
          (pending-goal (holding <mach10> <holding-dev11> <part> <side> <side-pair-12>))
          (or (diff <mach10> <mach7>)
              (diff <holding-dev11> <holding-dev8>)
              (diff <side-pair-12> <side-pair-9>))))
 (then prefer bindings ((<mach> . <mach10>) (<hd> . <holding-dev11>) (<sp> . <side-pair-12>))
                       ((<mach> . <mach7>)  (<hd> . <holding-dev8>)  (<sp> . <side-pair-9>))))
```

Figure 3.44: Example to illustrate the use of the proximity heuristic. The bindings control rule in the bottom of the figure fired twice suggesting two different sets of bindings for drill-with-spot-drill (bottom of the tree). The conflicting alternative to which the heuristic assigns the lowest value is preferred.

can at any time work on any goal in the set of candidate goals, by default it keeps its focus of attention and works on the subgoals that have just been expanded. These subgoals appear first in the list of candidate goals. The first candidate goal heuristic chooses, among the conflicting alternatives, the one that appears earlier in the list of candidate goals. This heuristic is different from PRODIGY4.0's default behavior because it chooses only among the conflicting goals, instead of considering all the candidate goals. The rationale for the heuristic is to keep focused on the subgoals recently expanded as the planner has probably already invested some effort in achieving them that could be wasted if the focus changes to other subgoals.

Finally, if the above heuristics do not apply the conflict is broken using PRODIGY4.0's default heuristics, which among the conflicting alternatives choose:

- the goal that appears first using the left-right depth-first order of goal expansion,
- the operator that appears first in the domain definition, or
- the set of bindings that appears first in the order in which they were generated by the matcher.

3.11.3 Learning Control Rule Priorities

The previous section described how priorities between learned control rules can be used to overcome the problem of conflicting preferences. This section describes how those priorities are learned.

Priorities among rules are represented as pairs of control rules such that the first element of the pair has priority over the second one. Obviously the two elements of the pair are control rules of the same type. Otherwise the pair of rules would never fire simultaneously and therefore would never conflict. In the case of operator control rules they suggest relevant operators for the same goal. In the case of bindings control rules they suggest bindings for the same operator. The priorities determine a partial order among control rules of the same type. Our learning algorithm does not check whether adding a priority causes cycles in such partial order.

We have considered extending the priorities with conditions under the which the priority among the rules applies. That would amount to meta-control-rules. However the schema that we have developed does not include such conditions. Note that if those conditions were determined, they could be used instead to modify appropriately the control rules themselves, making them more specific to constraint their applicability and therefore avoid the conflicts.[19]

Previous sections described how the learner is always called upon failure, i.e. when the plan obtained with the currently available control knowledge can be improved. The construction of the search trace for the improved plan returns the set of decision points *ps_decisions* where the control knowledge failed to make the right choice. Let *bad_alternative* be the alternative that was wrongly suggested by the current control knowledge at $decision\_point \in ps\_decisions$. Let *good_alternative* be the alternative that was selected at the same *decision_point* in order to guide the planner towards the improved plan. There are two cases in which the learner tries to add a priority among conflicting rules:

- At *decision_point* both *good_alternative* and *bad_alternative* were suggested by a control rule. The conflict was broken in favor of *bad_alternative* (otherwise *decision_point* would not be in *ps_decisions*).

[19] Determining those conditions and how to modify the rules appropriately is not trivial. For example, if certain conditions α are used to decide between rules A and B, and certain conditions β are used to decide between rules A and C, pushing both α and β into A's preconditions might not work. See [Borrajo and Veloso, 1994a] for ways to address the rule refinement problem using inductive methods.

- At *decision_point bad_alternative* was suggested by a control rule, and the learner has learned a new rule that will fire preferring *good_alternative*.

---

analyze_preference_conflicts(*ps_decisions,rule_relative_prefs*)
rule_relative_prefs stores the learned preferences.

```
1.  unsolved_decisions_p ← nil
2.  for each decision ∈ ps_decisions
3.      if preference_conflict_p(decision)
4.      then
5.          good_alternative  ← decision.good(decision)
6.          bad_alternative   ← decision.bad(decision)
7.          node              ← decision.node(decision)
8.          rule_for_good_alt ← which_rule_preferred_alt(good_alternative,node)
9.          rule_for_bad_alt  ← which_rule_preferred_alt(bad_alternative,node)
10.         if <rule_for_bad_alt,rule_for_good_alt> ∈ rule_relative_prefs
11.         then                                   ;; The opposite priority exists.
12.             unsolved_decisions_p ← t           ;; Cannot learn. Continue
13.         else                                   ;; Learn priority among the control rules.
14.             rule_relative_prefs ← rule_relative_prefs
                                      ⋃ <rule_for_good_alt,rule_for_bad_alt>
15.             ps_decisions ← ps_decisions \ {decision}
16.     else
17.         unsolved_decisions_p ← t
18. if unsolved_decisions_p
19. then ...continue learning ...
```

Figure 3.45: Learning preferences among control rules. This function is called before learning new control rules and also after adding new rules that may cause conflicts with existing ones in the problem being solved.

Figure 3.45 describes how priorities among control rules may be learned in both of those cases: **analyze_preference_conflicts** is called both before any other learning is attempted, i.e. before the call to **learn** in Figure 3.10, and also when new control rules have been learned which may cause a conflict with the existing rules. The preferences learned are stored in *rule_relative_prefs*. In Step 3 **preference_conflict_p** determines whether there exist a preference conflict at the decision point.[20] In Step 8 *rule_for_good_alt* is the control rule that fired recommending *good_alternative*. Or it may be a rule just learned that would fire

[20]We have easily extended PRODIGY4.0 to record which control rules fired at each node, how they were instantiated, and which alternative(s) they preferred.

```
operator op (<x> <y>)
:preconds (and (g1 <x>)
               (g2 <y>))
:adds     (g <x> <y>)
```

```
operator op1 (<x>)
:preconds (p)
forall (<obj> object)
  :dels   (g1 <obj>)
:adds     (g1 <x>)
```

```
operator op2 (<y>)
:preconds (q)
forall (<obj> object)
  :dels   (g2 <obj>)
:adds     (g2 <y>)
```

```
(control-rule EXPAND-MAIN-GOALS-FIRST
  (if (and (candidate-goal <goal>)
           (goal-instance-of <goal> g)
           (candidate-goal <other-goal>)
           (~ (goal-instance-of <other-goal> g))))
  (then reject goal <other-goal>))
```

Quality metric:

op	op1	op2
1	3	6

Figure 3.46: An artificial domain used to illustrate conflicting preferences among the learned control rules and how learning helps to break them. Higher values of the quality metric mean higher cost and worse quality.

at that node preferring *good_alternative*. In Step 9 *rule_for_bad_alt* is the control rule that fired recommending *bad_alternative*. The priority of *rule_for_good_alt* over *rule_for_bad_alt* is recorded in the list of rule priorities *rule_relative_prefs* in Step 14. If *rule_for_bad_alt* already had priority over *rule_for_good_alt*, the learner does not do anything (Steps 10-12). It trusts that a new rule will be learned later, as it does not have a memory of the problem for which such priority was learned. The decisions for which a conflict resolving priority is learned do not need to be analyzed further by the learner and are removed from *ps_decisions* in Step 15. Finally the learner continues analyzing the reminding decision points in *ps_decisions* by calling the learning functions described in previous sections.

3.11.4 An Example in an Artificial Domain

Figure 3.46 introduces a small domain to illustrate how over-general learned control rules may give conflicting preferences, and how control knowledge to break those conflicts can be learned automatically. The domain has only three operators, namely *op*, op_1, and op_2. Operator *op* is always used to achieve the top-level goal g by decomposing it in instances of g_1 and g_2. op_1 achieves a given instance of g_1 and deletes all other instances. Similarly op_2 achieves a given instance of g_2 and deletes all other g_2's instances.[21] Better plans in this domain reuse g_1 and g_2

[21]For simplicity in the representation, assume that the negative effects (:dels) are updated before the positive effects (:adds).

subgoals common to several top-level goals before deleting them. An example of this occurs in problem P_1 in Figure 3.47 (a), in which two of the top level goals share subgoal $g_1(a_1)$. In the better plan those two goals are worked on consecutively and they are achieved before $g_1(a_1)$ is deleted in order to achieve $g_1(a_2)$. Figure 3.47 (c) shows the plan tree corresponding to the better plan. The control rule in (e) is learned from this problem by the algorithms described in Section 3.9. Similarly Figures 3.47 (b), (d), and (f) show the solutions, plan tree and learned control rule for a second problem in this domain. In this case $g_2(b_1)$ is the subgoal shared by two of the top-level goals.

Now PRODIGY4.0 is given a third problem in the domain (Figure 3.48). In this problem two of the top-level goals need $g_2(b_1)$ as a subgoal and a different pair of the top-level goals need $g_1(a_2)$ as a subgoal. Both of the rules learned from the two previous problems are relevant in this problem. Figure 3.48 shows the begining of the trace of PRODIGY4.0 solving this problem. At `n22` both rules fire indicating conflicting preferences. The conflict is broken by a default heuristic: subgoal on goal introduced most recently in the set of candidate goals. PRODIGY4.0 obtains the plan in Figure 3.49 (a) which can be improved. According to the quality metric for this domain, the cost of reachieving $g_2(b_1)$, i.e. the cost of op_2, is higher than the cost of reachieving $g_1(b_2)$, i.e. the cost of op_1. Therefore the plan is improved if the top-level goals are achieved in a different order. Figure 3.49 (b) shows the improved plan. The crucial difference between the two plans from the point of view of their quality is the different cost of op_1 and op_2.

In order to obtain the better plan, the goal preference conflict at `n22` should be broken in a different way. This amounts to giving priority to control rule `prefer-goal-2`, which prefers keeping the focus on $g(a_1, b_1)$. This priority among the two control rules is learned from this problem solving episode.[22] Note that the rule priority learned is related to the plan quality metric. Rule `prefer-goal-2` protects $g_2(y)$. Rule `prefer-goal-1` protects $g_1(x)$. The priority captures the greater cost of reachieving $g_2(y)$, i.e. of operator op_2.

3.11.5 Examples in the Process Planning Domain

In this section we illustrate the usefulness of learning priorities among learned control rules in the process planning domain. Figure 3.50 shows two control rules learned by the algorithms described in Section 3.7. The first rule suggests bindings for drilling a spot hole using a machine on which the part and holding device set-up are ready. The second rule suggests bindings that use an existing tool set-up. A conflict appears in a problem in which the part is being held by one machine $drill_1$ (the `holding` precondition is true), and the tool, a spot-drill, is being

[22] There is an even better solution to this problem in which the top-level goals are achieved in the order and therefore none of their subgoals needs to be reachieved. The learning algorithm is not able to learn the more complex and global heuristic that by inspecting the top-level goal finds the optimal order of achieving them.

Problem P_1

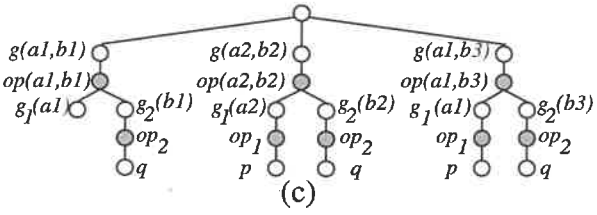
```
State: (and (p)(q))
Goal:  (and (g a1 b1) (g a2 b2) (g a1 b3))
Initial solution:
  1. <op1 a1>
  2. <op2 b3>
  3. <op a1 b3>
  4. <op1 a2>
  5. <op2 b2>
  6. <op a2 b2>
  7. <op1 a1>
  8. <op2 b1>
  9. <op a1 b1>
Cost = 30

Improved solution:
  1. <op1 a1>
  2. <op2 b3>
  3. <op a1 b3>
  4. <op2 b1>
  5. <op a1 b1>
  6. <op1 a2>
  7. <op2 b2>
  8. <op a2 b2>
Cost = 27
```

(a)

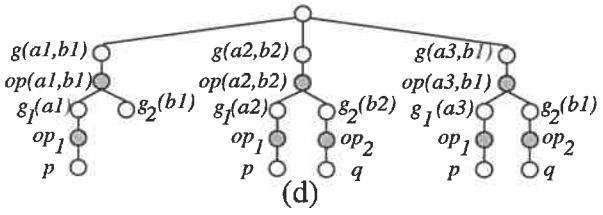
Problem P_2

```
State: (and (p)(q))
Goal:  (and (g a1 b1) (g a2 b2) (g a3 b1))
Initial solution:
  1. <op1 a3>
  2. <op2 b1>
  3. <op a3 b1>
  4. <op1 a2>
  5. <op2 b2>
  6. <op a2 b2>
  7. <op1 a1>
  8. <op2 b1>
  9. <op a1 b1>
Cost = 30

Improved solution:
  1. <op1 a3>
  2. <op2 b1>
  3. <op a3 b1>
  4. <op1 a1>
  5. <op a1 b1>
  6. <op1 a2>
  7. <op2 b2>
  8. <op a2 b2>
Cost = 24
```

(b)



(c)



(d)

```
(control-rule prefer-goal-1
 (if (and (candidate-goal (g1 <x-2>))
          (known (g1 <x-1>))
          (is-subgoal-of-ops (g1 <x-1>) <ops>)
          (diff <x-2> <x-1>)
          (first-pending-subgoal-in-subtree <pref-goal> <ops>)
          (diff <pref-goal> <other-goal>)
          (~(is-pending-subgoal-in-subtree <other-goal> <ops>))))
 (then prefer goal <pref-goal> <other-goal>))
```

(e)

```
(control-rule prefer-goal-2
 (if (and (candidate-goal (g2 <y-2>))
          (known (g2 <y-1>))
          (is-subgoal-of-ops (g2 <y-1>) <ops>)
          (diff <y-2> <y-1>)
          (first-pending-subgoal-in-subtree <pref-goal> <ops>)
          (diff <pref-goal> <other-goal>)
          (~(is-pending-subgoal-in-subtree <other-goal> <ops>))))
 (then prefer goal <pref-goal> <other-goal>))
```

(f)

Figure 3.47: (a) and (b): Two problems in the artificial domain of Figure 3.46 and their respective solutions. (c) and (d): Plan trees constructed from the improved solution traces. (e) and (f): Control rules learned respectively from problems (a) and (b).

```
State: (and (p)(q))
Goal:  (and (g a1 b1) (g a2 b2) (g a2 b1))

  2 n2 (done)
  4 n4 <*finish*>
  5   n5 (g a1 b1) [2]
  7   n7 <op a1 b1>
Firing delete goals EXPAND-MAIN-GOALS-FIRST
  8   n8 (g a2 b2) [1]
 10   n10 <op a2 b2>
Firing delete goals EXPAND-MAIN-GOALS-FIRST
 11   n11 (g a2 b1)
 13   n13 <op a2 b1>
 14     n14 (g1 a2) [3]
 16     n16 <op1 a2>
 17     n17 <OP1 A2>
 18     n18 (g2 b1) [2]
 20     n20 <op2 b1>
 21     n21 <OP2 B1>
 22   n22 <OP A2 B1>
Firing pref rule PREFER-GOAL-2 for (#<G1 A1>) over #<G2 B2>
Firing pref rule PREFER-GOAL-1 for (#<G2 B2>) over #<G1 A1>
Warning: cycle found in the preference control rules for type :GOAL
Resolving preference cycle...
Candidate #<G2 B2> has value 0.
Candidate #<G1 A1> has value 1.

 23     n23 (g2 b2) [1]
 25     n25 <op2 b2>
 26     n26 <OP2 B2>
 27   n27 <OP A2 B2>
...
```

Figure 3.48: A third problem in the artificial domain and PRODIGY4.0's problem solving trace for solving it. At n22 the control rules learned from the previous problems give conflicting preferences.

```
Initial solution:          Improved solution:
  1. <op1 a2>                1. <op1 a2>
  2. <op2 b1>                2. <op2 b1>
  3. <op a2 b1>              3. <op a2 b1>
  4. <op2 b2>                4. <op1 a1>
  5. <op a2 b2>              5. <op a1 b1>
  6. <op1 a1>              * 6. <op1 a2>
* 7. <op2 b1>                7. <op2 b2>
  8. <op a1 b1>              8. <op a2 b2>
Cost = 27                  Cost = 24
```

(a) (b)

Figure 3.49: Two plans of different quality for the problem in Figure 3.48. The difference in quality is due to the different cost of operators op_1 and op_2, marked with a * in the figure.

held in a different machine $drill_2$.[23] Both rules would fire suggesting different machines. The better solution according to our quality metric corresponds to using $drill_1$ since it reuses the most expensive part of the set-up, namely preparing the part and holding device and holding the part. Consistently with this, the rule priority learning mechanism learns a priority of the first rule over the second rule.

```
(control-rule pref-bnds-drill-with-spot-drill-in-milling-machine4
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (current-operator drill-with-spot-drill-in-milling-machine)
          (known (holding <mach4> <holding-dev5> <part> <side> <side-pair-6>))
          (or (diff <mach4> <mach1>) (diff <holding-dev5> <holding-dev2>)
              (diff <side-pair-6> <side-pair-3>))))
 (then prefer bindings ((<mach> . <mach4>)(<hd> . <holding-dev5>)(<sp> . <side-pair-6>))
                       ((<mach> . <mach1>)(<hd> . <holding-dev2>)(<sp> . <side-pair-3>)))))

(control-rule pref-bnds-drill-with-spot-drill-in-milling-machine2
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (current-operator drill-with-spot-drill-in-milling-machine)
          (known (holding-tool <mach3> <drill-bit-4>))
          (or (diff <mach3> <mach1>) (diff <drill-bit-4> <drill-bit-2>))))
 (then prefer bindings ((<mach> . <mach3>) (<drill-bit> . <drill-bit-4>))
                       ((<mach> . <mach1>) (<drill-bit> . <drill-bit-2>)))))
```

rule_relative_prefs = {...<prefer-bnds-drill-with-spot-drill-in-milling-machine4 over prefer-bnds-drill-with-spot-drill-in-milling-machine2> ...}

Figure 3.50: Two bindings control rules learned in the process planning domain. The algorithm in Section 3.11.3 learns a priority of the first rule over the second rule which is consistent with the meaning of the quality metric: changing a tool is cheaper than changing the set-up of the part and the holding device.

Figure 3.51 illustrates yet another example, this time when two goal prefer control rules conflict. The two rules were learned using the algorithms in Section 3.9. The first rule advises the planner to focus on the goals that need the holding set-up currently available. The second rule keeps the focus on the goals that need the current tool set-up. A priority is learned of the first rule over the second one and this priority is also consistent with the different cost of switching a tool and switching a part and holding device, an important factor captured by the plan quality metric.

3.12 Discussion

The previous sections have described how quality-enhancing control rules can be learned from problem solving experience. **Learning is triggered by failure**, when the current control

[23]Bindings control rules establish preferences only among valid bindings. Therefore they do not need to consider constraints on the bindings such as the type of the tool and machine.

```
(control-rule prefer-goal-1
 (if (and (candidate-goal (holding <drill15> <vise16> <part> <side27> <side3-side68>))
          (known (holding <mach1> <holding-dev2> <part> <side-3> <side-pair-4>))
          (is-subgoal-of-ops (holding <mach1> <holding-dev2> <part> <side-3> <side-pair-4>)
                             <ops>)
          (first-pending-subgoal-in-subtree <pref-goal> <ops>)
          (diff <pref-goal> <other-goal>)
          (~ (is-pending-subgoal-in-subtree <other-goal> <ops>))))
 (then prefer goal <pref-goal> <other-goal>))

(control-rule prefer-goal-2
 (if (and (candidate-goal (holding-tool <machine> <td22>))
          (known (holding-tool <machine> <drill-bit-1>))
          (is-subgoal-of-ops (holding-tool <machine> <drill-bit-1>) <ops>)
          (first-pending-subgoal-in-subtree <pref-goal> <ops>)
          (diff <pref-goal> <other-goal>)
          (~ (is-pending-subgoal-in-subtree <other-goal> <ops>))))
 (then prefer goal <pref-goal> <other-goal>))
```

rule_relative_prefs = {... <prefer-goal-1 over prefer-goal-2> ...}

Figure 3.51: Two goal control rules learned in the process planning domain and their learned priority. Again the priority captures the fact that in the plan quality metric changing a tool is cheaper than changing the set-up of the part and the holding device.

knowledge strategy does not lead to a good plan, according to the quality metric for the domain and to the critical eye of a domain expert. The divergences between the planner's initial solution and the improved solution, or better the divergences in the problem solving decisions made to arrive to each solution, suggest learning opportunities. These learning opportunities therefore correspond to decision points where the **current control strategy needs to be overridden** in order to obtain the improved solution. Initially that strategy is PRODIGY4.0's default one: depth-first search, left-right order of precondition expansion, default order of operators in the domain description, and default order of the bindings generated by the matcher. New control rules are learned only to override the current strategy. This need to learn when particular domain-independent search control heuristics do not produce the desired behavior is an issue for any planning architecture [Veloso and Blythe, 1994].

Previous work on learning for PRODIGY focused on learning control rules to **improve planning efficiency instead of plan quality** [Minton, 1988, Etzioni, 1990, Pérez and Etzioni, 1992]. The cited mechanisms worked for a constrained domain representation language that excluded disjunction and quantification. We are interested in more complex domains, since it is in those where plan quality is an interesting issue. Therefore our learning algorithms consider all of PRODIGY's rich domain knowledge representation language. As the focus of our thesis is on improving plan quality, we have assumed that control knowledge for planning efficiency can be learned by other mechanisms. In our experiments we have made use of such control knowledge as part of the domain description in order to speed the planning process and reduce

backtracking. However the use or lack of such control knowledge is not a factor for the success of our methods. The algorithm to build the plan trees takes into account only the successful paths in the search trace ignoring those who fail and lead to backtracking.

The algorithms described here learn goal, operator, and bindings *preference* rules. However PRODIGY4.0's control language is richer than that and more types of control rules are available. In particular select and reject control rules are more powerful in pruning candidate alternatives and reducing the search space. If an alternative is removed from the candidate set by one of these rules, it is not open for backtracking upon failure. Previous systems that learned control knowledge for PRODIGY [Minton, 1988, Etzioni, 1990] took advantage of this fact and learned select and reject control rules. **Why then learn only preference rules?** The reason is that our learner does not build complete explanations, or *proofs*, that the decision is always going to be a good one. As the rules are not guaranteed to be correct, learning select and reject control rules might lead to incompleteness. As the learned rules may be over-general we have chosen to allow several of them to fire[24] and then solve the preference conflict as described in Section 3.11.

Our learning mechanisms are fully independent of the planning domain. Nonetheless there are certain **characteristics of the domain and the plan quality metric** that make the methods succeed in learning control knowledge that improves the quality of the plans. Our method relies on the fact that the improvements in quality, in particular the savings in plan cost, are due to sharing the work among different parts of the plan. At problem solving time this means sharing subgoals among plan operators. If the operator and bindings choice can be guided so that a subgoal in its subgoaling tree is shared, plan steps are saved. These sources of improvement are nicely illustrated in the process planning domain where sharing of subgoals can be mapped to sharing of parts or subparts of a set-up, and the number of set-ups is commonly used by human experts as a rough measure of the quality of the plan (cf. Section 2.2). Similar characteristics appear in many other domains.

We have focused on quality metrics that capture the cost of the plan execution and in which the cost of the whole plan is the sum of the cost of the individual plan steps. The cost of a plan step depends of the cost of the operator used and the bindings with which it is instantiated. The fact that the **quality metrics are linear on the cost of the operators** is used by the algorithms, in particular to assign cost to the nodes of the plan trees. This disallows the use of nonlinear functions such as those in which the cost of the plan depends on the occurrence of combinations of plan steps, or in which the cost of one operator depends on the presence in the plan of another operator. There is a point of view however from which our quality metrics still capture nonlinearities as dependencies among steps, since the choice of operator, goal ordering, or bindings will influence further steps in the plan.

The improved plan can be obtained by asking a domain expert to criticize the first plan proposed

[24] PRODIGY4.0 allows only one select rule to fire.

by the planner. In this case our learner can be seen as a learning apprentice system [Mitchell *et al.*, 1990]. The improved plan can also be obtained by letting the planner search for multiple solutions and stop when a good enough solution is found. In this case the learning system would work completely autonomous as the domain expert is not required and the quality of the plan is determined solely by the use of the quality metric. **Why then did we design the system so the interaction with a human expert is possible?** A practical reason is that because of the large search spaces in complex domains, it would be too expensive to explore enough of it to find a good solution. The default search strategy is chronological backtracking. If the relevant decision (for example which operation to use to drill a hole) is made earlier in the search process, the space of alternatives explored lower in the tree before trying a different alternative at the relevant decision point is just too large. Allowing a domain expert to quickly point the better choices is much more efficient.

The **interaction with the human expert occurs at the plan level instead of at the problem solving level**. Note that by expert we mean an expert in the domain, not an expert in AI planning. For such expert it is easier to criticize the planner's solutions than its problem solving decisions. The expert does not need to understand how the problem solver works. It is easier for a person to think in terms of forward operator execution and change of the world state than in terms of backward chaining and means-ends analysis.

Lastly, there is a growing interest in the planning community for **mixed-initiative planning**. "A mixed-initiative system is one in which both humans and machines can make contributions to a problem solution, often without being asked explicitly." (Jaime Carbonell, Sr., cited in [Burstein and McDermott, 1994]). Allowing a domain expert to interact with the planner by criticizing a plan and suggesting improvements, and having the learner incorporate that expertise and use it in future problem solving, is a first step in the direction of an architecture with truly-mixed initiative. The learned knowledge captures a control strategy to generate plans of a quality consistent with the quality metric. If different quality metrics are available, different control strategies can be learned. Then for a given problem our planner can present to the domain expert different alternative plans, each of them good according to some quality metric.

The learning algorithms described suffer of important limitations. First of all, if the plan quality metric changes, the learned knowledge may become useless and, what is more important, incorrect. Second, the explanation is built from a single example. It is incomplete and leads to overgeneral rules. We have dealt with this problem by refining the learned knowledge upon failure. Third, we have described above that the class of quality metrics that can be captured is limited. If there are tradeoffs in the quality metric, there will be conflicts in the control rules that need to be broken in a more intelligent way than the one described. An example in the process planning domain illustrates this point. Assume that the quality metric we have used so far is modified to assign different cost to different operators for drilling a hole in a part. In particular using a milling machine becomes more expensive than using a drill press. Then the

cost of a plan to drill a hole depends not only on the availability of part of the set-up, but also on the type of operation used. The preference for one machine or other will depend on whether reusing parts of the milling machine set-up overcomes the savings in cost by using the drill press. If in addition several holes need to be made with the same set-up, the balance may be different.[25] These limitations are not only due to the algorithms themselves but to the choice of control rules as the formalism to represent quality control knowledge. These limitations motivated the approach described in Chapter 4.

3.13 Experimental Results

We have fully implemented the mechanisms to acquire automatically the types of control rules described in the previous sections. This section describes the results of our experiments aimed to evaluate the plan quality gained by using the learned control knowledge. We also analyze the cost of learning, and its effect along other dimensions.

3.13.1 The Setting

The experiments were run on the process planning domain described in Section 2.3. Some default control rules for planning efficiency[26] were available to the planner and reduce greatly the backtracking effort due to failure paths. These are orthogonal to the control rules learned for efficient plan execution, i.e. good plan quality. A *random problem generator* was built in order to generate problems for the experiments. The inputs to the generator are a set of constraints, namely the number of goals in the problem, the type of the goals, and the manufacturing environment, that is, how many machines, tools, holding devices, and so on, and of what types, are available. In the experiments we have concentrated in goals of cutting parts to desired sizes along their three dimensions, and on drilling and finishing holes of several different types (counterbored, countersunk, tapped, and reamed) in any of the six part sides.

The quality metric used in these experiments minimizes execution cost and in particular the cost of setting up the work on the machines. Table 3.1 describes the quality metric. Each operator is assigned a fixed cost independent of the operator instantiation (bindings). (Note however that QUALITY also allows metrics in which the cost of the operators depends on the operator bindings.)

The following two facts motivated our choice of quality metric:

[25] A similar example is ellaborated in Section 4.1.3.

[26] Those control rules prune early choices that would lead to failure paths.

Type	Cost	Operators
Machining operators	1	drill-with-spot-drill, drill-with-twist-drill, drill-with-high-helix-drill, tap, countersink, counterbore, ream, drill-with-spot-drill-in-milling-machine, drill-with-twist-drill-in-milling-machine, face-mill, side-mill
Machine and holding device set-up operators	8	put-holding-device-in-drill, put-holding-device-in-milling-machine, remove-holding-device-from-machine, put-on-machine-table, remove-from-machine-table, hold-with-vise, release-from-holding-device
Tool operators	1	put-tool-on-milling-machine, put-tool-in-drill-spindle, remove-tool-from-machine
Cleaning operators	6	clean, remove-burrs
Oil operators	3	add-soluble-oil, add-mineral-oil, add-any-cutting-fluid

Table 3.1: The quality metric used in the experiments described.

- "Set-ups are often the most expensive part of a plan. The number of set-ups roughly measures the cost of executing a plan, and it is commonly used as a heuristic to construct efficient plans."[Hayes, 1990]. Therefore our metric assigns a higher cost to the operators related to set-up plan steps, and assigns a uniform low cost to the machining operators (cutting, drilling, etc.)

- Switching tools is cheaper than holding or moving parts. In the machining center described in Section 2.4 tools are switched automatically, whereas moving holding devices and parts, cleaning the parts, and positioning the part precisely on the machine are done manually and therefore are more expensive operations.

The values assigned to each operator by the quality metric in Table 3.1 capture these aspects of the domain.

3.13.2 The Training Phase

QUALITY was given 60 randomly-generated problems in order to learn search control knowledge. For each of the 60 problems, QUALITY called PRODIGY4.0 to solve it and offered the human expert the plan obtained. If the expert proposed a better plan, QUALITY learned new control knowledge. The knowledge learned from that problem was incorporated and therefore used to solve the remaining problems. The expert proposed improvements to 19 of the 60 problems, and therefore QUALITY learned from 19 problems. Table 3.2 summarizes the results of the training phase.[27]

[27]The experiments used a Sun Sparcstation ELC running Allegro Common Lisp version 4.1 under the Mach/Unix operating system.

By *planning time* we mean the time spent by PRODIGY4.0 solving the problem initially and then solving it to obtain the expert-improved plan, by backtracking when PRODIGY4.0's decisions with the currently available control knowledge do not match the desired plan. *Learning time* includes the time to construct the plan trees from the problem solving traces, and the time to traverse and compare the plan trees and build the new control rules. Note that the *learning time* is considerably smaller than the *planning time*, that is, learning is cheap compared to planning.

- Average **planning** time (initial plan + improved plan): 26.5 s.
- Average **learning** time: 3.2 s.
- Quality-enhancing **rules** built (total): 36.

Table 3.2: Summary of the training phase. The numbers shown were computed for the 19 training problems in which learning was actually invoked.

Table 3.3 shows those results in more detail by separating the 60 training problems in 6 sets. Each of the sets was obtained by the random-problem generator for a particular set of parameters. The first three sets are one-goal problems to drill holes and spot holes, and to cut parts along one of their three dimensions. The next set contains problems with three goals to drill and/or finish holes in a part. The holes may be in different sides requiring setting up the part in the appropriate orientation. The problems in the last two sets have up to four goals both to cut parts and to drill and finish holes in one or more parts, and multiple tools are available in the shop to perform the required machining operations. The goals frequently interact as they may apply to the same part or can be achieved by sharing the shop resources (machines and/or tools). We show numbers for each problem set because different parameter settings lead to problems of increasing difficulty and to different usefulness of the learned knowledge. Note that overall efficiency improvements may not be as large for complex problems, where only part of the plan is affected by the new knowledge.

The second row of the table indicates in how many problems of the training set PRODIGY4.0 output a plan to which the expert could suggest any improvements. Only for those problems QUALITY was invoked and generated new control knowledge automatically. The third and fourth rows indicate planning and learning time for the problems in which QUALITY learned. The fourth row shows the number of rules learned for each set. Appendix C lists all those learned control rules.

The bottom part of Table 3.3 shows data on the quality of the plans obtained during the training phase. The control rules learned at each point were incorporated immediately and made available for solving the remaining problems. Thus the values reported in row 8 correspond to the costs for the plans with the control knowledge learned up to and included that problem.

Problem set (10 problems per set)	1	2	3	4	5	6
Number of problems learned from	3	3	1	3	6	3
Planning time (secs.)	24.8	34.8	26.1	91.8	140.5	186.6
Learning time (secs.)	2.5	2.9	8.1	7.7	15.7	24.6
Number of control rules learned	6	4	3	7	8	8

Cost before learning	263	309	335	835	655	1297
Cost after learning	245	240	307	668	517	1121
% Cost decrease (in improved problems only)	14%	45%	36%	48%	30%	32%

Table 3.3: Experimental results for the training phase. Each column corresponds to a set of 10 problems. The second row shows the number of problems of each problem set in which learning occurred, prompted by the expert suggestions to improve PRODIGY4.0's initial plan. The third and fourth rows show respectively the total time spent in planning and the total time spent in learning considering only those problems in which learning occured. Row 5 shows the number of control rules learned for each problem set. The bottom table shows plan quality data for the training problem. Each problem was solved using the control knowledge learned for the previous problems. The total cost of the plans initially output by PRODIGY4.0 for the 10 problems in each set is shown in row 7. Row 8 shows the total cost for the plans for the 10 problems in the set after the human expert suggestions. The last row shows the improvement in quality (or cost decrease) due to the expert suggestions, now considering only the problems for which the expert made any suggestions (row 2 from the top of the table).

With this in mind, the last row of the table shows the increase in quality for the problems of each set in which learning occurred.

3.13.3 The Test Phase

The goal of the test phase was to evaluate the plan quality gained by using the learned control knowledge. For the test phase PRODIGY4.0 was given 180 randomly-generated problems *different* from those in the training set. The 180 problems are grouped in six problem sets, numbered 1 to 6. Each problem set was produced by the random problem generator given the same parameters than for the corresponding problem set in the training phase.

3.13.3.1 Effect of the Learned Knowledge in Plan Quality

Table 3.4 shows the effect of the learned knowledge on the quality of the plans for those 180 problems, according to the quality metric of Table 3.1. Row 4 of the table indicates the number of problems in which the plan quality was actually improved. Row 5 shows the rate

of plan quality improvement considering only those problems.[28] The lack of improvement in the remaining problems was due to the fact that the planner was able to obtain a good plan without using control knowledge. In three of the 180 problems the quality of the plan decreased with the use of the learned knowledge, which indicates that there is room for further learning. Appendix D shows the results for each of the 180 problems.

Problem set (30 problems per set)	1	2	3	4	5	6
Cost without learned control knowledge	886	1528	1716	2811	3421	3834
Cost with learned control knowledge	689	755	1330	2056	1472	3245
Number of problems with improvement	9	24	14	25	30	20
% Cost decrease (in improved problems only)	49%	55%	40%	28%	57%	20%
Max % cost decrease	63%	71%	87%	88%	96%	34%

Table 3.4: Improvement on the quality of the plans obtained for 180 randomly-generated problems in the process planning domain. The numbers refer to the quality metric of Table 3.1. Quality improvement with that metric corresponds to execution cost reduction. The second row of the table shows the total quality of the plans obtained for the 30 problems in the each set without using any quality-enhancing control knowledge. Row 3 shows the corresponding values using the automatically acquired control knowledge. Row 4 shows the number of problems in each set in which the solution obtained using the learned knowledge was better than that obtained without any quality-enhancing control knowledge. Row 5 shows the rate of improvement considering only those problems. The last row shows the maximum rate of improvement in a single problem of the set.

To further evaluate the quality of the output plans, the human expert analyzed a sample of them and found them optimal in the sense that the expert could not suggest improvements (other than systematically exploring the complete space in the hope of finding some overlooked optimization). Although this is not guarantee of optimality as the expert did not exhaust the plan space, we consider them *virtually optimal.*

It is worth pointing out that the rate of quality improvement is quite different form the rate of planning efficiency (time or nodes explored by the planner) obtained by the speedup learning systems in the literature [Etzioni, 1990, Knoblock, 1994, Minton *et al.*, 1989, Veloso, 1994]. The reason for this is simple: as the planner's search space is exponential in size, in the best case a speedup learner can reach exponential improvements in search reductions. However the quality of plans according to our metrics is linear in the cost of the operators, and therefore cannot be improved exponentially. Still even a small quality improvement in production plans,

[28]These values reflect the improvement between the solutions obtained *without* any quality-enhancing control knowledge and those obtained *with* the learned control knowledge. Therefore they are not directly comparable to those in the last row of Table 3.3, which captured the improvement between the solution to a problem with the knowledge *learned so far* and the solution *suggested* by the human expert.

and in many other domains, is economically very important as the savings in cost and resource use multiply when the plans are executed many times.

3.13.3.2 Effect of the Learned Knowledge in Planning Efficiency

Although the goal of the learning algorithms introduced in this thesis is to produce better quality plans, we studied the effect of the learned knowledge in planning efficiency, in particular due to the overhead of matching the rules during planning. Table 3.5 shows the effects of using the learned control on the efficiency of planning, namely on the time spend and on the number of nodes expanded during planning. The numbers are averaged over 5 runs for each problem. The table shows how both the planning time and the number of expanded nodes are reduced when the learned rules are used. In spite of the slightly increased matching cost that the planner experiences when the control rules are added to the domain, the planner solves the problems faster due to the shorter length of the solutions obtained, and therefore the smaller number of nodes searched.[29] Therefore quality-rule learning has positive utility in the Minton sense [Minton, 1988] even for speed-up learning.

Problem set (30 problems per set)	Planning CPU time (secs.)		Number of nodes	
	Without learned rules	With learned rules	Without learned rules	With learned rules
1	84	71	1405	1135
2	141	99	2196	1594
3	152	204	2111	1852
4	272	249	3633	3083
5	313	214	3915	2432
6	509	426	5546	4590

Table 3.5: Effect of the learned control knowledge in the planning time and in the number of nodes searched. Each row displays the total planning CPU time and nodes for the 30 problems in the test set. The values shown are averaged over 5 runs for each problem. Columns 2 and 4 show the numbers without any quality-enhancing control knowledge. Columns 3 and 5 show the values when the control rules learned in the training phase are used during planning. Planning time is reduced (average 15%) when the learned control knowledge is used due to the shorter length of the plans found.

We have argued before that shorter plan lengths do not mean better plans with the quality metric used. However for many of the experiment problems the better plan was also the shorter plan. Note that because some planning efficiency select and reject rules are available in the domain (in all the experiments described), backtracking due to failure paths is greatly reduced. Most

[29] Appendix D shows the planning time and the length of the plans obtained for each problem.

times the planner is able to find a solution without much backtracking. Therefore the number of nodes explored was highly correlated with the plan length, and better plans required smaller numbers of explored nodes. This is the reason for the reduced CPU time shown in the table.

However for quality metrics in which better plans have greater length finding them may take more time and nodes, even if the matching cost of the learned control knowledge were null. Therefore a price in efficiency would necessarily be paid in order to obtain better quality plans. In future work we plan to further analyze the effort of using the learned knowledge, and the possible tradeoff between the matching cost and the savings obtained by using the learned knowledge instead of doing a more exhaustive search until a reasonably good solution is found according to the quality metric.

3.14 Summary

This chapter presented a procedure to automatically acquire search-control rules that guide the PRODIGY4.0 planner towards better plans. QUALITY, the learning architecture, is given a domain theory and a metric of the quality of the plans specific to that domain. It is also given a problem to solve in that domain. It compares the search trace (sequence of planning decisions made by the planner) with the current control knowledge, and the sequence of decisions required to obtain a better solution, according to the quality metric. The better solution is obtained by QUALITY in two alternative modes: autonomously, by searching until a better solution is found; or by interacting with a human expert in the application domain. QUALITY then translates both search traces into plan trees, compares the plan trees, and explains the differences in them in terms of why they lead to different quality plans. Finally, operator, binding, and goal ordering PRODIGY4.0 control rules are generated.

The interaction with the domain expert occurs at the level of plan steps (and not of problem solving decisions). Thus the domain expert remains oblivious to the planner's control representation language and planning algorithm.

QUALITY is domain-independent and has been fully implemented. We have reported experimental results in the process planning domain when the metric of quality is plan execution cost. The learned control rules considerably improve plan quality, generating plans hard to improve by the human domain expert. Learning is fast compared to planning. The learned rules have positive utility, as they speed up planning due to shorter plan lengths.

Chapter 4

Learning Control Knowledge Trees

Chapter 3 described how quality-enhancing search control rules can be automatically learned from past planning experience. The chapter concluded with some limitations of the learning approach and of using search control rules as a representation formalism to capture quality control knowledge. This chapter presents a novel way to represent control knowledge for improving plan quality, as well as domain-independent algorithms to use such knowledge and to automatically acquire it incrementally from experience. The first two sections motivate this approach. Section 4.3 introduces the new knowledge representation formalism that we call control knowledge trees. Section 4.4 presents the learning algorithms that *automatically build* control knowledge trees and Section 4.5 describes the procedures to *use them as control knowledge* to guide search. Both sections include illustrative examples from the process planning domain, aunque both the formalism and the algorithms described are independent of the application domain. Section 4.6 illustrates how the algorithms are used to produce good plans in a transportation domain. The chapter concludes with a discussion of this approach and some experimental results.

4.1 Motivation

The quality metric used throughout the process planning examples of Chapter 3 was a relatively simple one. It did not account for differences in the costs of the operators corresponding to machining actions. For example all the operators to drill a hole had the same quality value. Suppose that this quality metric (in Table 2.1) is modified so that different machining operators have different cost. In particular, using the drill press for drilling a hole or a spot-hole is much cheaper than using the milling machine. A problem is given to the planner in which the goals are to reduce the size of the part (by milling it) and to drill a hole in it. (A similar problem was described in Figure 2.5). The planner is confronted with a choice of operator to drill the hole.

Operator	Preconditions	Adds	Deletes
op_1	$p_{1i}, i = 1, ...n$	g	
op_2	$p_{2i}, i = 1, ...m$	g	
$op_{1i}, i = 1, ...n$		p_{1i}	
$op_{2i}, i = 1, ...m$		p_{2i}	

Table 4.1: Operators in an example artificial domain.

This choice may turn out to be a difficult one. There is a tradeoff between the savings in cost of the drilling operations per se, and the savings on seting up the work in each case. If the difference in cost between the drilling operators is large enough (the drill press being cheaper) it may be better to perform the two machining operations, namely drilling the spot hole and milling the part, in different machines. In that case the savings in cost due to the use of the drill press would be large enough to overcome the costs of setting up the work twice, once per machine. Setting up the work on the milling machine is needed in any case, since it is the only way to mill the part. Given all this, in this problem the choice of operator would depend both on the difference in cost of the alternative drilling actions and on how expensive is to set up the work on the drill press.

In general, complex quality metrics require reasoning about tradeoffs. And acquiring control rules that apply at individual decision points may prove insufficient. Instead, a more globally-scoped method is required. The next sections explore these issues with examples taken from several domains. The examples illustrate the limitations of the control rule learning algorithms of the previous chapter to capture more complex quality metrics. These limiting factors provide the motivation for the approach presented in this chapter.

4.1.1 Example 1: An Artificial Domain

The artificial domain that we now introduce illustrates the difficulty of capturing certain kinds of quality-enhancing control knowledge in the form of control rules. The domain operators are summarized in Table 4.1. Two operators, namely op_1 and op_2, achieve the top-level goal g. They have as preconditions $p_{1i}, i = 1, ...n$ and $p_{2i}, i = 1, ...m$ respectively. Each p_{ij} can be achieved by applying a corresponding single operator op_{ij} whose preconditions are always satisfied. None of the operators have any side effects (deleting or adding goals for which it was not chosen as relevant operator). Now consider a problem with goal g and initial state $P_1 \bigcup P_2$ where P_1 is the set of p_{1i} that are true initially and P_2 the set of p_{2i} that are true initially. A plan $\mathcal{P}_{op_1}$ to achieve g using operator op_1 will consist of operators op_{1i} such that $p_{1i} \notin P_1$, followed by op_1, and will have a quality value

$Q(\mathcal{P}_{op_1}) = q(op_1) + \sum_{p_{1i} \notin P_1} q(op_{1i})$

where $q(op_1)$ is the value assigned by the quality metric to op_1 and $q(op_{1i})$ is the value assigned by the quality metric to the operator op_{1i} that achieves p_{1i}. Lower values of q and Q correspond to better quality. Similarly, the quality of the plan $\mathcal{P}_{op_2}$ to achieve g using operator op_2 is

$Q(\mathcal{P}_{op_2}) = q(op_2) + \sum_{p_{2i} \notin P_2} q(op_{2i})$

Therefore if the planner is confronted with the choice of operator to achieve g, it should prefer op_1 if

$q(op_1) + \sum_{p_{1i} \notin P_1} q(op_{1i}) > q(op_2) + \sum_{p_{2i} \notin P_2} q(op_{2i})$

Let $\Delta_{op} = q(op_2) - q(op_1)$, that is, the difference in quality between the operators themselves, as determined by the quality metric. Then the control knowledge should suggest *prefer* op_1 if

$$\sum_{p_{1i} \notin P_1} q(op_{1i}) - \sum_{p_{2i} \notin P_2} q(op_{2i}) < \Delta_{op} \qquad [1]$$

Note that this value depends on the problem at hand, since P_1 and P_2 correspond to the initial state of the problem. There are many possible initial states, as there are many combinations of p_{1i} $i = 1, ..n$ and p_{2j} $j = 1, ..m$.

A problem with goal g and initial state defined by P_1 and P_2, has two solutions[1], namely the plans $\mathcal{P}_{op_1}$ and $\mathcal{P}_{op_2}$. Assume that plan $\mathcal{P}_{op_1}$ has better quality than $\mathcal{P}_{op_2}$. The conditions that contribute to the smaller value $Q(\mathcal{P}_{op_1})$ are p_{1i} s.t $p_{1i} \in P_1$, because they add 0 to $Q(\mathcal{P}_{op_1})$. The conditions that contribute to the higher value of $Q(\mathcal{P}_{op_2})$ are p_{2i} s.t $p_{2i} \notin P_2$, which respectively add $q(op_{2i})$ to $Q(\mathcal{P}_{op_2})$. These two sets of conditions *explain* why $\mathcal{P}_{op_1}$ is better than $\mathcal{P}_{op_2}$. A learner can use those sets of conditions to build a control rule of the form

if (current-goal g)
  $\bigwedge_{p_{1i} \in P_1}$ (known p_{1i})
  $\bigwedge_{p_{2i} \notin P_2} \neg$ (known p_{2i})
then prefer op_1 over op_2

Such rule is very specific to the problem from which it is learned: if the sets P_1 or P_2 change slightly, *the rule does not apply*, as all the rule preconditions must be true for it to fire. This is the key problem. There is *not a possibility of a partial match.* Rules that do not transfer from one problem to similar ones are far from ideal, and may lead to the utility problem [Minton, 1988], as the number of rules grows with the number of problems and rules are useful only for a small set of problems.

Generalizing the rule is not an easy task though. The quality difference, as shown in inequality [1], is due to the relative costs of the p_{ij} and of the operator alternatives. If some $p_{1i} \in P_1$ *is not* satisfied in the new problem's initial state, the sign of the equality may change, depending

[1]For the purposes of this discussion we are ignoring differences in the plans due to the ordering of the op_{ij} operators.

on the value of $q(op_{1i})$. If the sign changes, then operator op_2 should be preferred and therefore the precondition corresponding to that p_{1i} must stay in the rule so it does not fire in the new problem. However if the inequality sign does not change, choosing op_1 still leads to the better plan, but the rule does not fire in the new problem because it is too specific. Something similar happens if some $p_{2i} \notin P_2$ *is* satisfied in the new problem. In some cases changing the truth value in the initial state of one of the p_{ij} does not affect the relative quality of the plans, but changing a combination of them does.[2]

Generalization becomes even more difficult in a domain in which there exist several alternative operators for achieving each of the subgoals p_{ij}, and the planner needs to subgoal in turn to achieve the preconditions of those operators. The quality of the alternative operators may be different and depend on the particular operator instantiation chosen. Which alternative is preferred may also depend on which other goals the planner needs to solve. All this makes the generalization of the above explanation and the rule built from it a non-trivial task. The underlying problem is that matching a control rule requires that all of the preconditions are matched. If any of them fails to match, the rule does not fire. But there is no indication of how relevant the failed precondition(s) is to the rule recommendation, as the quality metric values are not captured in the rule and are not used when the rule is matched.

Additionally, as the *quality metric is implicitly captured in this control rule representation*, if the *quality metric changes the control rules become invalid*, as the recommendation may change. Then the control knowledge must be learned anew. Section 4.7.1 lists a number of situations in which a variety of metrics may be available for a domain. In those cases, if the learned control knowledge could be reused across quality metrics, the learning effort could be amortized among a larger number of problems in which it may be useful.

Given all these issues, what is needed is a formalism to represent control knowledge that explicitly captures the quality metric and uses it when such control knowledge is invoked to provide guidance for the planner's current decision.

4.1.2 Example 2: A Transportation Domain

This section illustrates the issues just presented by introducing a small train and van transportation domain. Figure 4.1 presents some of the operators. In this domain objects can be transported between cities using trains and vans. There are operators to load and unload trains and vans, an operator that moves a van between a pair of cities linked by a road, and an operator that moves a train between two cities connected by a railway.[3] Therefore there may be a choice

[2]It is interesting to note that in this example a control rule can be built whose precondition were precisely the test of inequality [1]. However this simple scheme would not work if achieving some p_{ij} would require subgoaling and therefore achieving other subgoals. The rule would require enumerating those subgoals.

[3]This domain is a variation of the logistics domain in [Veloso, 1994]. Jaime Carbonell suggested this example as a good one to illustrate the point of the previous section.

of transport to move an object between two cities. In general that choice influences considerably the quality of the final plan. The quality of the operators that move vehicles depend both on the class of vehicle, i.e. train or van, and on the distance traveled. Table 4.2 shows an example quality metric for this domain.

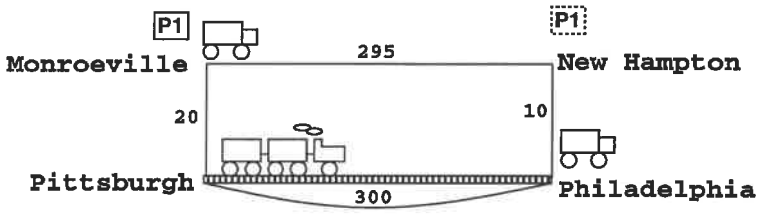
```
(OPERATOR LOAD-TRAIN
  (params <obj> <train> <city>)
  (preconds
   ((<obj> OBJECT)
    (<train> TRAIN)
    (<city> CITY))
   (and (train-station <city>)
        (at-obj <obj> <city>)
        (at-train <train> <city>)))
  (effects ()
   ((del (at-obj <obj> <city>))
    (add (inside-train <obj> <train>)))))

(OPERATOR UNLOAD-TRAIN
  (params <obj> <train> <city>)
  (preconds
   ((<obj> OBJECT)
    (<train> TRAIN)
    (<city> CITY))
   (and (train-station <city>)
        (inside-train <obj> <train>)
        (at-train <train> <city>)))
  (effects ()
   ((del (inside-train <obj> <train>))
    (add (at-obj <obj> <city>)))))

(OPERATOR RIDE-TRAIN
  (params <train> <from> <to>)
  (preconds
   ((<train> TRAIN)
    (<from> CITY)
    (<to> (and CITY (diff <from> <to>))))
   (and
    (railway <from> <to>)
    (at-train <train> <from>)))
  (effects ()
   ((del (at-train <train> <from>))
    (add (at-train <train> <to>)))))
```

Figure 4.1: Three of the operators in a train and van transportation domain. There are three additional operators similar to the three above for loading, unloading and driving a van.

Operators	Cost
load-train, unload-train(obj,train,loc)	5
load-van, unload-van(obj,van,loc)	5
ride-train(train,x,y)	distance(x,y)
drive-van(van,x,y)	5×distance(x,y)

Table 4.2: An example quality metric for the transportation domain which corresponds to plan execution cost. Operators for loading and unloading objects have a fixed cost. The cost of the operators for moving vehicles depends on the type of vehicle and on the distance traveled.

Figure 4.2 shows a very simple problem in this domain. Monroeville and New Hampton are cities in the neighborhood of Pittsburgh and Philadelphia respectively. Only the larger cities, Pittsburgh and Philadelphia, are connected by railways. The numbers next to the lines indicate distances between cities. Initially there is a van `ebro` at Monroeville and another van `avia` in Philadelphia, and a train `loco` ready at Pittsburgh's train station. The problem consists on




```
(state (at-obj package1 monroeville)
       (at-train loco pittsburgh)
       (at-van avia philadelphia)
       (at-van ebro monroeville) ...)
(goal  (at-obj package1 new-hampton))
```

Figure 4.2: An example problem in the transportation domain. A package must be transported from Monroeville, in the outskirts of Pittsburgh, to New Hampton, near Philadelphia.

Solution 1	Solution 2
load-van, drive-van(Mon,NH), unload-van	load-van, drive-van(Mon,Pit), unload-van, load-train, ride-train(Pit,Phil), unload-train, load-van, drive-van(Phil,NH), unload-van

Figure 4.3: Two plans for the problem in Figure 4.2. Some of the operator parameters have been omitted for clarity.

Total Plan Cost		cost(drive-van(van,x,y))		
		5×d(x,y)	1.25×d(x,y)	d(x,y)
1 package	Solution 1	1485	378	**305**
	Solution 2	**480**	**367**	360
2 packages	Solution 1	1495	**388**	**315**
	Solution 2	**510**	397	390

Table 4.3: Quality of different plans for two problems (moving one and two packages) in the transportation domain. The plans for the first problem are in Figure 4.3. The plans for the second problem are in Figure 4.4. Three different quality metrics are shown, each column corresponding to one of them. They differ in the cost of the *drive-van* operator relative to the distance traveled. The costs of the remaining operators are as shown in Table 4.2. The first column corresponds to the metric in that figure. The quality of the best plan in each case is highlighted.

moving a package `package1` from Monroeville to New Hampton. Consider the two solutions for this problem listed in Figure 4.3. The first one sends the package by van directly from Monroeville to New Hampton. The second one uses the train from Pittsburgh to Philadelphia

and the van for the shorter distances, i.e. from Monroeville to Pittsburgh and from Philadelphia to New Hampton. With the quality metric described, the first plan has quality value 1485, and the second plan has value 480. Although the total distance traveled in the second case is longer, the second plan is much better because traveling by train is less expensive.

The first and second row entries of Table 4.3 summarize the costs of the two plans using a variety of plan-quality metrics. The metrics differ in the cost of driving the van. The purpose of the table is to illustrate how for different metrics, different alternative actions need to be chosen. The quality of the best plan in each case is highlighted. For example, the third metric (right column) assigns the same quality to the individual operators for moving the package by van or by train, and therefore solution 1 is cheaper as the overall distance traversed by the package is much shorter. The choice depends both on the cost of the transport used and on the distance traversed in each of the solutions.

Solution 1	Solution 2
load-van(obj1, Mon), load-van(obj2, Mon), drive-van(Mon,NH), unload-van(obj1, NH), unload-van(obj2, NH),	load-van(obj1, Mon), load-van(obj2, Mon), drive-van(Mon,Pit), unload-van(obj1, Pit), unload-van(obj2, Pit), load-train(obj1, Pit), load-train(obj2, Pit), ride-train(Pit,Phil), unload-train(obj1, Phil), unload-train(obj2, Phil), load-van(obj1, Phil), load-van(obj2, Phil), drive-van(Phil,NH), unload-van(obj1, NH), unload-van(obj2, NH)

Figure 4.4: Two plans for a variant of the problem in Figure 4.2 in which *two* objects need to be transported between Monroeville and New Hampton.

The choice of action gets more complex when more than one object need to be transported. Two alternative plans for the case of two objects are shown in Figure 4.4. If the train route is selected, the objects have to be loaded and unloaded several times, increasing the cost of the plan linearly on the number of objects. The third and fourth rows of Table 4.3 show the costs of the two plans for different quality metrics. Note that in the case of the second metric (middle column), the choice of transport that leads to the best plan is different depending on the number of objects.

Figure 4.5 graphically shows the quality of the two plans for a variety of quality metrics. The desired quality-enhancing control knowledge must take into consideration the number of objects that must be transported and the metric that defines plan quality in order to make the appropriate choices that lead to the better plans. For the first and fourth metrics of the figure, the choice is clear (for less than 5 packages), i.e. using the van only and the train respectively. With the second metric (top rigth) 2 is the minimum number of objects that make solution 1

(using the van only) less costly. This is consistent with the data in Table 4.3. In the third metric (bottom left) 3 objects are sufficient to choose that solution.

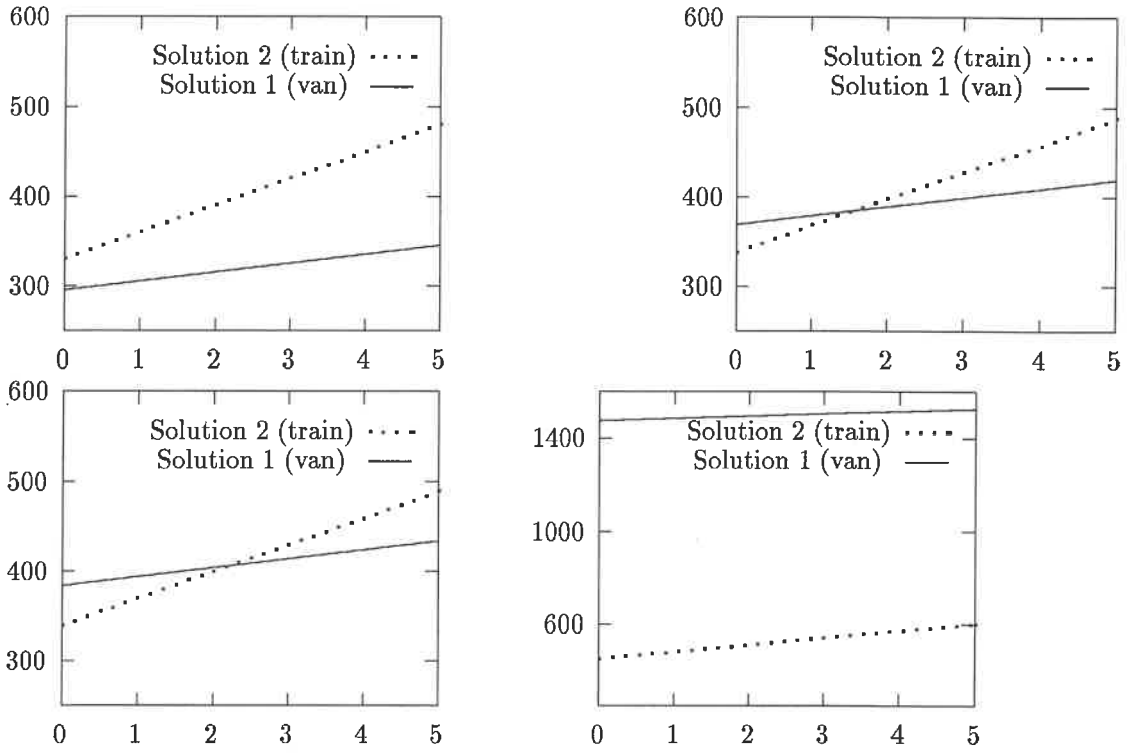


Figure 4.5: Four examples of quality metrics in the transportation domain and their influence in the planner's decision to obtain better plans. The quality metrics used are obtained from the one in Table 4.2 by making $cost(drive\text{-}van(van, x, y)) = m * distance(x, y)$ where m is 1, 1.25, 1.3, and 5 respectively (left to right, top to bottom). In each case the costs of solution 1 and solution 2 (Figure 4.3) are shown. The quality of each plan is plotted against the number of packages in the problem. The graphs show how the minimum number of packages that makes Solution 1 better varies with the quality metric.

This example illustrates how *plan length is not a good indicator of plan quality*. In this case it is not necessarily related to cost minimization. The longer plan can be much cheaper, as in the case of the first quality metric (left-most column of Table 4.3) where rail transportation is cheaper than hiring a van for a given distance. Planners would tend to output the first, shorter-length, solution unless they consider explicitly plan quality knowledge at planning time.

As in the example of Section 4.1.1 the kind of knowledge required to produce good plans is difficult to express in the form of control rules. The knowledge must capture non-local tradeoffs

	Q_1	Q_2
drill-in-drill-press	3	3
drill-in-milling-machine	5	10
...		

Goal	Initial state	Plan	Q_1	Q_2
1 spot hole	Set-up on mm7 (tool and part on mm7)	drill2	17	17
		mm7	**5**	**10**
2 spot holes	Set-up on mm7 (tool and part on mm7)	drill2	20	**20**
		mm7	**10**	**20**
	Tool on drill2, part held on mm7	drill2	18	**18**
		mm7	**12**	22
	Holding device on drill2, tool and part on mm7	drill2	**14**	**14**
		mm7	18	28

Table 4.4: Quality of plans for four problems in the process planning domain, where quality corresponds again to execution cost. The problems differ in the number of goals (one or two holes) and in the initial states (parts of the set-up ready in any of the machines). For each problem the quality of two plans is shown. The first plan for each problem uses a drill press drill2 to drill the spot hole(s). The second plan uses a milling machine mm7. Bold-faced numbers indicate the cost of the best plan for each of the problems. Two different quality metrics Q_1 and Q_2 are portrayed in the upper-right corner. Each metric assigns a different cost to drilling in the milling machine. The costs of the other operators do not change.

due to the existence of other goals (other packages) and to the differences in cost of the different alternatives. Section 4.6 will ellaborate this example.

4.1.3 Example 3: The Process Planning Domain

The issues discussed in the previous two sections are also relevant to the process planning domain. This section presents them in that context. Table 4.4 summarizes the quality of plans for four problems with different initial states, and goals to have one or two spot holes on the same side of a part. For each problem the quality of two different plans is shown, corresponding respectively to using a drill press drill2 and a milling machine mm7. Using the second quality metric (right-most column), the best plan for the first two problems (which have the same initial state) may change depending on the number of spot holes to drill, and more generally on the presence of more than one goal. In the third and fourth problems the choice of alternative is influenced both by the parts of the set-up already available, and by the quality metric used.

It is clear from the numbers in the table that a control rule, or a set of them, able to make the appropriate choice of operator in each of these four problems needs to capture the *tradeoffs* between the cost of the operators themselves, in this case the machining operators, and the cost of setting up the work. More generally, these tradeoffs may capture different criteria for quality evaluation, such as robustness versus cost, or execution time versus dollar cost (cf. Section 5.3.4).

```
(control-rule drillhole0
  (if (and (current-goal
            (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>))
           (known (holding <machine> <holding-device> <part> <side> <side-pair>))
           (type-of-object <machine> Milling-machine)
           (forall
            (and (type-of-object-gen <drill> DRILL)
                 (type-of-object-gen <hd> HOLDING-DEVICE))
            (and (~ (has-device <drill> <hd>))
                 (has-device <other-mach> <hd>)
                 (~ (is-clean <part>))
                 (~ (on-table <drill> <part>))))))
  (then prefer operator drill-with-twist-drill-in-milling-machine
                        drill-with-twist-drill))

(control-rule drillhole1
  (if (and (current-goal
            (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>))
           (known  (has-device <machine> <holding-device-4>))
           (type-of-object <machine> milling-machine)
           (known (on-table <machine> <part>))
           (known (has-spot  <part> <hole> <side> <loc-x> <loc-y>))
           (forall
            (and (type-of-object-gen (<holding-device> HOLDING-DEVICE))
                 (type-of-object-gen (<drill> DRILL)))
            (~ (has-device <drill> <holding-device>)))))
  (then prefer operator drill-with-twist-drill-in-milling-machine
                        drill-with-twist-drill))

(control-rule drillhole2
  (if (and (current-goal
            (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>))
           (known  (has-device <machine> <holding-device-4>))
           (type-of-object <machine> milling-machine)
           (known (on-table <machine> <part>))
           (~ (known (has-spot  <part> <hole> <side> <loc-x> <loc-y>)))
           (forall
            (and (type-of-object-gen (<holding-device> HOLDING-DEVICE))
                 (type-of-object-gen (<drill> DRILL)))
            (~ (has-device <drill> <holding-device>)))))
  (then prefer operator drill-with-twist-drill
                        drill-with-twist-drill-in-milling-machine))
```

Figure 4.6: Three control rules that prefer a different alternative for drilling a hole depending on the subparts of the set-up currently available. If several instances of the selected machine type are available in the problem, one or more bindings prefer rules, somewhat similar to the ones above, are needed to choose the correct machine.

Control rules to guide the planner towards the better plan in all these cases need to be specific enough to capture all the relevant details of the state and goal. In fact, they need to be much more specific than the rules learned by the algorithms of Chapter 3. Figure 4.6 shows three operator control rules that suggest an operator to drill a hole in a part. Recall that drilling in the drill press is cheaper in the milling machine. The first rule prefers drilling in the milling machine if the part is already being held there, and there is no drill press such that setting up the work on it would be cheap enough. Note that the drill press is universally quantified to make sure that no drill press would be better, and the rule prefers the operator independent of how it gets instantiated. If several instances of a milling machine are available, a bindings rule should choose the one that is actually holding the part. The rule in the figure is capturing the fact that even though the drill press operator is cheaper, if enough of the set-up is ready on the milling machine, the latter is preferred. The second rule says that if a holding device is ready on the milling machine, and no holding device is ready on the drill press, drilling in the milling machine is better. The third rule is very similar but prefers a different operator. Prior to drilling a hole, a spot-hole needs to be drilled. If the spot-hole must be drilled, the savings of drilling in the drill press add up and overcome the savings of using the partial set-up on the milling machine.

4.1.4 Limitations of Using Control Rules to Produce Quality Plans

The examples just described and the ones in the previous sections point out several limitations of using control rules to produce good plans in these cases:

- The rules capture very specific features of the state and goal in order to suggest a preference. Unfortunately there may be very many possible combinations of state facts and goals. See the simple example of drilling one or two spot holes in Table 4.4. Furthermore such very specific rules would transfer little to new problems, as Section 4.1.1 discussed for the example artificial domain. These rules would probably have low utility and be expensive to match, due to their long, specific preconditions, and thus cause the utility problem [Minton, 1988].

- The rules would probably turn useless if the quality metric changes. For example, if the gap in cost between the two drill operators becomes larger, the drill press operator will be preferred in most cases in spite of the set-up available in a milling machine.

- The rules suggest an alternative at a particular decision point, but that decision may affect the decisions that the planner should make at other points. The example rules suggest an operator to drill a hole. If a spot-hole must be drilled first, the choice of the best operator and bindings to drill a spot-hole is related to the choice made to drill the hole itself. A good plan uses the same machine for both operations. Although the control rules make

local decisions, the decisions should be *global* in nature. Capturing global decisions and solving *global tradeoffs* may be hard using local control rules.[4]

The difficulty of the decisions is greater when the difference in quality among alternative choices is small, as in the example quality metrics showed for the example domains. Consider now instead a metric that assigns a much higher cost to one of the alternatives. For example, consider a metric in which the cost of using the milling machine is much higher that that of using the drill press, and in fact it is so high that it would always be better to use the drill press, even if the part and tool were completely set on the milling machine. Then a simple control rule would be enough: prefer always using the drill press. That control rule would be tested (matched) and applied very efficiently without a need to consider the particular details of the problem, i.e. where the tool and part are located. However in our research we are more concerned with the more interesting cases in which the distinctions are tighter. (See for example the differences in quality in the problems of Table 4.4.) Note that in these cases the improvements in plan quality by making the right choice will not be as compelling, because the differences in quality between alternative plans are not very large. Still in real domains those small reductions in the plan quality value may lead to large economic (or other) savings.

4.1.5 Should We Still Learn Control Rules?

Motivated by the examples in the previous sections, we implemented a learning mechanism to build the kind of specific control rules described above. Section 4.1.1 suggested how such rules could be built. We are not going to present this algorithm in detail here. The algorithm compared the plan trees for the two different quality solutions for a given problem and extracted the conditions that supported the difference in quality: the rules were built with the conditions of cost 0 in the plan tree for the better plan, and the conditions with cost greater than 0 in the plan tree for the worse plan. Figure 4.7 shows the rule learned for a simple spot hole problem. The rule captures which parts of the set-up were and were not available and ready in each machine. In that training problem the available set-up in the milling machine is enough to overcome the higher drilling cost associated with it. The algorithm did not reason which of the conditions were necessary. As Section 4.1.1 pointed out, this is a non-trivial problem. For example, assume that in a new problem the last precondition of the rule is false, i.e. there exists a drill press also holding a spot-drill tool. If that is enough, for the given quality metric, to decide using the drill press, the rule is fine as it will not fire. Otherwise, the rule is too specific, would not fire for that new similar problem, and the learner would create another over-specific rule. Removing the precondition from the existing rule is not enough because in other problems

[4]Note that reasoning globally is always more expensive than reasoning locally. An interesting approach is taken in [Simmons, 1988a] by creating an initial plan using over-general, but local, rules and then "debugging" the plan with a small amount of global reasoning.

```
(control-rule prefer-drill-with-spot-drill-in-milling-machine9
   (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
            there is a <milling-mach> and a <holding-device> such that
              (and <milling-mach> is available
                   <holding-device> is empty
                   <milling-mach>'s table is available
                   the machine is holding a spot-drill tool <drill-bit>)
            for every drill press <drill7>
              (and <drill7> is not holding the part
                   the part is not on <drill7>'s table
                   no holding device is on <drill7>
                   <drill7> is holding a tool different from a spot-drill
                   every spot-drill <drill-bit6> is being held by some other machine)))
   (then prefer operator drill-with-spot-drill-in-milling-machinedrill-with-spot-drill))
```

```
(control-rule prefer-drill-with-spot-drill-in-milling-machine9
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (type-of-object-gen <milling-mach> milling-machine)
          (type-of-object-gen <holding-device>
                              (or 4-jaw-chuck vise collet-chuck toe-clamp))
          (forall (and (type-of-object-gen <part3> part)
                       (type-of-object-gen <side3> side)
                       (type-of-object-gen <side-pair3> side-pair))
            (and (~(holding-weakly <milling-mach> <holding-device> <part3> <side3> <side-pair3>))
                 (~(holding <milling-mach> <holding-device> <part3> <side3> <side-pair3>))))
          (forall (and (type-of-object-gen <part2> part)
                       (type-of-object-gen <holding-device2> holding-device)
                       (type-of-object-gen <side2> side)
                       (type-of-object-gen <side-pair2> side-pair))
            (and (~(on-table <milling-mach> <part2>))
                 (~(holding <milling-mach> <holding-device2> <part2> <side2> <side-pair2>))))
          (forall (type-of-object-gen <another-holding-device-1> holding-device)
            (~(has-device <milling-mach> <another-holding-device-1>)))
          (holding-tool <milling-mach> <drill-bit>)
          (type-of-object <drill-bit> spot-drill)
          (forall (and (type-of-object-gen <machine7> drill)
                       (type-of-object-gen <holding-device7> (or 4-jaw-chuck vise toe-clamp))
                       (type-of-object-gen <side-pair7> side-pair))
            (and (~(holding <machine7> <holding-device7> <part> <side> <side-pair7>))
                 (~(on-table <machine7> <part>))
                 (~(has-device <machine7> <holding-device7>))
                 (forall (type-of-object-gen <drill-bit6> spot-drill)
                   (and (~(holding-tool <machine7> <drill-bit6>))
                        (holding-tool <machine4> <drill-bit6>)
                        (holding-tool <machine7> <tool5>)))))))
 (then prefer operator drill-with-spot-drill-in-milling-machine drill-with-spot-drill))
```

Figure 4.7: A rule learned from a spot hole problem. The top part of the figure shows an informal description of the rule. The bottom part shows the actual rule. The rule says that if there is a milling machine that is free and holding the appropriate tool, and no drill press is even partially ready, prefer drilling the spot hole in the milling machine.

other preconditions may be the ones not satisfied, i.e. it is the combination of preconditions and their added achievement costs what matters. These rules do not transfer well because a single precondition being false makes the rule unapplicable. There is not a possibility of a *partial match.*

These limitations of control rules to represent quality control knowledge suggest the need to consider the relative costs of achieving each of the preconditions and of applying the operators themselves, and so to build control knowledge that can transfer better to new problems. Such mechanism would ideally:

- reuse quality control knowledge learned from a single problem, even when that knowledge is incomplete and there is only a partial match,
- represent the quality knowledge, that is the quality metric, more explicitly than the above control rules,
- produce control knowledge with a reasonable matching cost. That cost should depend on the difficulty of making the choice (cf. end of Section 4.1.3),[5]
- consider more global information and provide more global suggestions than control rules.

These desired characteristics lead to our design and implementation of the novel formalism to represent control knowledge described in this chapter.

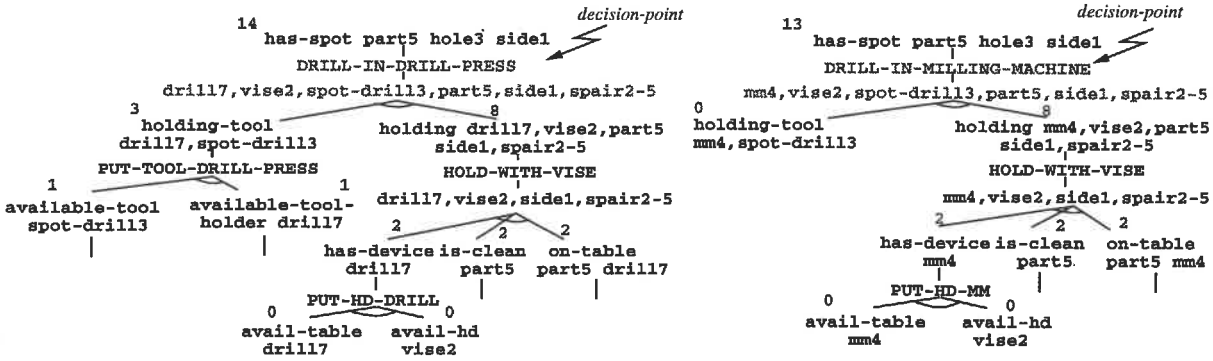
4.2 A Different Approach (A Sketch)

In this section we explain how control knowledge with the properties described above could be used and learned. Assume the planner is given a simple problem in which the goal is to drill a spot hole on a part and the *tool is initially set on the available milling machine*. Figure 4.8 summarizes the initial state and goal of the problem. The left side of Figure 4.9(a) shows the plan tree corresponding to the initial plan obtained by the planner, which uses operator *drill-in-drill-press* to drill the hole. When this plan was presented to a human expert, he or she improved it by choosing operator *drill-in-milling-machine* instead, as the tool is ready in the milling machine. The right-hand side plan tree of Figure 4.9(a) corresponds to that improved plan. Figure 4.9 (b) partially shows the quality metric. The plan that uses the milling machine (plan tree on the right) has slightly better quality as the higher cost of the drill in milling-machine operator is overcome by the savings of having the tool set already.

[5]It is interesting to see this as a tradeoff between operationality and generality of the learned knowledge, where the generality extreme would be to say just "choose the highest quality plan".

```
(objects
;;machines                              ;;tools
 (object-is mm4 MILLING-MACHINE)        (object-is spot-drill3 SPOT-DRILL)
 (object-is drill7 DRILL)               (object-is twist-drill5 TWIST-DRILL)
                                        ...
;;holding devices                       (object-is brush1 BRUSH)
 (objects-are vise2 VISE)               (object-is soluble-oil SOLUBLE-OIL)
                                        (object-is mineral-oil MINERAL-OIL))
;;parts and holes
 (object-is part5 PART)
 (object-is hole3 HOLE)

          (state (and (diameter-of-drill-bit twist-drill5 9/64)
                      ...
                      (holding-tool drill7 twist-drill5)
                      (holding-tool mm4 spot-drill3)))

          (goal (has-spot part5 hole3 side1 1.375 0.25))
```

Figure 4.8: A problem in the process planning domain. The goal is to drill a spot hole on `part5`. The spot drillbit is initially set on the available milling machine.



(a)

Operator	Cost	Operator	Cost
drill-in-drill-press	3	hold-with-vise	2
drill-in-milling-machine	5	put-holding-device-in-drill	2
put-tool-in-drill-press	1	clean	2
put-tool-in-milling-machine	1	put-on-machine-table	2
remove-tool-from-machine	1	...	

(b)

Figure 4.9: (a) The plan trees for two solutions to the problem of drilling a spot hole when the appropriate *tool is set on the available milling machine.* (b) The quality metric used in the example.

A new problem (Figure 4.10) with the same goal is now presented to the planner. The *drill*

```
(state (and (diameter-of-drill-bit twist-drill5 9/64)
            ...
            (holding mm4 vise2 part5 side1 side2-side5)
            (has-device mm4 vise2)))

(goal (has-spot part5 hole3 side1 1.375 0.25))
```

Figure 4.10: A new problem in the process planning domain, also to drill a spot hole on `part5`. In this case the part is ready on the milling machine `mm4`.

press tool holder is now free, and so is the spot drillbit. The *part is all set on the milling machine* `mm4`. How can the experience in the previous problem be used to find the best plan in the new problem? The planner knows that in the previous example the choice of operator to achieve the `has-spot` goal, and the particular instantiation of such operator, was relevant to obtain the better plan. Therefore, when the planner is solving the new problem and reaches that decision point, it stops and thinks about each of the alternatives and how good they may be according to the given quality metric.

Obviously, computing exactly the quality values of each alternative would amount to finding the complete plan, i.e. to solve the problem using each alternative. Instead we want to use the past planning experience to *estimate* those values. By looking at the plan trees for the previous problem the planner can decide how good each of the alternatives may be in the current problem. We sketch now that process and Figure 4.11 summarizes it (Q indicates the current quality estimate). To determine how expensive using the drill press would be, the planner could look at the plan tree on the left of Figure 4.9(a) and add the cost of the *drill-in-drill-press* operator to the costs of achieving the operator's preconditions (the grandchildren of the operator node). Those preconditions are in the second column in Figure 4.11. The cost of each precondition can be estimated recursively in the same way using the information in the plan tree. For example, the cost of the `holding-tool` subgoal is the cost of the *put-tool-drill-press* operator (1) plus the cost of each of its preconditions, namely `available-tool` and `available-tool-holder` (third column in the figure). In the previous problem these preconditions were achieved by applying some operators. In the current problem they are true in the curent state, and therefore have cost 0. Given all these, the estimate quality of this alternative (*drill-in-drill-press*) is 12. Similarly the cost of using the milling machine can be estimated. In the previous problem the tool was being held in the milling machine and therefore the `holding-tool` subgoal had cost 0. In the current problem the tool is not being held in the milling machine and therefore the `holding-tool` subgoal needs to be achieved, but there is no information in the plan tree to estimate the cost. The planner can only guess that the cost is going to be at least the cost of the cheapest operator relevant to that subgoal (1). The estimate found for this alternative (*drill-in-milling-machine*) is 7. Given the estimates found (12 and 7), the planner chooses to work on the alternative with the better one (*drill-in-milling-machine*) and continue planning.

drill-in-drill-press $Q \geq 3$	holding-tool $Q \geq 1$ holding $Q \geq 2$	is-available-tool $Q = 0$ is-available-tool-holder $Q = 0$ has-device $Q \geq 2$ is-clean $Q \geq 2$ on-table $Q \geq 2$
alt drill-in-drill-press: estimated $Q \geq 3$	≥ 6	≥ 12

drill-in-milling-machine $Q \geq 5$	holding-tool $Q \geq 1$ holding $Q = 0$	*don't know*
alt drill-in-milling-machine: estimated $Q \geq 5$	≥ 6	≥ 7

← **choose this alt**

Figure 4.11: A sketch of the process of reusing past experience to generate a good plan for the current problem. Reading the figure from left to right corresponds to traversing the plan trees of Figure 4.9(a) starting at their roots. Each of the two boxes corresponds to one of the alternatives at the decision point which were used in the past episode captured by the plan trees. The values of Q are the estimates for the alternative given the part of the plan tree explored so far.

The process described motivated the design of a new control knowledge representation formalism that we present in the next sections. We have only sketched here how the information from previous planning experience of the type of that stored in the plan trees can be useful to guide the planner towards better plans. Of course there are many issues not considered in this simple example. The next sections formalize the process into a new control knowledge representation formalism (Section 4.3) and the algorithms to learn (Section 4.4) and use (Section 4.5) such knowledge.

4.3 A New Representation Formalism for Control Knowledge

The previous section suggested a way in which previous problem solving experience can be used as control knowledge that guides the planner towards the better plan for the current problem. This section formalizes it proposing a new formalism, *control-knowledge trees*, to represent planning control knowledge. The next sections describe algorithms to learn automatically such control knowledge from planning experience and to use it during problem solving.

A control-knowledge tree (*cktree*) has three types of nodes corresponding respectively to

parameterized goals, operators, and bindings. They also correspond to the types of decisions in PRODIGY's planning algorithm. The root of a cktree is a parameterized goal. The cktree captures past problem solving experience relevant to that goal that can be used in future problems to prefer alternatives towards good solutions. The children of a goal node are operators that may achieve the goal, in fact they are only the operators that in the past have been used by the problem solver to achieve the goal.[6] The child of an operator node is a binding node. The children of a binding node are the preconditions of the corresponding operator. Figure 4.12 shows part of a cktree learned from the episode described in Section 4.2.[7]

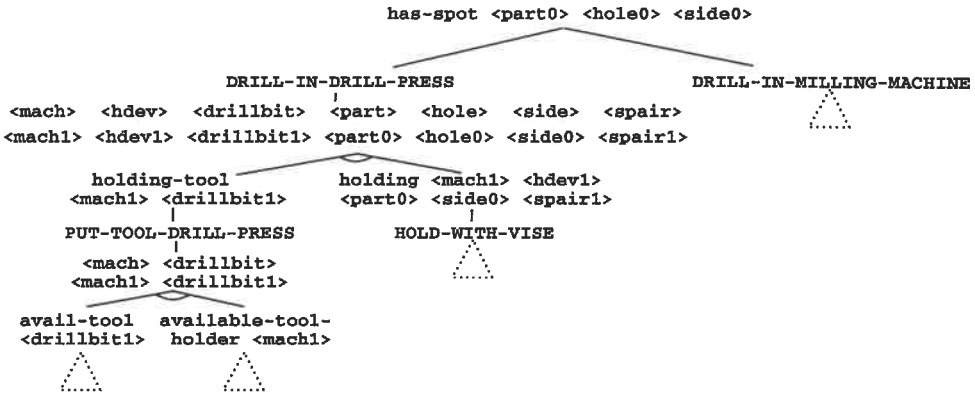


Figure 4.12: Partial view of the control knowledge tree generated automatically from the plan trees for the problem in Figure 4.9. Dotted triangles indicate subtrees omitted from the figure for simplicity purposes.

Note the similarity between the cktrees and the plan trees.[8] Section 3.5 explained how a plan tree is built from a successful path in the problem solving trace for a problem. On the other hand, a cktree is incrementally built from the two plan trees obtained in particular problem solving episodes as we will describe in the next sections. The differences between plan trees and cktrees are:

- A *plan tree* corresponds to a single planning episode. A *cktree* gathers planning experience from solving one or more problems.
- The root of a *plan tree* is PRODIGY's `(done)` goal and its descendants correspond to each of the current problem's top-level goals. Each goal is linked to the operator used to achieve it in that particular episode. On the other hand, a *cktree* captures the planner's past

[6] As in the case of the plan trees, operator nodes refer to both operators and inference rules. The cktree learning and matching algorithms do not make distinctions among the two types.

[7] The description in the footnote in page 43 for plan trees also applies to control knowledge trees.

[8] Section 4.7.1 analyzes the similarity between the cktrees and the statically-built problem space graphs (PSGs) [Etzioni, 1990].

experience for solving a particular goal in one or more different ways. The experience captured comes from problems in which that goal was either the only top-level goal, or one of several goals or subgoals.

- A goal node in a *plan tree* has only one child, namely the operator used to achieve the goal in the current episode. A goal node in a *cktree* may be linked to several operator nodes, which correspond to the operators that achieved the goal in past episodes. Therefore goal nodes are seen as OR nodes, since they propose alternative ways to achieve a goal. In both *plan trees* and *cktrees* the binding nodes are seen as AND nodes since all of the children, which correspond to the operator preconditions, must be achieved for the operator to be applicable. Also in both kinds of trees operator nodes have a single binding node child.[9]

- The nodes in the *plan trees* are completely instantiated, and they correspond to nodes in a given problem solving trace. The nodes in the *cktrees* are parameterized. They are created from plan tree nodes by replacing constants with variables. The variable types are constrained by the type specification of the operator variables, as declared in the operator schema.

The data structures that represent the nodes in a plan tree and in a cktree are the same, but the meaning of their slot contents varies slightly:

- The `children` slot of a *plan-tree goal node* contains the operator used to achieve that goal in the given problem. The children slot of a *cktree goal node* contains a list of operators that may be used to achieve the goal. In fact it only contains operators that were actually used in past problem solving episodes.

- The `name` slot of a *plan-tree binding node* contains the bindings used to instantiate the operator, i.e. a list of variable/value pairs, where the values correspond to real problem objects. The `name` slot of a *cktree binding node* contains a mapping between the names of the operator variables and the names of the variables used in the cktree. The variable names in the cktree nodes are created by *goal regression* from the root goal, and therefore they may be different from the names used in the operator schemata.

- The `children` slot of a *plan-tree binding node* contains a list of pointers to goal nodes corresponding to the operator preconditions instantiated for the particular problem, as they were generated during problem solving. The `children` slot of a *cktree binding node* contains a list of the operator's uninstantiated preconditions. If a precondition is universally quantified, it appears only as one child and an indication is kept that its variables are universally quantified in a slot called `forall-expanded-p`.

[9]Binding nodes in cktrees simply store the names of the operator variables as they appear in the cktree. We chose to have separate operator and binding nodes to facilitate cktree building as plan trees and PRODIGY4.0's own decision structure distinguish among those types of nodes.

- The `how-achieved` slot of a *plan-tree goal node* contains a pointer to another plan-tree node for the same goal, that was achieved first in the particular planning episode. The `how-achieved` slot in a *cktree goal node* contains one or more pointers to other cktree goal nodes that may achieve the same goal, since they did in some previous planning episode(s). The cktree nodes are parameterized. Therefore the fact that the linked nodes actually correspond to the same goal depends on the instantiation of their variables in the particular problem being solved. The same applies to the `achieves-too`, `how-deleted`, and `applied` slots of cktree nodes.

```
Goal cktree node          Operator cktree node     Binding cktree node
 :name                     :name                    :name
 :parent                   :parent                  :parent
 :children                 :children                :children
 :cost                     :cost                    :cost
 :marked-p                 :marked-p                :marked-p
 :how-achieved                                      :applied
 :achieves-too                                      :forall-expanded-p
 :how-deleted                                       :introduces-new-vars-p
                                                    :constraints
```

Figure 4.13: The data structures used to store each of the types of cktree nodes.

Figure 4.13 summarizes the slots for each of the types of cktree nodes. Some of the slots are particular to cktrees and do not appear in the plan trees. The first five slots appear in every cktree node. The `cost` and `marked-p` slots are used only at *cktree matching* time, that is, when the cktree is used to provide guidance for a planner's decision. They store the current estimated cost of the subtree rooted at that node, and a mark indicating whether that cost should be recomputed, for purposes of speeding up the matching. The meaning of these and the remaining slots will be described in the next sections as their use is justified.

Cktrees are used to guide the planner at any decision point in the process of achieving the goal at the root of the cktree. That is, the cktree may provide guidance for a *sequence* of operator and binding decisions. In this very simple example, the cktree may be used to suggest an operator to achieve the `has-hole` goal and an instantiation of that operator. Note that these decisions correspond to two different control points in the planner's algorithm. The cktrees could also provide operator and bindings guidance for the subgoals that appear in it (for example, an operator to hold the part) and that the planner will confront during search. Therefore the cktrees provide global guidance. By using the cktrees the planner makes use of global information to make decisions. These are major differences with using search control rules to capture quality knowledge.

The cktrees are generated from planning experience captured in the form of plan trees. Chapter 3 described how the plan trees are obtained automatically from a planning episode. From them,

the cktrees are built also automatically. Therefore the complete process of *cktree learning is fully automated* given the domain, a quality metric, and a planning episode. How cktrees are *built from planning experience* is the subject of the next section. Obviously the motivation for learning cktrees is to be able to *use them to guide planning* towards good solutions in new problems. Section 4.5 explains in detail the algorithms to use the learned cktrees as search-control knowledge.

4.4 Learning Control-Knowledge Trees

In the previous chapter (Figure 3.8) we described the top-level view of the procedure to learn quality-enhancing control knowledge. In Step 7 of the procedure the plan trees for the initial and improved plans are built. With those as an input, Step 8 is the core of the learning architecture. The previous chapter presented novel algorithms to learn quality-enhancing control rules. This chapter proposes a different implementation for the learning step (Step 8) of the procedure: learning control-knowledge trees. Note that Steps 1-7 are common for both learning approaches.

---

learn($plantree_A$,$plantree_B$,*decisions*) ;; op and bnds decisions only

1. *dec_point* ← **earliest_op_or_bnds_decision**(*decisions*)
2. g ← **parent_goal**(*dec_point*)
3. *ckroot* ← **relevant_cktree**(g,*cktrees*) ;; *cktrees* is a global variable
4. if *ckroot* = $\emptyset$
5. then **learn_new_cktree**($plantree_A$,$plantree_B$,*dec_point*) ;; Figure 4.18
6. else **learn_update_cktree**($plantree_A$,$plantree_B$,*dec_point*,*ckroot*) ;; Figure 4.29

Figure 4.14: Top-level call to the cktree learning mechanism.

Figure 4.14 describes the initial call to the cktree learning algorithm, the **learn** function. Its inputs are the two plan trees $plantree_A$ and $plantree_B$ corresponding respectively to the improved and initial plans. **Learn** has also as an input the set of *decisions* in which the choices made in order to output the improved plan were different from those suggested by the existing control knowledge. Section 3.4 described how those decisions are obtained as a problem solving trace is created for the expert-given improved plan. In Step 1 of Figure 4.14 the learner focuses in the earliest of those decisions and will learn control knowledge in the form of a cktree to make the right choice at the decision point. Earliest refers to the order in the planning decisions (not in the final totally-ordered plan). We will refer only to operator and bindings decisions. (See Section 4.5.11 for a discussion of goal decisions.)

The root of a cktree is a parameterized goal (Section 4.3) that corresponds to the top-level goal above the decision for which the cktree was built (Step 2). For example, a cktree rooted at a

`has-spot` goal is learned if the planner's decision involved choosing between operators *drill-in-drill-press* and *drill-in-milling-machine* in order to achieve a `has-spot` goal, or choosing bindings for one of those operators. **Relevant_cktree** (Step 3) looks for an existing, previously learned, cktree relevant to the decision, that is, one whose root matches the top-level goal that is an ancestor of the decision. Section 4.4.2.1 explains the motivation for this choice. *cktrees* is a global variable that stores the roots of the existing cktrees. Depending on whether such cktree exists, the learner has two alternatives:

- If no existing cktree is relevant to *dec_point*, build a new one by calling **learn_new_cktree** (Step 5).
- If there is a relevant cktree *ckroot*, update it. The current knowledge in the cktree was incomplete, or else a correct decision would have been made at *dec_point* in which case *dec_point* would have not been in *decisions*. Therefore the existing relevant cktree is updated in Step 6 by calling **learn_update_cktree**.

Both the creation and update of cktrees can be seen as a process of translation and generalization of the information stored in the plan trees. The aim is to store problem solving experience about preferred and preferred-over alternatives in a way that can be efficiently reused in the future. The next subsections describe in detail the creation and update algorithms, but before that we introduce an example in the process planning domain that will serve to illustrate the learning process.

4.4.1 An Example

Figure 4.15(a) summarizes a problem in the process planning domain that will be used throughout the next sections. The goal is to *drill a hole* of a given diameter on a part. In the domain description there are at least two operators to drill a hole, including *drill-with-twist-drill*, which uses a drill-press, and *drill-with-twist-drill-mm*, using a milling machine. Both operators have as a precondition the existence of a spot-hole at the target location.

Initially the part and the spot-drill, that is, the tool to drill the spot-hole, are set on the milling-machine. Figure 4.15(b) shows two plans of different quality to solve that problem. The quality metric is partially described in Figure 4.15(c). The first plan, using the drill press, was output initially by the planner in the absence of quality control knowledge. The second plan, using the milling machine, corresponds to the improvements made to the first plan by the human expert. Note that although the operator to drill in the drill press is cheaper, the plan that uses the milling machine is better because the work is initially set on the milling machine. Figure 4.16 shows the planning decisions in which the planner made choices different from the default ones in order to obtain the improved plan. The learner will focus on the earliest of those decisions, choosing

```
(objects
;;machines
 (object-is mm4 MILLING-MACHINE)
 (object-is drill7 DRILL)

;;holding devices
 (objects-are vise1 vise2 VISE)

;;parts and holes
 (object-is part5 PART)
 (object-is hole1 HOLE)

;;tools
 (object-is spot-drill3 SPOT-DRILL)
 (object-is twist-drill5 TWIST-DRILL)
 (object-is tap4 tap)
 ...
 (object-is brush1 BRUSH)
 (object-is soluble-oil SOLUBLE-OIL)
 (object-is mineral-oil MINERAL-OIL))
```

```
(state (and (diameter-of-drill-bit twist-drill5 9/64)
            (diameter-of-drill-bit tap4 9/64)
            (is-clean part5)
            ...
            (holding-tool drill7 tap4)
            (has-device drill7 vise1)
            (holding-tool mm4 spot-drill3)
            (holding mm4 vise2 part5 side1 side3-side6)
            (has-device mm4 vise2)))

(goal (has-hole part5 hole1 side1 0.3 9/64 1.375 0.25))
```

(a)

1. *remove-tool drill7 tap4*
2. *remove-tool mm4 spot-drill3*
3. *put-tool-drill drill7 spot-drill3*
4. *release mm4 vise2 part5 side1 side3-side6*
5. *put-on-machine-table drill7 part5*
6. *hold-with-vise drill7 vise1 part5 side1 side3-side6*
7. *drill-with-spot-drill drill7 spot-drill3 vise1 part5 hole1 side1 side3-side6*
8. *remove-tool drill7 spot-drill3*
9. *put-tool-drill drill7 twist-drill5*
10. *drill-with-twist-drill drill7 twist-drill5 vise1 part5 hole1 side1 side3-side6*

cost = 17

1. *drill-with-spot-drill-mm mm4 spot-drill3 vise2 part5 hole1 side1 side3-side6*
2. *remove-tool mm4 spot-drill3*
3. *put-tool-mm mm4 twist-drill5*
4. *drill-with-twist-drill-mm mm4 twist-drill5 vise2 part5 hole1 side1 side3-side6*

cost = 12

(b)

Operator	Cost
drill-with-twist-drill, drill-with-spot-drill (in drill press)	3
drill-with-twist-drill-mm, drill-with-spot-drill-mm	5
put-tool-drill, put-tool-mm, remove-tool	1
hold-with-vise, put-holding-device, put-on-machine-table, release	2
clean	2
...	

(c)

Figure 4.15: (a) A problem in the processs planning domain. (b) Two solutions for that problem. The plan on the left is the plan initially obtained by the planner. The plan on the right is the improved plan suggested by a human expert. (c) Quality metric used in this example (higher values indicate lower quality).

an operator to achieve `has-hole`, as indicated in Step 1 of **learn** (Figure 4.14). Figure 4.17 shows the plan trees built for the two plans.

> At goal node 5 `(has-hole part0 hole0 side1 0.3 9/64 1.375 0.25)`
> operator *drill-with-twist-drill-in-milling-machine* was chosen over
> operator *drill-with-twist-drill.*
>
> At goal node 11 `(has-spot part0 hole0 side1 1.375 0.25)`
> operator *drill-with-spot-drill-in-milling-machine* was chosen over
> operator *drill-with-spot-drill.*

Figure 4.16: Planning decisions forced in order to obtain the improved, user-given plan of Figure 4.15 (b). Step 1 of Figure 4.14 chooses the first of those decisions.

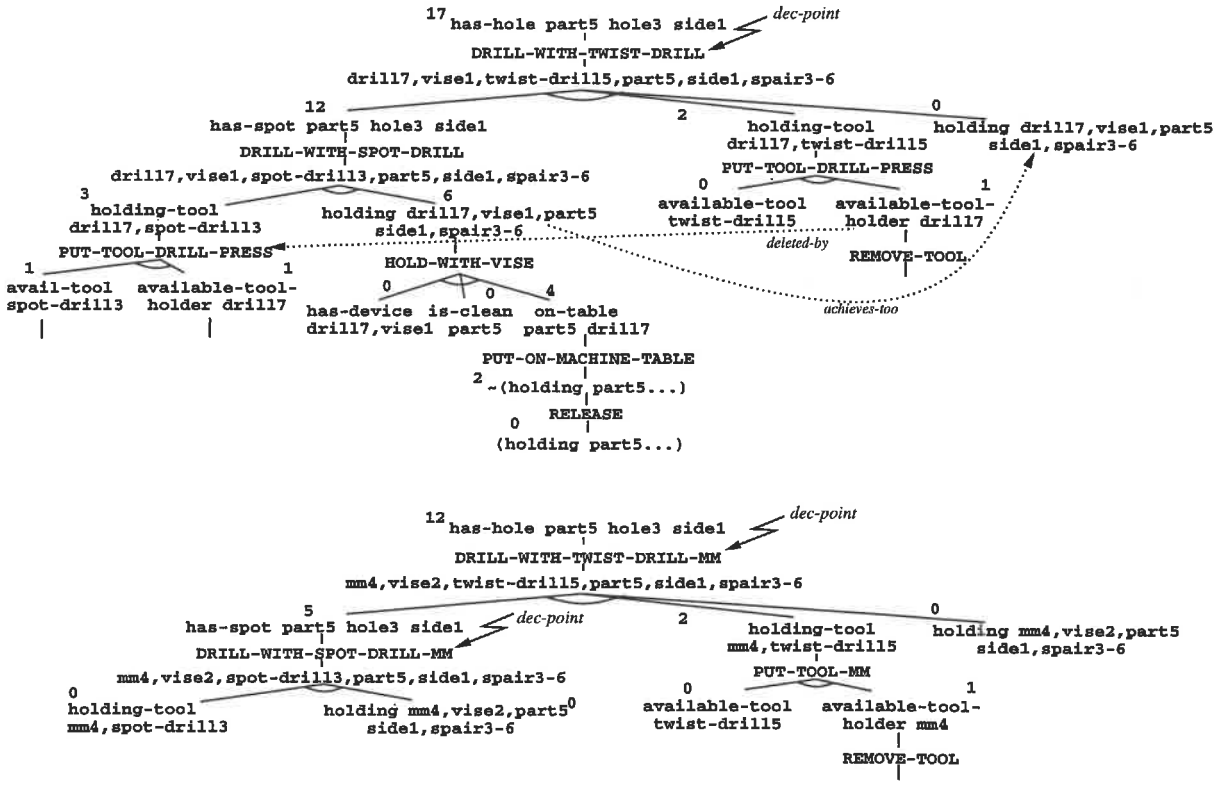


Figure 4.17: Plan trees corresponding to the plans of Figure 4.15(b). Some nodes have been omitted for clarity purposes.

learn_new_cktree($plantree_A$,$plantree_B$,*dec_point*)

1. $q_A \leftarrow$ **relevant_subtree**($plantree_A$,*dec_point*)
2. *ckroot* $\leftarrow$ make_ck_goal_node :name **parameterize**(q_A)
3. **build_ck_op**(q_A,*ckroot*) ;; Figure 4.19
4. **learn_other_cktree**($plantree_A$) ;; Figure 4.33
5. $<q_B, g_{ck}> \leftarrow$ **which_qnode_and_cknode**(*ckroot*,$plantree_B$,*dec_point*)
6. **build_ck_op**(q_B, g_{ck})
7. **learn_other_cktree**($plantree_B$)
8. *cktrees* $\leftarrow$ push(*ckroot*,*cktrees*)

Figure 4.18: Building a new cktree.

4.4.2 Building a New Control-Knowledge Tree

Let *dec_point* be the choice point for which learning is invoked (cf Step 1 of Figure 4.14). If no cktree is available to guide the decision at *dec_point*, **learn_new_cktree** (in Figure 4.18) uses the plan trees of the current problem solving episode to build a new cktree. The process of building the cktrees can be seen as translating and generalizing the parts of the plan trees that are relevant to the decision at *dec_point*. For example, if the problem goal was the conjunction of two independent goals g_1 and g_2, the root of the plan tree for a solution to that problem will have two children s_1 and s_2 corresponding to the subtrees rooted at g_1 and g_2 respectively. If the decision at *dec_point* is only relevant to g_1, only the s_1 subtree will be considered when building the new cktree. The fact that the subgoals are independent is captured by the way the plan trees are built, in particular in the achievement and deletion links.

4.4.2.1 The Root of the Cktree

The first step in building a new cktree is creating its root node (Steps 1-2 of Figure 4.18). Recall that $plantree_A$ ($plantree_B$) is the plan tree corresponding to the improved (initial) plan. The algorithm looks for the node q_A in $plantree_A$ such that (a) q_A corresponds to a top-level goal, i.e. it is a child of the $plantree_A$ root; and (b) it is an ancestor of the node that corresponds to *dec_point*. The goal at node q_A is parameterized (by replacing constants with variables) and becomes the root of the new cktree (Step 2). The cktree is built starting at that root by traversing the subtree of $plantree_A$ rooted at q_A and building a cktree node for each plan tree node (Step 3).

Next (Steps 5-6) $plantree_B$ is processed to capture in the cktree the information from the decisions that lead to the initial, worse quality, solution. Recall that the purpose of storing that is to estimate the cost of different alternatives in future problem solving at a similar decision

point. This problem solving knowledge from $plantree_B$ is stored in the cktree just built from $plantree_A$ and is attached as a new subtree under the appropriate cktree node. $plantree_A$ and $plantree_B$ divergences start at *dec_point*. Let g_{ck} be the cktree goal node corresponding to *dec_point* and q_B the plan tree goal node in $plantree_B$ also corresponding to *dec_point* (Step 5). The new cksubtree will be a child of g_{ck}, and it will be built by traversing the subtree of $plantree_B$ rooted at q_B.

In the example of Section 4.4.1 *dec_point* corresponds to the choice of operator to achieve `has-hole`. The plan tree in the bottom (top) of Figure 4.17 corresponds to $plantree_A$ ($plantree_B$). The root of the bottom plan tree ($plantree_A$) is parameterized to obtain the root of the cktree: `(has-hole <part0> <hole0> <side0>)`. Then $plantree_A$ is used to start building the cktree. The first child of the cktree root (Step 3) will be an operator node named *drill-with-twist-drill*. After that subtree is built, the top plan tree ($plantree_B$) is used to build another subtree of the cktree for `has-hole` rooted at operator *drill-with-twist-drill-mm* (Step 5). Next section describes how the cktree nodes are built.

Assume for a moment a slightly different case in which learning is needed to guide the choice of operator for `has-spot`, and $plantree_A$ is the same as in the previous example. $plantree_B$ is different, sharing the first three level of nodes with $plantree_A$ but diverging on the choice of operator for `has-spot`. In this case the root of the cktree would still be `has-hole`, the top-level goal in the problem. However the plan tree for the worse plan ($plantree_B$) would produce a subtree rooted at cktree goal node `has-spot` (Step 5). The motivation for learning a cktree for `has-hole` instead of `has-spot` (the decision point) is to capture the context for the decision: a good choice of operator for drilling the spot hole in this example is related to the operator chosen to drill the hole.[10]

If the goal for which the cktree is being built interacts with other goals in the problem, those other goals should also be considered to provide guidance to the planner. Therefore the learner also builds cktrees for the other goals in the problem that interact with the one for which guidance is being learned (Steps 4 and 7). Section 4.4.4 ellaborates on this.

4.4.2.2 Creating Cktree Nodes

The actual construction of the cktree is done by calling recursively a set of functions (**build_ck_op**, **build_ck_bindings**, and **build_ck_goal**) that create cktree nodes of different types. In the description that follows a ck subindex indicates a cktree node, and a q subindex indicates a plan tree node.

[10]In the current implementation if a cktree for a top-level goal does not already exist, it is built from scratch and does not reuse (by pointing at them) possibly existing cktrees rooted at its subgoals. There is no sharing of cktrees. Adding that capability would increase the transfer of the learned knowledge to new problems and reduce the space to store the cktrees.

build_ck_op(p_q, p_{ck}) ;; p_q is a plan-tree goal node; p_{ck} is a cktree goal node

1. $op_q \leftarrow$ **child**(p_q)
2. $op_{ck} \leftarrow$ make_ck_op_node :name **operator_name**(op_q)
3. :parent p_{ck}
4. **children**(p_{ck}) $\leftarrow$ push(op_{ck},**children**(p_{ck}))
5. if **decision_was_made_p**(p_q) ;; add unsuccessful alternatives
   then
6. *other_dec_point* $\leftarrow$ **decision_point**(p_q) ;; node in search trace corresponding to p_q
7. for each *other_op* $\in$ **bad_alts**(*other_dec_point*)
8. *other_op*$_{ck}$ $\leftarrow$ make_ck_op_node :name *other_op*
9. :parent p_{ck}
10. **children**(p_{ck}) $\leftarrow$ push(*other_op*$_{ck}$,**children**(p_{ck}))
11. **build_ck_bindings**(op_q, op_{ck}) ;; Figure 4.20

Figure 4.19: Building a cktree operator node. In this and the next figures a q subindex indicates a plan tree node and a ck subindex indicates a cktree node.

Creating an operator node

Build_ck_op (Figure 4.19) is given a plan tree goal node p_q and a cktree goal node p_{ck} and creates a cktree operator node op_{ck}, which becomes a child of p_{ck}. The operator at the new cktree node is op_q, the operator chosen during planning to achieve p_q. op_q and op_{ck} are passed as inputs to **build_ck_bindings** to build the child cktree binding node (Step 11).

Since the cktree will be used to guide a sequence of planning decisions (not only the top one), it is useful to store at each cktree node other alternatives that were explored but rejected at planning time. The reason for the rejection could be that the alternative caused the planner to fail and backtrack, or that the alternative was not in the path to achieve the desired solution (the improved expert-given solution). Both kinds of abandoned alternatives are bookept by the planner at problem solving time and by the backtracking mechanism described in Section 3.4. The existence of failed alternatives is checked by **decision_was_made_p** (Step 5). If they exist, that is, if other operator alternatives were tried at planning time when solving goal p_q, they are also added to the cktree as children of p_{ck} (Steps 6-10).

Creating a binding node

Build_ck_bindings (Figure 4.20) is given a plan tree operator node op_q and a cktree operator node op_{ck} and creates a cktree binding node b_{ck}. The binding node b_{ck} stores the mapping between the variables used throughout the cktree and the variables used in the operator schema.

The call to **generic_bindings** constructs such mapping: Given p_{ck}, parent goal of op_{ck}, the operator is partially instantiated using the arguments of p_{ck} to bind variables in the operator right-hand side. Note that the arguments of p_{ck} are variables but are used as constants in this instantiation process. Which of the operator effects was relevant for the parent goal in the planning episode is determined from op_q (in the plan tree). This is needed because the operator may have different effects with the same predicate name, that is, it could be instantiated in different ways to achieve p_{ck}. New names are created for the remaining operator variables that are not bound from the operator right-hand side. If the operator introduced universally quantified variables, the slot `forall-expanded-p` is set.

---

build_ck_bindings(op_q, op_{ck}) ;; op_q is a plan-tree operator node
;; op_{ck} is a cktree operator node

1. $b_q \leftarrow$ **child**(op_q)
2. $b_{ck} \leftarrow$ make_ck_binding_node :name **generic_bindings**(op_q, op_{ck})
3. :parent op_{ck}
4. :forall-expanded-p **forall-expanded-p**(op_q)
5. if **side_effects_p**(b_q) then **fill_side_effects**(b_q, b_{ck}) ;; bookkeep side effects (Figure 4.27)
6. **children**(op_{ck}) $\leftarrow \{b_{ck}\}$
7. **build_ck_g**(b_q, b_{ck}) ;; Figure 4.22

Figure 4.20: Building a cktree binding node.

Finally the binding node b_q in the plan tree, and the newly created binding node b_{ck} in the cktree are passed as inputs to **build_ck_goal** in order to build the children goal nodes corresponding to the operator preconditions.

Figure 4.21 shows how the first operator and binding nodes of the `has-hole` cktree are being built in the example. Figure 4.21 (a) shows the part of $plantree_A$ relevant to these first steps; (b) shows the first nodes built in the cktree, namely the root goal node, the operator node (*drill-with-twist-drill-mm*), and the binding node. The binding node stores the mapping between the variables in the operator schema (e.g. `<part>`, `<mach>`) and the variables used throughout the cktree (e.g. `<part0>`, `<mach1>`). Some of these variables are regressed from the root goal node (e.g. `<part0>`). Others are introduced by the operator itself (`<mach1>`) and correspond to the variables for which bindings would be chosen at the binding node. New, unique names are generated for those variables. (The coming figures will depict binding nodes more succintly for clarity purposes.)

Side effects are the effects of an operator application different from the effect that matches the goal for which the operator was selected as relevant. In addition to subgoaling parent/child links, a plan tree stores information about achievement and deletion caused by the side effects of operator applications (cf Section 3.5). The side effects of a plan tree binding node b_q are stored

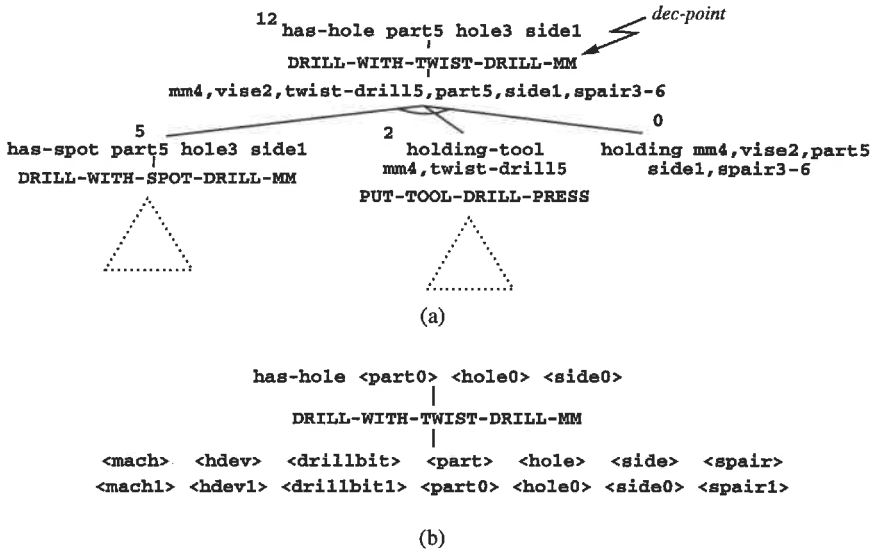


Figure 4.21: The first two steps of cktree construction in the example: (a) The relevant part of $plantree_A$ (cf Figure 4.17). (b) The first goal, operator, and binding cktree nodes built.

in its `applied` slot. As the side effects cause the operator to delete or achieve other goals, they may have an influence in the quality of the plan. Therefore the cktree should also capture those side effects. Step 5 bookkeeps them by calling **fill_side_effects** described in Section 4.4.2.3.

Creating a goal node

Build_ck_goal (Figure 4.22) is given a plan tree binding node b_q and a cktree binding node b_{ck} and creates a set of cktree goal nodes $prec_{ck}$. These cktree goal nodes correspond to the preconditions of b_{ck} and become the children of b_{ck} in the cktree. At planning time PRODIGY matches the operator precondition and expands it into a conjunction of literals. Preconditions universally quantified are expanded into multiple instantiated preconditions, one for each set of variable bindings. The plan tree node b_q has a child goal node $prec_q$ for each of those conjuncts. Each $prec_q$ is explored in turn by **build_ck_goal** (Step 3). A cktree goal node $prec_{ck}$ is created for each $prec_q$ except in the case of universally quantified preconditions. The new nodes $prec_{ck}$ correspond to parameterized preconditions p, obtained in Step 4 by the partial instantiation of the operator with the variable substitution σ stored in b_q. The above description of **build_ck_bindings** explained how σ is obtained. For each universally quantified precondition only one cktree node is created; *created_precs* keeps track of the preconditions already created to that purpose.

In order to make the use of the cktrees at planning time more efficient, the learner stores pointers from each variable used in the cktree to the cktree nodes where the variable is used (Step 10).

build_ck_goal(b_q, b_{ck}) ;; b_q is a plan-tree binding node; b_{ck} is a cktree binding node

1. $\sigma \leftarrow$ **name**(b_{ck}) ;; See Step 2 in Figure 4.20
2. *created_precs* $\leftarrow \emptyset$
3. for each $prec_q \in$ **children**(b_q)
4. $p \leftarrow$ **regress_prec_name**($prec_q$, b_q, σ) ;; parameterized precondition
5. if $p \notin$ *created_precs*
   then
6. *created_precs* $\leftarrow$ push(p,*created_precs*)
7. $prec_{ck} \leftarrow$ make_ck_goal_node :name p
8. :parent b_{ck}
9. **children**(b_{ck}) $\leftarrow$ push($prec_{ck}$,**children**(b_{ck}))
10. **store_var_pointers**($prec_{ck}$) ;; cf. Section 4.5.8
    else ;; it is a quantified precond already created
11. $prec_{ck} \leftarrow$ find $p \in$ **children**(b_{ck})
12. if **how_achieved**($prec_q$) $\notin$ {*:subgoaled,:initial-state*}
13. then **fill_how_achieved**($prec_q$, $prec_{ck}$) ;; Figure 4.25
14. if **achieves_too**($prec_q$) then **fill_achieves_too**($prec_q$, $prec_{ck}$) ;; Figure 4.24
15. if **deleted_by**($prec_q$) then **fill_deleted_by**($prec_q$, $prec_{ck}$) ;; Figure 4.26
16. **add_links_to_other_trees**($prec_q$, $prec_{ck}$) ;; Figure 4.32
17. if **children**($prec_q$) then **build_ck_op**($prec_q$, $prec_{ck}$)

Figure 4.22: Building a cktree goal node.

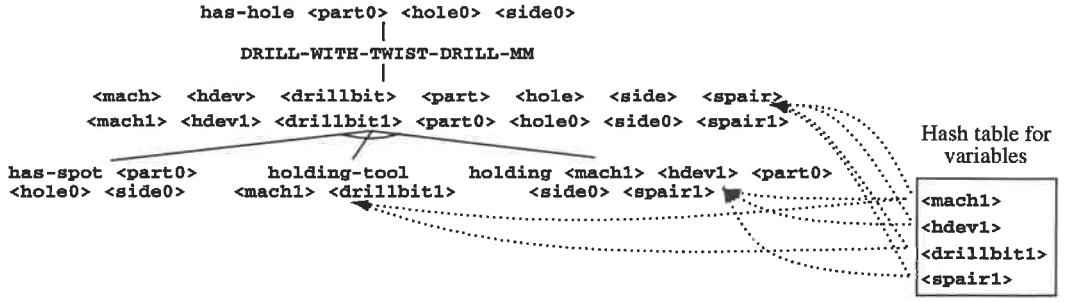


Figure 4.23: Building cktree goal nodes in the example. Dotted lines indicate the pointers from each variable to the nodes that use. Those pointers are stored in a hash table. Section 4.5.8 describes how those pointers are used.

The pointers are stored in a hash table. Entries are added as new variables are created when the cktree nodes are built. Entry contents are updated when new nodes are created that use those variables. Section 4.5.8 describes their use in detail.

To illustrate the steps just described Figure 4.23 shows the partial cktree for the example with one more level of nodes, corresponding to the preconditions of *drill-with-twist-drill-mm*. (Actually the nodes for the second and third precondition are built by separate calls to **build_ck_goal**.) The figure also shows the pointers from each variable introduced in the cktree to the nodes that use it.

The information about achievement and deletion that occurred at planning time was stored in the plan tree and is now added to the created cktree (Steps 12-16 of **build_ck_goal**). The next section describes this process in detail. Finally (Step 17) if the plan tree node $prec_q$ has any children, that is, it was achieved by subgoaling, **build_ck_op** is called to build the new cksubtree to record how that precondition was achieved.

4.4.2.3 Keeping Track of Achievements and Deletions

The achievement and deletion links are recorded in the cktree as it is being created using the experience stored in the plan tree. The functions responsible for this bookkeeping are called when a cktree goal or binding node is created. Figures 4.24 to 4.27 describe those functions. Several types of links are used to store achievement and deletion information in a cktree:

- `how_achieved`, in a cktree goal node g_{ck}, stores one or more pointers to (a) other cktree goal nodes that may correspond to the same goal (upon instantiation) and whose achievement could achieve g_{ck}, since it did in previous planning episodes, or (b) cktree binding nodes that may achieve g_{ck} as a positive side effect, since they did in previous planning episodes.

- `achieves_too`, in a cktree goal node g_{ck}, is the reverse link of `how_achieved`. It stores one or more pointers to other cktree goal nodes that may correspond to the same goal, and may be achieved by achieving g_{ck}, as they were in some previous planning episodes.

- `deleted_by`, in a cktree goal node g_{ck}, stores a list of pointers to cktree binding nodes that may delete g_{ck} as a side effect, since they did in some previous planning episodes.

- `applied`, in a cktree binding node b_{ck}, stores a list of pointers to cktree goal nodes that b_{ck} may add or delete as side effects, as it happened in some previous planning episodes. The `applied` link is a reverse of the `deleted_by` and `how-achieved` links that point to binding nodes.

Note that whether a cktree node actually adds or deletes another node depends on how the variables in the cktree nodes are instantiated when the cktree is used to provide guidance at planning time. The link is recorded when in some past planning experience the instantiation was such that the addition or deletion actually occurred.

fill_achieves_too($prec_q, prec_{ck}$)

1. $L \leftarrow <g_q, g_{ck}> \in$ *achievement_links* s.t. $g_q = prec_q\}$
2. if $L \neq \emptyset$
   then
3. for each $<prec_q, g_{ck}> \in L$;; it is <*achiever*,*achieved*>
4. **how_achieved**(g_{ck}) $\leftarrow prec_{ck}$
5. **achieves_too**($prec_{ck}$) $\leftarrow$ push(g_{ck},**achieves_too**($prec_{ck}$))
   else
6. for each $achieved_q \in$ **achieves_too**($prec_q$)
7. *achievement_links* $\leftarrow$ push(<$prec_{ck}$,*achieved*>,*achievement_links*)

Figure 4.24: Keeping track of achievement links. This function is called when building a cktree goal node $prec_{ck}$ for a plan tree node $prec_q$ that achieves other goals.

The above links are bookkept when a cktree goal node $prec_{ck}$ is being created, and achievement or deletion information for the current planning episode had been stored in the corresponding plan tree goal node $prec_q$ (Steps 12-15 in Figure 4.22). The links in the plan tree have the same names enumerated above. The bookkeeping is the task of **fill_achieves_too**, **fill_how_achieved**, and **fill_deleted_by**. The three functions work in similar ways. (Figures 4.24 to 4.27 give further details on each function). Assume that $prec_{ck}$ is the cktree goal node being created, and that $prec_q$ and $prec'_q$ are linked in the plan tree by one of the listed achievement links. The purpose of the bookkeeping functions is to link in the same way the corresponding nodes $prec_{ck}$ and $prec'_{ck}$ in the cktree. Note however that $prec'_{ck}$ may have not been created yet, and thus the link creation must be postponed. Thus two cases are possible when $prec_{ck}$ is being built:

- If node $prec'_{ck}$ has not been created yet, the pair $<prec_{ck}, prec'_q>$ is stored in some bookkeeping global variable so that whenever $prec'_{ck}$ is created the link is set.
- If node $prec'_{ck}$ already exists, a pair of the form $<prec'_{ck}, prec_q>$ is in the corresponding bookkeeping global variable and can be accessed by looking for $prec_q$. Then the link between $prec_{ck}$ and $prec'_{ck}$ is added.

Additionally, a fourth bookkeeping function, **fill_side_effects** (Figure 4.27), is called when a cktree binding node b_{ck} is being created and the corresponding plan tree node b_q had side effects (Step 5 of Figure 4.20). Each side effect of b_q, is translated into an element of the form $<p_{ck}, type, effect>$ that is stored in the `applied` slot of b_{ck}. p_{ck} is the cktree node corresponding to the side effect added or deleted, $type$ indicates whether it was an add or a delete, and *effect* indicates which of the operator schema effects is the side effect. **effect_in_op** computes the effect (Steps 4 and 7 of Figure 4.27).

fill_how_achieved($prec_q$, $prec_{ck}$)

1. $g_{ck} \leftarrow$ find g_{ck} s.t. $<g_{ck}, prec_q> \in$ *achievement_links*
2. if $g_{ck} \neq \emptyset$
   then
3. **how_achieved**($prec_{ck}$) $\leftarrow g_{ck}$
4. **achieves_too**(g_{ck}) $\leftarrow$ push($prec_{ck}$,**achieves_too**(g_{ck}))
5. **add_bnds_constraints**($prec_{ck}, g_{ck}$) ;; Section 4.4.2.4
   else
6. $achiever_q \leftarrow$ **how_achieved**($prec_q$)
7. *achievement_links* $\leftarrow$ push($<achiever_q$,$prec_{ck}>$,*achievement_links*)
8. if **side_effect_links_p**(**how_achieved**($prec_q$)) ;; now link side effects with operators
   then
9. *binding_node* $\leftarrow$ **how_achieved**($prec_q$) ;; search node that added $prec_q$ as a side effect
10. $b_{ck} \leftarrow$ find b_{ck} s.t. $<$*binding_node*,$b_{ck}>\in$ *side_effect_links*
11. if $b_{ck} \neq \emptyset$
    then
12. **how_achieved**($prec_{ck}$) $\leftarrow b_{ck}$
13. **applied**(b_{ck}) $\leftarrow$ push($<prec_{ck}$,`:pos`,**effect_in_op**($prec_q$)$>$,**applied**(b_{ck}))
    else
14. *side_effect_links* $\leftarrow$ push($<$*binding_node*,$prec_{ck}$, $p_q>$,*side_effect_links*)

Figure 4.25: Keeping track of achievement links. This function is called when building a cktree goal node $prec_{ck}$ for a plan tree node $prec_q$ that was achieved by other node.

Two bookkeeping global variables are maintained and used by the functions described above:

- *achievement_links* bookkeeps information to build the `how_achieved/achieves_too` links. Each element of *achievement_links* is a pair $< achiever, achieved >$ of nodes. Given a call to **build_ck_goal**($prec_q$, $prec_{ck}$):
  - if **how_achieved**($prec_q$)=$achiever_q$ and the cktree node corresponding to $achiever_q$ has not been created yet, an element $<achiever_q, prec_{ck}>$ is added to *achievement_links*.
  - if **achieves_too**($prec_q$)=$achieved_q$ and the cktree node corresponding to $achieved_q$ has not been created yet, an element $<prec_{ck}$,$achieved_q>$ is added to *achievement_links*.

  These pairs are added respectively by **fill_how_achieved** (Steps 6-7) and **fill_achieves_too** (Steps 6-7), and are used by those two same functions to set the links when the cktree

fill_deleted_by($prec_q$, $prec_{ck}$)

1. *binding_node* ← **deleted_by**($prec_q$) ;; search node that deleted $prec_q$ as a side effect
2. b_{ck} ← find b_{ck} s.t. <*binding_node*,b_{ck}>∈ *side_effect_links*
3. if $b_{ck} \neq \emptyset$
   then
4. **deleted_by**($prec_{ck}$) ← push(b_{ck},**deleted_by**($prec_{ck}$))
5. **applied**(b_{ck}) ← push(<$prec_{ck}$,`:neg`,**effect_in_op**($prec_q$)>,**applied**(b_{ck})
   else
6. *side_effect_links* ← push(<*binding_node*,$prec_{ck}$,$prec_q$>,*side_effect_links*)

Figure 4.26: Keeping track of deletion links. This function is called when building a cktree goal node $prec_{ck}$ for a plan tree node $prec_q$ that was deleted by other node.

fill_side_effects(b_q, b_{ck})

1. *binding_node* ← **applied**(b_q) ;; Search node corresponding to b_q
1. for each <*binding_node*,p_{ck}, p_q>∈ *side_effect_links*
2. if **deleted_by**(p_q) ≠ ∅
   then ;; adding a negative side effect
3. **deleted_by**(p_{ck}) ← push(b_{ck},**deleted_by**(p_{ck}))
4. **applied**(b_{ck}) ← push(<p_{ck},`:neg`,**effect_in_op**(p_q)>,**applied**(b_{ck}))
5. if **how_achieved**(p_q) ≠ ∅
   then ;; adding a positive side effect
6. **how_achieved**(p_{ck}) ← b_{ck}
7. **applied**(b_{ck}) ← push(<p_{ck},`:pos`,**effect_in_op**(p_q)>,**applied**(b_{ck}))
8. *side_effect_links* ← push(<*binding_node*,b_{ck}>,*side_effect_links*)

Figure 4.27: Keeping track of side effects. This function is called when building a binding node b_{ck} corresponding to a plan tree node b_q that had side effects.

nodes are created.

- *side_effect_links* bookkeeps information to build the side effect links, that is links stored in `how_achieved/applied` or `deleted_by/applied` slots. It has two kinds of elements:
  - <*binding_node*,p_{ck}, p_q>: added by **fill_how_achieved** and **fill_deleted_by** when a cktree goal node p_{ck} is being created, and the corresponding plan tree node p_q was added or deleted as a side effect of *binding_node*. The element is added if the corresponding plan tree binding node has not been created yet. When it is created,

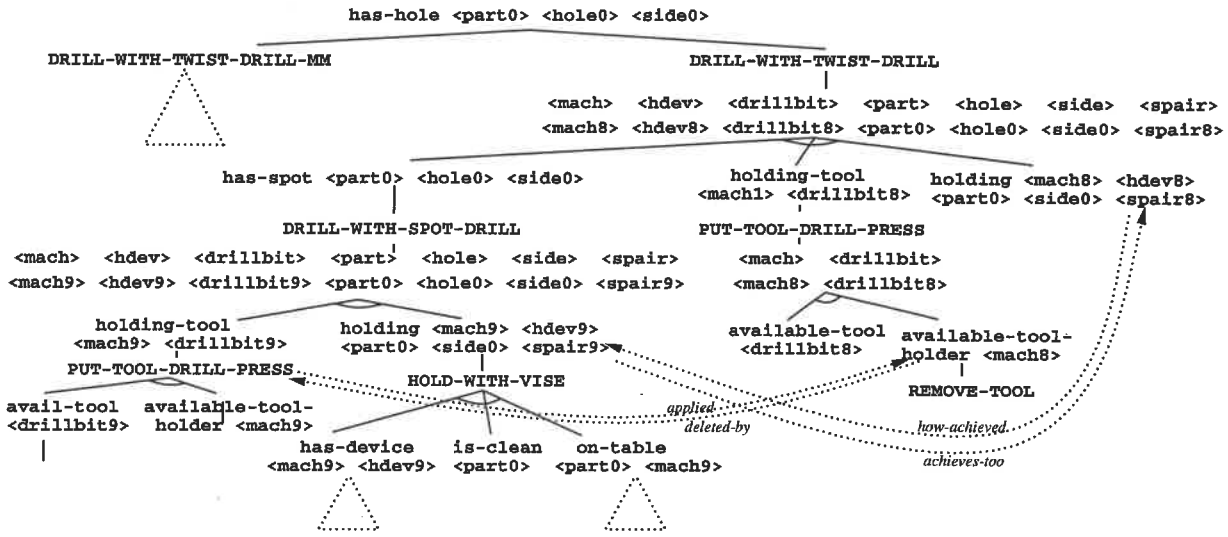


Figure 4.28: The part of the cktree built in the example from $plantree_B$ (top of Figure 4.17) showing the achievement and deletion links created.

the link will be set by **fill_side_effects**.

- <*binding_node*,b_{ck}>: added by **fill_side_effects** when a cktree binding node b_{ck} is being created and its corresponding plan tree node b_q had positive or negative side effects. It will be used by **fill_how_achieved** and **fill_deleted_by** to set the side effect links when the cktree goal nodes for the side effects are being built.

In both cases *binding_node* corresponds to a node in the planner's search trace where an operator was applied.

There is an additional bookkeeping step, **add_links_to_other_trees** (Step 16 of Figure 4.22), that keeps track of the achievement links that point to nodes corresponding to other cktrees. It will be described in Section 4.4.4.

Figure 4.28 shows the achievement and deletion links added to the cktree when learning from $plantree_B$ in the example (top of Figure 4.17). Note that both in the plan trees and the cktrees the achievement and deletion links are bidirectional, as described above. The `deleted_by` link was built in two steps: first, when the node for *put-tool-drill-press* was created, the link from it to goal *is-available-tool-holder* (an `applied` link) was bookkept in *side_effect_links* as the goal node had not been created yet; second, when the goal node for *is-available-tool-holder* was created, the `applied` and `deleted_by` links were added to the cktree. The `how_achieved` and `achieves_too` links were built in a similar way.

4.4.2.4 Learning Constraints on Bindings

The cktrees store additional information that is useful to speed up their use at problem solving time. This information has the form of constraints on the values that operator variables may take and it is stored at the time in which achievement links are added (Step 5 of Figure 4.25). If g_1 achieves g_2, a set of constraints is introduced so the arguments of g_1 and g_2 are the same. The constraints are stored in the cktree binding nodes that introduce the constrained variables, and will be used to prune out bindings for that operator that would lead to lower quality solutions. The purpose of these constraints is only to make the use cktrees more efficient. How these constraints are used will be explained in Section 4.5.4.

The example in Figure 4.28 serves to illustrate the binding constraints created when an achievement link is added. In the plan trees for the example (Figure 4.17), the cost of the `holding` precondition of *drill-with-twist-drill* was 0 because it was achieved as a precondition of *drill-with-spot-drill*. This is captured by the achievement link in the figure. Therefore the choice of bindings for *drill-with-twist-drill* constrains the choice of bindings for *drill-with-spot-drill*, so that the two `holding` preconditions are instantiated in the same way. Consequently when the achievement link is added to the cktree, constraints on the variables introduced by operator *drill-with-spot-drill* are stored at its binding node: the machine `<mach9>` is constrained to have the value of `<mach8>`, the holding device `<hdev9>` is constrained to be the same as `<hdev8>`, and the orientation of the part `<spair9>` should be the same as `<spair8>`. The purpose of the constraints is to focus the planner on bindings that lead to better quality solutions, in this case to save steps in holding the part.

4.4.3 Updating an Existing Control-Knowledge Tree

Figure 4.14 described the top-level call to the cktree learning mechanism. We discussed how the learner makes a distinction depending on whether a cktree already exists to guide the current decision. Let *dec_point* represent again a control knowledge gap, that is a decision for which the planner failed to make the choice that leads to the better plan. If a cktree relevant to that decision exists, it has failed to suggest the desired alternative when used to guide the search at planning time. Therefore the learner needs to update the existing cktree(s) with the current planning episode (Step 6 of Figure 4.14).

To update a cktree the learner simultaneously traverses the cktree and the relevant parts of the plan trees for the current planning episode. Whenever a subtree that appears in the plan tree, does not have a match in the cktree, the cktree is extended. The extension is built by translating and generalizing the subtree in a similar way as when a new cktree is created altogether, as described in Section 4.4.2. Consistency is ensured by the way the new knowledge is added to the existing cktree: the process consists of adding new subtrees under appropriate nodes or adding new achievement links in order to capture the planning experience of the current

learn_update_cktree($plantree_A$,$plantree_B$,*dec_point*,*ckroot*)

1. $q_A \leftarrow$ **relevant_subtree**($plantree_A$,*dec_point*)
2. **match_and_learn**(*ckroot*,q_A)
3. **learn_other_cktree**($plantree_A$) ;; Figure 4.33
4. $q_B \leftarrow$ **relevant_subtree**($plantree_B$,*dec_point*)
5. **match_and_learn**(*ckroot*,q_B)
6. **learn_other_cktree**($plantree_B$)

Figure 4.29: Updating an existing cktree. Note the similarity with Figure 4.18.

episode. Those new subtrees or links may represent alternative additional ways to achieve a goal different from the ones that the cktree knew about. Thus the new knowledge is added to the existing one without invalidating it. Figure 4.29 shows **learn_update_cktree**, the function that updates the cktree.[11]

To illustrate how an existing cktree is updated with a new planning episode, assume that, after learning from the previous example the cktree in Figure 4.28, a new problem is presented to the planner and the existing cktree is not able to suggest the correct alternative (*drill-with-twist-drill-mm* again). The new problem is summarized in Figure 4.30(a). Again the goal is to drill a hole, but in the initial state for the new problem there is only one holding device, `vise2`, which is on the milling machine, and the drill press is holding the spot-drill tool. Figure 4.30(b) shows the plan tree for the lower quality solution, $plantree_B$.

The existing cktree does not contain information to estimate the cost of achieving `has-device` since it was true initially in the first example (top of Figure 4.17). Therefore the cktree is updated with the experience of this planning episode by adding a subtree rooted at `has-device`. To find that point the learner traverses simultaneously the `has-hole` cktree and $plantree_B$ (Figure 4.30(b)) and identifies node `has-device` as a place where knowledge can be updated. Figure 4.31 shows the updated cktree.

4.4.4 Learn and Update Other Cktrees

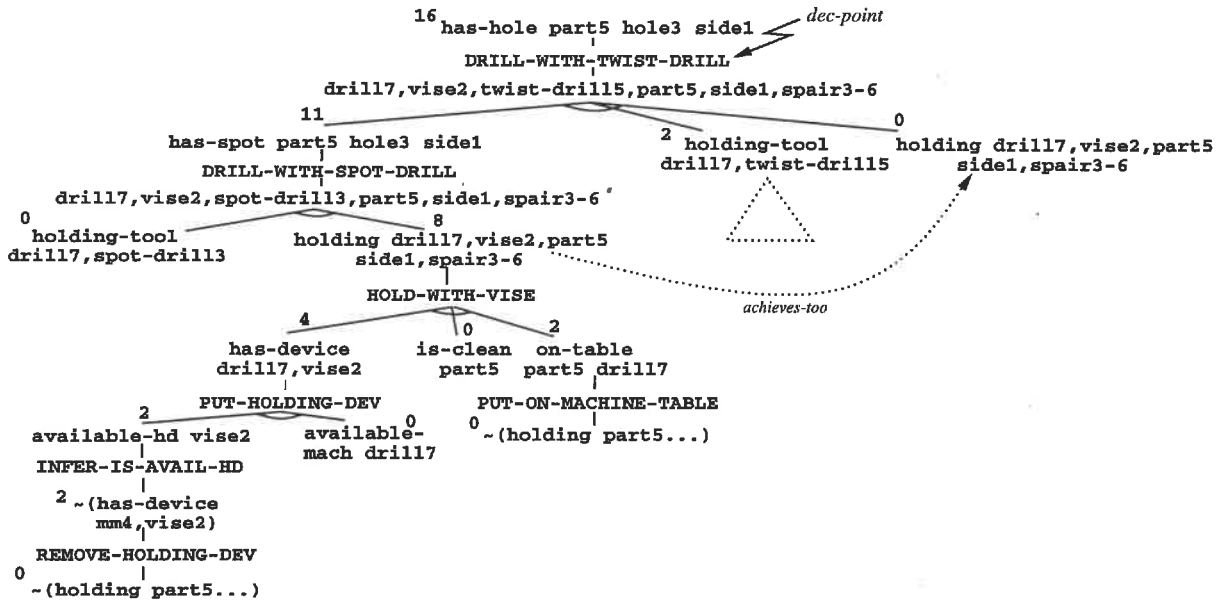
When several goals in the problem are not independent, there exist achievement and deletion links that connect the plan trees corresponding to those goals. See for example the bottom plan tree in Figure 3.25. Therefore estimating the cost of an alternative operator or binding

[11] In the implementation of **learn_update_cktree** the cktree is matched as it would be at planning time (Steps 2 and 5). The purpose of this preprocessing is to aid the bookkeeping described in Section 4.4.2.3 so the appropriate achievement and deletion links are set between the existing nodes and the nodes that are being created.

```
(objects
;;machines
 (object-is mm4 MILLING-MACHINE)
 (object-is drill7 DRILL)

;;holding devices
 (object-is vise2 VISE)

;;parts and holes
 (object-is part5 PART)
 (object-is hole1 HOLE)

;;tools
 (object-is spot-drill3 spot-drill7 SPOT-DRILL)
 (object-is twist-drill5 TWIST-DRILL)
 (object-is tap4 tap)
 ...
 (object-is brush1 BRUSH)
 (object-is soluble-oil SOLUBLE-OIL)
 (object-is mineral-oil MINERAL-OIL))
```

```
(state (and (diameter-of-drill-bit twist-drill5 9/64)
            (diameter-of-drill-bit tap4 9/64)
            (is-clean part5)
            ...
            (holding-tool drill7 spot-drill3)
            (has-device mm4 vise2)
            (on-table mm4 part5)))

(goal (has-hole part5 hole1 side1 0.3 9/64 1.375 0.25))
```

(a)



(b)

Figure 4.30: (a) Initial state and goal of a new example problem. (b) The plan tree corresponding to the lower quality solution. Some parts have been omitted for clarity.

for a goal may require considering not only the cktree for the goal at the decision point, but also cktrees corresponding to other problem goals if there are achievement and deletion links among those cktrees. This section briefly describes how the learner builds the relevant cktrees

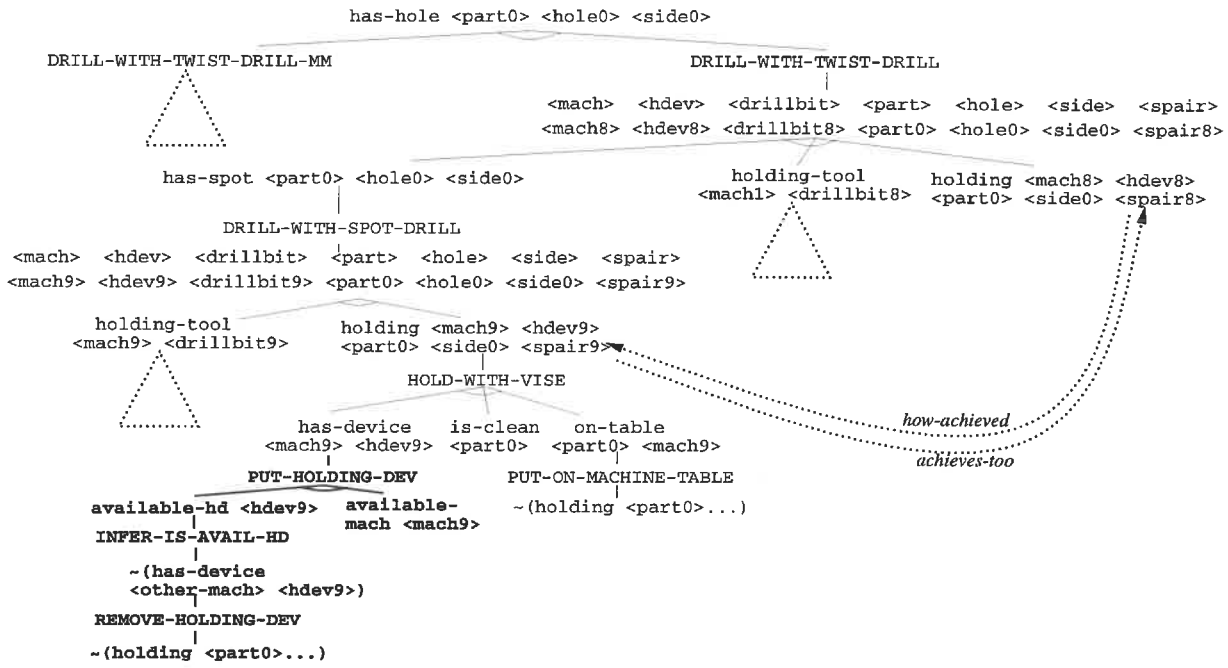


Figure 4.31: The cktree of Figure 4.28 updated with the new planning episode described in Figure 4.30. The cktree existing prior to learning from this episode is indicated with a lighter font.

for other goals. Section 4.5.9 will describe how the cktree matcher uses the cktrees for other goals.

---

add_links_to_other_trees($prec_q, prec_{ck}$) ;; *other_tree_nodes* is a global variable

1. for each $achieved_q \in$ **achieves_too**($prec_q$) s.t. **in_other_tree_p**($prec_q$,$achieved_q$)
2. $\quad$ *other_tree_nodes* $\leftarrow$ push($achieved_q$,*other_tree_nodes*)
3. $achiever_q \leftarrow$ **how_achieved**($prec_q$)
4. if $achiever_q \neq \emptyset \wedge$ **in_other_tree_p**($prec_q$,$achiever_q$)
5. then *other_tree_nodes* $\leftarrow$ push($achiever_q$,*other_tree_nodes*)

Figure 4.32: Keeping track of achievement links that point to other trees.

When a cktree goal node is being built for a plan tree goal node, bookkeeping of the achievement and deletion links is performed (Figure 4.22). The bookkeeping includes recording (Step 16) the achievement links that point to subtrees of the plan tree that correspond to other goals. Figure 4.32 describes this bookkeeping. The achieved nodes which would correspond to other cktrees are stored in a global variable *other_tree_nodes*.

Then after the cktree is built for the first goal, *other_tree_nodes* is used by **learn_other_cktree**

(Figure 4.33) to build other cktrees, in a way similar to the first cktree (cf Figures 4.14, 4.18 and 4.29). First the top nodes of the plan trees for the interacting goals are found, that is, nodes that (a) are ancestors of nodes in *other_tree_nodes* and (b) correspond to top-level goals. If no cktree exists yet for that goal, a new one is built by traversing the plantree (Steps 5-7). If a relevant cktree exists, it is updated with the current episode (Step 8). We will not elaborate further on this, since the process is very similar to the one described in previous sections.

---

learn_other_cktree($plantree$)

1. *roots* ← {**relevant_subtree**(*plantree*,q) s.t. $q \in$ *other_tree_nodes*} ;; cf Figure 4.32
2. for each $g_{root} \in$ *roots*
3. *ckroot* ← **relevant_cktree**(g_{root},*cktrees*) ;; *cktrees* is a global variable
4. if *ckroot* = $\emptyset$;; cf Figure 4.14
   then ;; cf **learn_new_cktree** in Figure 4.18
5. *ckroot* ← make_ck_goal_node :name **parameterize**(g_{root})
6. **build_ck_op**(g_{root},*ckroot*)
7. *cktrees* ← push(*ckroot*,*cktrees*)
   else ;; cf **learn_update_cktree** in Figure 4.29
8. **match_and_learn**(*ckroot*,g_{root})

Figure 4.33: Learning and updating other cktrees.

Figure 4.34 shows the plan tree for the better solution to a problem to achieve goals g_1 and g_2. The relevant decision point in the problem, for which learning is invoked, was the operator choice to achieve g_1. For our purposes it is enough to show the plan tree for one operator alternative, op_1^B. The learner builds first the cktree for g_1, the relevant top-level goal for the decision point, as described in Section 4.4.2. In the process it keeps track of relevant nodes in the subtrees for other goals, as indicated by the achievement and deletion links of the plan trees, namely g_{22} and g_{211}. The rational is that achieving g_{22} and applying operator op_{11} have effects in the cost of goal g_2, and therefore of the whole plan. Then it proceeds to learn (in this case build anew) a cktree for the other goal, g_2, the top-level goal that is an ancestor of the two nodes g_{22} and g_{211}. Figure 4.35 partially shows the cktrees built from the plan tree.

4.5 Using Control Knowledge Trees

Section 4.4 explained how cktrees are learned from a planning episode. This section describes how the cktrees are used at planning time to guide the planner towards good plans. We first overview the process. Then we present the cktree matching algorithms in detail and illustrate them with some examples.

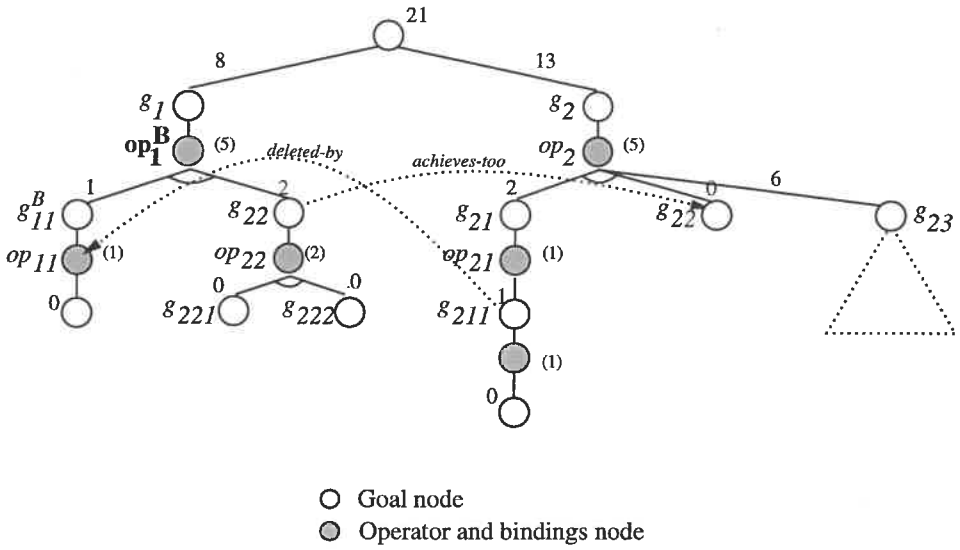


Figure 4.34: A plan tree for a plan to solve goals g_1 and g_2. The decision point in the planning episode for which learning occurs is the operator to achieve g_1. Only the plan tree corresponding to the better alternative, op_1^B, is shown in the figure. A number in brackets next to an operator is the cost of the operator given the quality metric. A number without brackets next to a goal node is the cost of the subtree rooted at that node. Operator and binding nodes have been merged together in the figure for clarity.

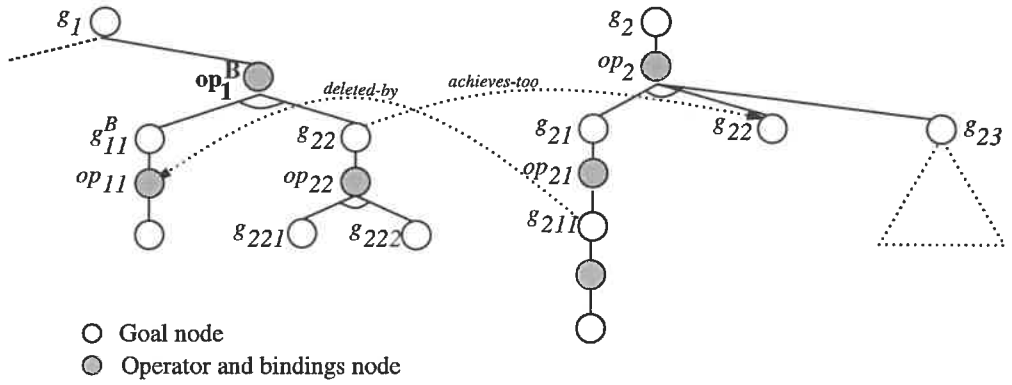


Figure 4.35: Two cktrees for goals g_1 and g_2 learned from the plan tree in Figure 4.34.

4.5.1 Overview of Control Knowledge Matching

When the planner is making an operator decision or a bindings decision, it checks whether control knowledge in the form of a cktree that may guide that decision is available. If so, the cktree is used to estimate the cost of each of the alternatives. We term this process *cktree matching*. Note that we use the term *estimate* since the cktrees capture previous planning

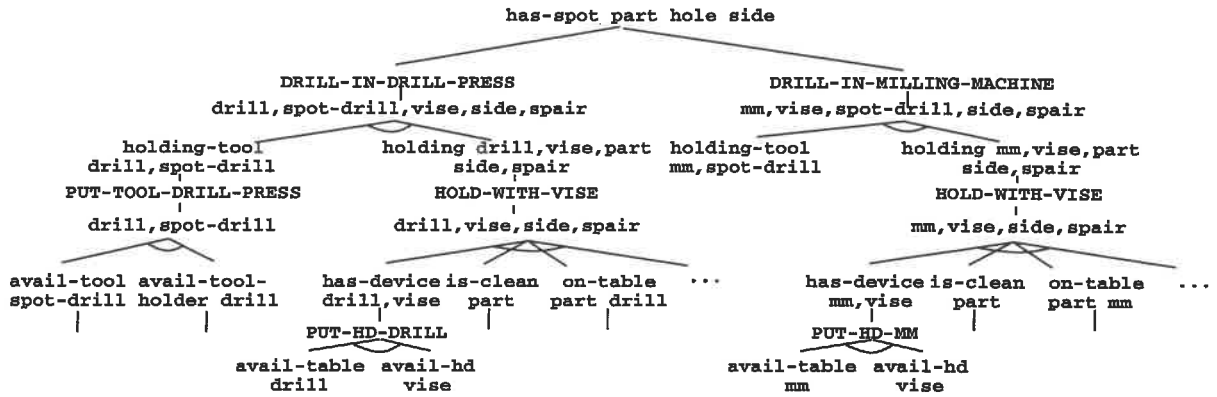


Figure 4.36: The control knowledge tree learned from the plan trees for the problem in Figures 4.8 and 4.9. Goal and operator arguments are variables (<> have been omitted for space purposes).

experience and they may have incomplete information to determine exactly the quality of the alternatives. Figure 3.9 described how the cost of a plan tree is computed by traversing the plan tree. Similarly a cktree can be traversed and provide an estimate of the cost of an alternative. Since the cktree is parameterized, its variables need to be instantiated for the particular problem being solved by providing bindings for each cktree binding node visited. As there may be several ways to instantiate an operator, several alternative bindings are tried to choose those that would lead to a better plan.

Before describing the cktree matching algorithms in detail, we will use some examples in the process planning domain to explain how the cktrees are exploited to provide guidance for search decisions. The first example was introduced in Section 4.2. Figure 4.9 (a) showed the two plan trees available after solving a simple problem in which the goal is to drill a spot hole on a part (the root of the plan trees) and the *tool is initially set on the available milling machine* (note the 0-cost of `holding-tool` in the second plan tree). Figure 4.9 (b) partially showed the quality metric. The plan that uses the milling machine (plan tree on the right) has slightly better quality as the higher cost of operator *drill-in-milling-machine* is overcome by the savings of having the tool set already. Note that if either of the preconditions of *put-tool-drill-press* had cost 0, the two alternative drill operators would lead to plans of cost 13. Examples of this situation would occur if *drill7*'s tool-holder were free, or if there were a free spot-drill different from *spot-drill3*. Thus seemingly small differences in the state may lead to prefer different alternatives at the choice point and therefore generate different plans. Capturing this kind of knowledge with control rules turned out to be difficult and motivated the development of control knowledge trees.

Figure 4.36 depicts the cktree learned form this problem using the algorithms described in Section 4.4. The learned cktree can be used to guide planning for a new problem. To estimate the cost of achieving a goal using a particular operator alternative, the cktree matcher adds the cost of the instantiated operator, given by the quality metric, to the cost of achieving the operator

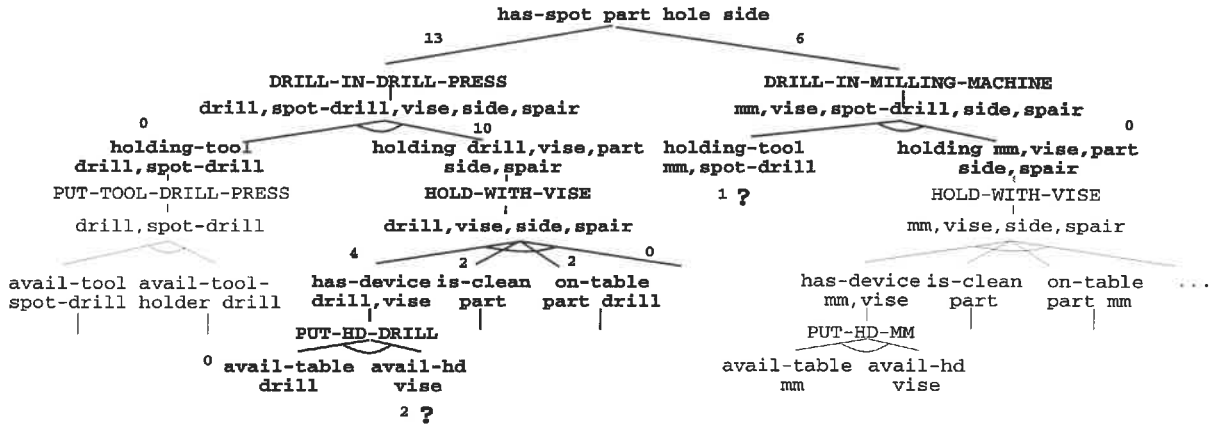


Figure 4.37: Using the cktree previously learned to solve a new problem in which the *the part is set on the milling machine* and *the tool is set on the drill press*. A lighter font is used in parts of the cktree that are not explored by the cktree matcher. **?** indicates a default value assigned to a goal node when no knowledge was available to estimate its cost.

preconditions. The operator is instantiated as it would at planning time, that is, with bindings for the relevant operator effect coming from the goal, and with the problem objects. The cost of achieving each precondition is recursively computed in turn. This process corresponds to traversing the cktree. The traversal stops at goal nodes that have estimated cost 0 because in the current problem they are true in the initial state or added (as indicated by the achievement links), and are not deleted (as indicated by deletion links). The matcher keeps track of the added and deleted nodes. The traversal also stops at the cktree leaves, in which case the matcher has no knowledge to estimate the cost of achieving them. A default value is assigned, which is the minimum cost of achieving that goal by just applying a relevant operator.

To illustrate this process, we now describe how the learned cktree in Figure 4.36 can be used to guide planning for a new problem in which the goal is also to drill a spot hole. For example, assume the planner is asked to solve a second problem in which in the initial state *the part is set on the milling machine* and *the tool is set on the drill press*. When the planner must make a choice of operator to achieve the goal `has-spot`, it uses the cktrees as control knowledge. Here we sketch how the cktree is used to estimate the cost of each drilling alternative. Figure 4.37 shows the cktree matching process and result. The matcher explores the two operator alternatives in the `has-spot` cktree, drilling in the drill press and drilling in the milling machine. For explanation purposes only assume the latter alternative is considered first. It corresponds to the right subtree. According to the quality metric, the cost of the operator is 5. Then the matcher estimates the cost of each of the subgoals. Recall that in the current problem the part is being held in the milling machine. Therefore the cost of `holding` is 0 and the matcher does not need to explore further that subtree. Unexplored parts of the cktree are shown with a lighter font in the figure. When estimating the cost of the `holding-tool` subgoal, no further

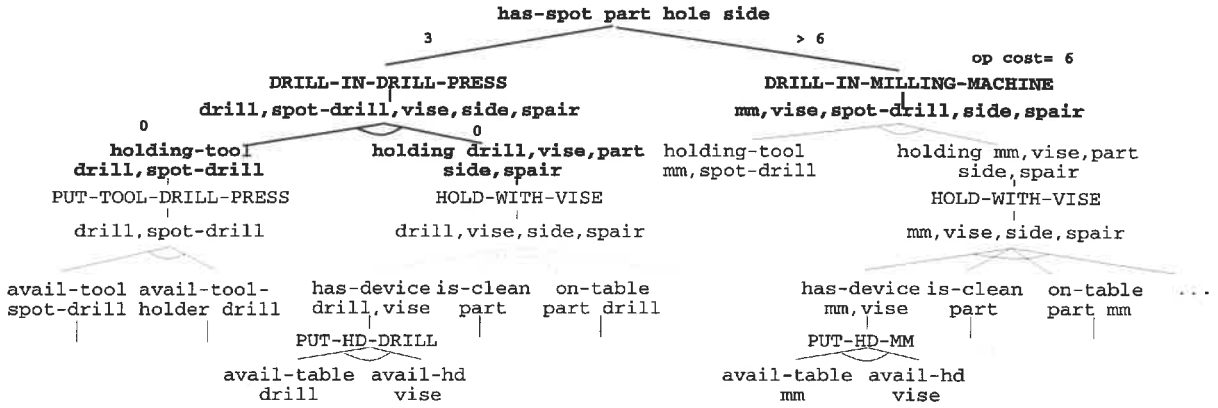


Figure 4.38: Using the cktree previously learned to solve a new problem in which the *the part and the tool are set on the drill press.* The cktree is matched efficiently because the matcher stops before exploring the *drill-in-milling-machine* subtree. A lighter font is used to depict parts of the cktree that are not explored by the cktree matcher.

information is available in the cktree on how to achieve that subgoal, as in the problem from which the cktrees were learned the tool was being held in the milling machine and the cost of that subgoal was 0. The matcher gives a default value of 1 (the minimum cost of putting a tool in the machine according to the quality metric) which is propagated up. Therefore the cost of drilling in the milling machine is estimated as 6 (5 for the operator, plus 0 and 1 for the operator preconditions). The matcher now proceeds to estimate the cost of the drill press alternative, which turns out to be more expensive (13). Therefore the planner chooses the milling machine operator and continues planning. The actual cost of the final plan that uses the milling machine operator is 7 (since the real cost of `holding-tool` is 2 as the tool has to be released from the drill press before holding it). The actual cost of the plan that uses the drill press operator is 15 as the real cost of making the vise available is 4 (it is holding the part in the milling machine) instead of the estimated 2. In spite of limited knowledge the matcher was able to suggest an adequate choice.

Consider now a third problem with the same goal in which *both the part and the tool are set initially on the drill press.* Figure 4.38 illustrates the matching process in this case using the cktree learned from the first problem. The planner calls again the cktree matcher for guidance at the operator choice for `has-spot`. The matcher starts by exploring the drill press alternative, which has cost 3 (as its subgoals have cost 0). Then it turns to the milling machine alternative. The cost of the operator itself (drill in the milling machine) is 5, larger than 3. Therefore there is no need to proceed to estimate the subgoal cost. Matching stops and drilling in the drill press is suggested. (Note that the correct choice in this example is different from that in the previous ones.)

This example also illustrates another point. In some cases there is more than one way to

instantiate an operator. Even if there is only one tool and one drill press available, the drill operator can be instantiated in two different ways depending on the part orientation, that is, on the choice of sides facing the holding device (`spair` in the cktree of Figure 4.36). The best instantiation is the one whose `holding` precondition matches the way the part is being held in the state and thus has cost 0. Otherwise the part would have to be released and held again. The cktree matcher finds the good instantiation and records it. That information is used when the planner, in the next decision point after the operator choice of the planning algorithm, must choose bindings for the drill operator. In this way the matcher records alternatives that were suggested by the cktree traversal process as guidance for choices that will be made later on in planning.

The examples just used to introduce the use of cktrees are simple one-goal problems and the cktree contains no achievement (other than subgoaling) and deletion side effect links. When those links are present the cktree matcher is able to follow them and keep track of the added and deleted goals. When the problem has multiple goals and the matcher is finding an alternative for a given one, more than one cktree may be considered and so goal interactions are captured. Only the relevant parts of other cktrees are explored. The next sections describe in detail the cktree matching algorithm.

4.5.2 Calling the Cktree Matcher

The invocation of the cktree matcher is done through a number of PRODIGY4.0's operator and bindings preference control rules. We have implemented several meta-predicates that are used in those rules:

- `(current-goal-and-pref-op <op>)`: calls the cktree matcher for the current goal. The matcher finds the estimated best alternative operator and binds `<op>` to it. As a side effect it stores suggestions for future search decisions.
- `(op-suggested-for-current-goal <op>)`: check if a suggestion for the current operator decision is available from previous exploration of cktrees. If so, `<op>` is bound to the suggested operator.
- `(suggested-bindings <op> <bnds>)`: binds `<op>` to the current operator and checks if a suggestion for the current binding decision is available from previous exploration of cktrees. If so, `<bnds>` is bound to the suggested bindings.

Figure 4.39 (a) shows the control rule that fires the cktree matching at a goal node when an operator decision must be made. Control rule preconditions are tested in the order in which they appear in the rule. The rule first tests `op-suggested-for-current-goal` to check if there is an available suggestion (from previous exploration of cktree(s)). If there is not, it uses the

cktrees to provide guidance by testing `current-goal-and-pref-op`. In the case of a bindings decision, that is, at an operator node, the control rule in Figure 4.39 (b) is tested to find some suggested bindings from previous cktree exploration.

```
(control-rule use-cktrees
  (if (or (op-suggested-for-current-goal <op>)
          (current-goal-and-pref-op <op>)))
  (then prefer operator <op> <other-op>))
```

(a)

```
(control-rule use-suggested-bindings
  (if (and (current-operator <op>)
           (suggested-bindings <op> <bnds>)))
  (then prefer bindings <bnds> <other-bnds>))
```

(b)

Figure 4.39: Control rules that invoke the control knowledge stored in the cktrees.

As an implementation detail note that the planner only fires operator preference rules, in particular the rule in Figure 4.39(a), if more than operator is a candidate to achieve the current goal. However more than one binding set may be candidate for that operator and the cktrees are used to guide that binding choice. Therefore rules like the one in Figure 4.40 are constructed automatically when the cktree is learned for a binding decision. These rules only fire if there are not `suggested-bindings`, that is, if the operator control rule has not matched the cktree(s) and stored some suggested bindings. The rule preconditions are matched in sequence and PRODIGY4.0's syntax requires that the bindings rules specify the current operator (`tap` in the example) instead of a variable that gets bound to it at matching time. The test of `current-goal-and-pref-op` stores as a side effect bindings suggestions and the final test of `newly-suggested-bindings` binds `<bnds>` to those suggestions.

```
(control-rule bnds-for-tap
 (if (and (current-operator TAP)
          (not (suggested-bindings TAP <bnds>))
          (current-goal-and-pref-op
           (is-tapped <part> <hole> <side> <hole-depth> <hole-diameter>
                      <loc-x> <loc-y>)
           TAP)
          (newly-suggested-bindings TAP <bnds>)))
 (then prefer bindings <bnds> <other-bnds>))
```

Figure 4.40: A control rule that invokes the cktree matcher. This rule was built automatically when the cktree for `is-tapped` was learned.

current-goal-and-pref-op

1. $g \leftarrow$ **current_goal**
2. $ckroot \leftarrow$ **relevant_cktree**$(g, cktrees)$;; cf Figure 4.14
3. $\beta_0 \leftarrow$ **bnds**(g,**name**($ckroot$))
4. for each $op \in$ **children**($ckroot$)
5. $<alt_0, cost_0> \leftarrow$ **match_cktree**(op, β_0, β_0) ;; Figure 4.42
7. $<alt_{best}, cost_{best}> \leftarrow$ **match_all_alts**$(op, \beta_0, alt_0, cost_0)$;; Figure 4.46
9. **store_prefs_for_subgoals**(alt_{best}) ;; See Section 4.5.7.1
11. return(**choice**(alt_{best}))

Figure 4.41: Definition of meta-predicate `current-goal-and-pref-op`. The variable in the argument of the meta-predicate (see Figure 4.39(a)) gets bound to the meta-predicate's result (Step 11). Steps 1-2 finds the relevant cktree for the current goal and Steps 3-7 explore the cktree to find the best alternative. In the process the matcher may suggest alternatives for subgoals of the current goal that will be used later by the planner. Those suggestions are stored in Step 9. This definition of `current-goal-and-pref-op` will be completed in Section 4.5.9.2. For clarity we have kept the same step numbering as in the complete definition.

4.5.3 Cktree Matching as Traversing the Cktree

Figure 4.41 shows a first version of the definition of the `current-goal-and-pref-op` meta-predicate. The complete definition will be introduced in Section 4.5.9.2. The matcher starts by finding an existing cktree relevant to the current goal. The initial set of bindings β_0 for the cktree variables is formed by matching the goal at the cktree root with the current planning goal.

The matcher traverses the cktree, which is an and/or tree. If more than one alternative operator is available for a subgoal, that is, if the node corresponding to the subgoal has more than one child, all of them are tested to find the best alternative. Similarly if an operator variables can be instantiated in different ways for the current problem, those alternative bindings are explored. At each point during matching the current alternative is being maintained. It contains bindings for the cktree variables in the nodes that the matcher has explored already, and the operator being considered for each subgoal if there are more than one. The remaining alternatives are stored as the cktree is traversed so they are tested later on. The current alternative initially contains the bindings β_0 that come from the current goal, and is completed as the cktree is traversed.

The matcher is called for each of the operators relevant to the current goal which are stored in the cktree as children of the root. These are a subset of the relevant operators, those that the learner has seen in the past and has stored in the cktree. The matcher starts by generating and

testing the first alternative for the first operator. This corresponds to the call to **match_cktree** in Step 5, which returns the complete first alternative alt_0 and its cost $cost_0$. $cost_0$ becomes a cost threshold as further alternatives are explored. Then the remaining alternatives recorded during the cktree exploration are tried (Step 7). Finally the best operator according to the estimated costs is returned. The complete alternative alt_{best} is stored by **store_prefs_for_subgoals** (Step 9) in a global variable *preferred_alts_for_subgoals* that is used by the control rule meta-predicates `suggested-bindings` and `op-suggested-for-current-goal` to generate suggestions for further decisions during planner (see Section 4.5.2).

Figure 4.42 describes the basic cktree matching routine and will be explained throughout the next sections. Function **match_cktree** is given a cktree node q, the bindings β_q for the variables in q if it is a goal node, or in q's parent goal node, and the alternative being explored, that is the bindings for the variables seen so far in the cktree and the choices of operators being tried. It recursively computes the cost of the subtree rooted at q for the given quality metric. The computation is done by traversing the cktree. In the case of a binding cktree node, the cost of the subtree is computed by adding the cost of the operator itself given the quality metric with the cost of achieving the operator preconditions (Step 28). The call to **op_bnds_cost** applies the quality metric to the operator instantiation. Note that the cost of the operator may depend of the particular instantiation chosen.[12] When computing the cost of the preconditions, the case of universally quantified operator preconditions is dealt with separately (Steps 21-22) and will be explained later. Otherwise **match_cktree** is called in turn to estimate the cost of achieving each of the preconditions, and those costs are added (Steps 23-27).

4.5.4 Generating and Pruning Alternatives

The previous section explained how at each point the current alternative being explored is maintained. If a goal node has more than one child operator, the first one is kept as part of the alternative and the rest are stored to be explored later (Steps 8-10) of **match_cktree**. Similarly, in the case of a binding node the matcher computes the possible instantiations (Step 15). If there are more than one, the first one is kept as part of the alternative and the rest are stored to be explored later (Steps 17-19).

Figure 4.43 describes how the possible instantiations for a binding node are computed. First, all the legal bindings β_{legal} are generated by calling the planner's operator matcher (**get_all_bindings**): given the bindings that come from matching the goal with the operator's right hand side, the rest of the variables of the operator are instantiated in all possible ways that satisfy their type specification.

[12] Inference rules are represented in the cktree in the same way as operators. However, as inference rules do not affect the cost of the plan, if the binding node corresponds to an inference rule, **op_bnds_cost** returns 0.

match_cktree(q, β_q, alt)

1. case **type**(q)
2. goal cknode:
3. if [**true_in_state**(l) $\vee$ **in_marked_goals_p**(q, β_q)] $\wedge \neg$ **in_deleted_goals_p**(q, β_q) then
4. $<alt', cost> \leftarrow <alt, 0>$
5. else if **children**$(q) = \emptyset$;; no children
6. $<alt', cost> \leftarrow <alt,$**default_cost**$(q)>$
7. else
8. $O \leftarrow$ **children**(q) ;; the operator alternatives
9. $op_0 \leftarrow$ car(O)
10. if $|O| > 1$ then **store_remaining_alts**$(O - \{op_0\})$
11. $<alt', cost> \leftarrow$ **match_cktree**$(op_0, \beta_q, alt \cup \{op_0\})$
12. operator cknode:
13. $b \leftarrow$ car(**children**(q)) ;; operator nodes have only one child
14. $<alt', cost> \leftarrow$ **match_cktree**(b, β_q, alt)
15. binding cknode: ;; β_q are the bnds for the parent goal
16. $\mathcal{B} \leftarrow$ **all_bnds_for_ckop**(q, β_q, alt)
17. if $\mathcal{B} = \emptyset$ then return(*no legal bnds*)
18. $\beta_0 \leftarrow$ car$(\mathcal{B})$;; first bindings alternative
19. if $|\mathcal{B}| > 1$ then **store_remaining_alts**$(\mathcal{B} - \{\beta_0\})$
20. $alt' \leftarrow alt \cup \beta_0$
21. if **applied**(q) then **add_side_effects**(q, β_0)
22. if **forall_expanded_p**(q) then
23. $<alt', cost_{precs}> \leftarrow$ **match_expanded_forall**(q, β_0, alt')
24. else
25. $cost_{precs} \leftarrow 0$
26. for each $p \in$ **children**(q).
27. $<alt', cost_p> \leftarrow$ **match_cktree**(p, β_0, alt')
28. $cost_{precs} \leftarrow cost_{precs} + cost_p$
29. **add_marked_goal**(p, β_0)
30. $cost \leftarrow cost_{precs}$ + **op_bnds_cost**(q, β_0) ;; using the **quality metric**
31. **cknode_cost**$(q) \leftarrow cost$
32. return$(<alt', cost>)$

Figure 4.42: The basic cktree matching function. Note how the quality metric is used in Step 28.

all_bnds_for_ckop(b, β_{rhs}, alt)

1. $\beta_{legal} \leftarrow$ **get_all_bindings**(**parent**(b),β_{rhs}) ;; the planner's matcher
2. *constraints* $\leftarrow$ **instantiate**(**constraints**(b), alt)
3. $\beta_{pruned} \leftarrow \{bnd \in \beta_{legal} :$ **satisfy_p**(*bnd*,*constraints*)$\}$
4. If $\beta_{pruned} = \emptyset$
5. then return(β_{legal})
6. else return(β_{pruned})

Figure 4.43: Generating instantiations for a cktree binding node.

Then the set of those instantiations is pruned using the learned constraints that were described in Section 4.4.2.4. The purpose of the constraints is to make cktree use more efficient by pruning out bindings that would lead to lower quality plans. The constraints were learned when an achievement link was set between two nodes: if g_1 achieved g_2, a set of constraints was introduced so the arguments of g_1 and g_2 are the same. The constraints were stored in the binding node that introduced those variables, i.e. that generated values for them. The constraints are stored in the `constraints` slot of the binding node b as a list of constraints. Each element corresponds to an achievement link and has the form $((v_1\ v_2)^+)$ where v_1 is a variable introduced by the operator at the binding node b and v_2 is a variable at the constraining operator (that is, at the other extreme of the achievement link). To test the constraints when the matcher is visiting node b and generating values for the v_1's, the matcher instantiates the v_2's with the bindings that appear in the alternative (alt in Figure 4.43) currently being explored. If any of the v_2 in a constraint is not yet bound, the constraint is discarded. The remaining constraints are used to filter the possible bindings: every binding in β_{legal} that do not satisfy any of the constraints is discarded. If none of the bindings satisfy a constraint, no pruning occurs and all the legal bindings are explored.

The cktree in Figure 4.28 serves to illustrate how the constraints are used. A constraint of the form

```
(<mach9> <mach8>) (<hdev9> <hdev8>) (<spair9> <spair8>)                [1]
```

was learned and stored in the binding node for *drill-with-spot-drill* as described in Section 4.4.2.4. Assume now the cktree is used for guidance in a problem where the goal is to drill a hole and the available machines and tools are a drill press `drill7`, two holding devices `vise3` and `vise5`, and two drill bits (a twist drill `twist-drill4` and a spot drill `spot-drill1`). The cktree matcher starts by exploring one alternative instantiation for *drill-with-twist-drill*, say `(<mach8> drill7) (<hdev8> vise3) (<drillbit8> twist-drill4) (<spair8> side2-side5)`. As it traverses the cktree it needs to generate a choice of operator and bindings to achieve the `has-spot` precondition of drilling the hole. The only operator

known in the cktree is *drill-with-spot-drill*. Again there are several possible instantiations for it by combining the available holding devices and the allowed orientations: `<hdev9>` can be `vise3` or `vise5`, and `<spair9>` can be `side2-side5` or `side3-side6`. Applying the constraint above [1] the resulting instantiation is `(<mach9> drill7) (<hdev9> vise3) (<drillbit9> spot-drill1) (<spair9> side2-side5)`.

The aim of this pruning is to reduce the set of alternatives tried by focusing on those that can lead to better plans, as they did in the past. Satisfying the constraints means that the alternative will lead to a 0-cost subgoal. Note that the constraints may be contradictory, as different instantiations may lead to different savings by sharing subgoals with different nodes. This is the case when several achievement links have been learned from different episodes. In that case the algorithm keeps all the alternatives that satisfy any the constraints.

4.5.5 When to Stop Traversing the Cktree

The traversal stops when the cost of a subtree rooted at a node can be obtained directly, without having to traverse its children. This occurs when a cktree goal node is being explored in the following cases:

- The goal node's cost can be estimated as 0 (Steps 3 and 4 of Figure 4.42). Determining that a goal node has cost 0 is not trivial. It may have cost 0 if it is true in the initial state and has not been deleted by an operator. It may have cost 0 if it is added by another operator. Step 3 in Figure 4.42 captures the conditions under which a goal node has estimated cost 0. Section 4.4.2.3 described how the adding and deleting information is stored in the cktree's achievement and deletion links. Those links were recorded from previous planning experience in which the additions and deletions actually occurred. Whether they will occur in the actual planning problem depends on how the variables in those nodes are instantiated. For example, two `holding` goal nodes g_1 and g_2 may be connected by a `how-achieved` link if they were instantiated with the same bindings (machine, orientation, etc) in a past problem; if in the current problem the two goals are instantiated with different objects, say machines, the `how-achieved` link will not be used to estimate a 0 cost for one of the goals.

  When the cktree matcher visits a goal node or a binding node, it records its possible side effects, that is, the nodes linked as achieved or deleted by the current node with the bindings required for the achievement or deletion to actually happen. Those (possibly) added or deleted goals are stored respectively in two variables called *marked_goals* and *deleted_goals*. Steps 27 and 20 call respectively functions **add_marked_goal** and **add_side_effects** which are described in Figure 4.44(a) and (b). **add_marked_goal** is called after a goal node is explored (Step 27 of Figure 4.42) and adds to the set *marked_goals* those nodes that are pointed by the goal's `achieves-too` link. **add_side_**

effects is called when a binding node is explored (Step 20 of Figure 4.42) and adds the possible side effects of the binding node (stored in its `applied` slot) to *marked_goals* and *deleted_goals* depending on whether the effect is positive (add) or negative (delete). When the matcher is exploring a goal node (Step 3 of Figure 4.42 it looks at these lists to estimate whether the cost of the goal is 0.

---

add_marked_goal(p, β)

1. $\beta_{rel} \leftarrow \{<var, val> \in \beta : var \in$ **arguments**$(p)\}$
2. for each $g \in$ **achieves_too**(p)
3. $\quad marked\_goals \leftarrow$ push$(<g, p, \beta_{rel}>, marked_goals)$
4. $visited\_cknodes \leftarrow$ push$(<p, \beta_{rel}>, visited_cknodes)$ visited_cknodes

---

(a)

---

add_side_effects(b, β)

1. for each $<g, type, e> \in$ **applied**(b)
2. $\quad \beta_{rel} \leftarrow \{<var, val> \in \beta : var \in$ **arguments**$(g)\}$
3. $\quad$ if $type =$ `:positive`
4. $\quad$ then $marked\_goals \leftarrow$ push$(<g, b, \beta_{rel}>, marked_goals)$;; a positive side effect
5. $\quad$ else $deleted\_goals \leftarrow$ push$(<g, b, \beta_{rel}>, deleted_goals)$;; a negative side effect

---

(b)

Figure 4.44: Maintaining the achieved and deleted goals.

- The node has no children, and therefore there is no further information in the cktree to estimate the cost of achieving that subgoal (Steps 5 and 6). In that case a default cost is assigned. For example one can use the minimum of the costs assigned by the quality metric to the domain operators that are relevant to that goal.
- The estimate for the subtree rooted at the goal node is the same as for the previous alternative explored. Therefore it has been computed before and can be reused. This leads to considerable savings in the matching process and will be explained in detail in Section 4.5.8.
- A threshold value is exceeded. Section 4.5.7 will describe how the matcher explores multiple alternatives. The best value for the alternatives tried so far is used as a threshold. When that threshold is exceeded, exploration stops and the current alternative is discarded.

match_expanded_forall(b, β, alt) ;; b is a cktree binding node

1. **initialize_masks**
2. $cost \leftarrow 0$
3. for each $\beta_\forall \in$ **generate_forall_bnds**(b, β)
4. for each $p \in$ **children**(b)
5. if [**mask_not_visited_p** $\vee$ **in_deleted_goals_p**$(b, \beta_\forall)$] then
6. $<alt, c_p>$ $\leftarrow$ **match_cktree**$(p, \beta_\forall, alt)$
7. $cost \leftarrow cost + c_p$
8. for each $p \in$ **children**(b) **add_marked_goal**$(p,$ `:forall`$)$
9. return$(<alt, cost>)$

Figure 4.45: Estimating the cost of achieving a universally quantified precondition.

4.5.6 Matching Universally Quantified Cktree Preconditions

If an operator precondition is universally quantified the planner expands it as a conjunction of all the possible instantiations of the precondition given the scope of the quantification, and then proceeds to achieve each of the conjuncts. Therefore in a plan tree built from such planning episode the binding node corresponding to that operator will have one child for each of the instantiated preconditions. When a cktree is learned from the plan tree a single cktree goal node is created for all the instantiations of a universally quantified precondition (see Section 4.4.2.2). Therefore **match_cktree** needs to deal in a special way with this case (Steps 21-22 of Figure 4.42). Figure 4.45 describes the matching of quantified preconditions. The cost of all the possible instantiations of the preconditions is accumulated and returned as the cost of achieving the operator precondition.

In what follows we will describe how matching is made more efficient in the case of universally quantified preconditions by using *masks*. The instantiated preconditions are generated incrementally. The possible bindings are computed (Step 3) incrementally. Then the preconditions are instantiated in turn using each of those bindings (Step 4). Note that the same instantiation may be generated several times when not all the quantified variables are arguments of the precondition. For example, if the precondition has the form `(forall (<a> <b>)(and (p1 <a>)(p2 <a> <b>)))` and `<a>` may take value a_1 and `<b>` may take values b_1, b_2, then `(p1` a_1`)` will be generated twice, once per value of `<b>`. The matcher must avoid adding the cost of the instantiated precondition several times unless the precondition is deleted when other preconditions are achieved. To speed up the matching process by avoiding the computational cost of instantiating the goal node for the precondition and of calling **match_cktree**, the matcher keeps track of the preconditions generated.

Assume that $var_1; var_2, ...var_n$ are the universally quantified variables, and that an instantiation is represented as a vector $inst[v_1, v_2, ...v_n]$ where v_i is the value assigned to var_i in that instantiation. A *mask* is stored in each goal cktree node corresponding to a quantified precondition (Step 1). A mask for a precondition p is an n-element vector indexed on the quantified variables, such that $mask_p[i]$ is 1 if var_i is an argument of the precondition, and 0 otherwise. Every time **match_cktree** is called for a given goal node and instantiation, a vector $inst_{masked}$ is also stored in the goal node; $inst^p_{masked}$ represents the instantiation $inst$ masked with $mask_p$. Masking means that $inst^p_{masked}[i] = 0$ if $mask_p[i] = 0$ and $inst^p_{masked}[i] = inst[i]$ otherwise. When a new instantiation $inst'$ is proposed (Step 5) **mask_not_visited_p** checks whether the instantiated precondition has been considered already by masking $inst'$ and finding it in the set of visited instantiations $\{inst^p_{masked}\}$. If it has been visited already (that is, the cost of achieving it was computed by calling **match_cktree**), and has not been deleted subsequently, the precondition is ignored. This amounts to assigning it cost 0.

Finally, the goal nodes are added to the list of *marked_goals*. Only one occurrence of each precondition is added (instead of an element for each possible instantiation). **initialize_masks** (Step 1) computes the masks if they have not been computed in previous explorations of the cktree, or empties the list of visited masks $\{inst^p_{masked}\}$ stored in the goal node otherwise.

4.5.7 Exploring Multiple Alternatives Efficiently

Section 4.5.4 described how multiple alternatives may be possible at a node when the cktree is being traversed. There may be several operators available to achieve a goal stored as children of the goal cktree node. These correspond to operators that the planner used in the past to solve the goal either planning by itself or as part of an improved plan proposed by the human domain expert. There may also be several ways to instantiate the operator in the given problem, even after using binding constraints to prune them.

Assume that the planner is making an operator decision and op_1 and op_2 are the alternatives. If the planner commits to op_1, it will then have to commit to an instantiation among the legal ones. Then when working on each of its subgoals, it may have to choose an operator among a set of relevant ones, and so on. Assume that the cktree matcher is invoked (by calling **current_goal_and_pref_op**, Section 4.5.2) to estimate the cost of operators op_1 and op_2 and thus choose the one that looks more promising. The estimation of the cost depends on what further alternatives (bindings, operators for subgoals) are chosen. For example, if op_1 and op_2 are using a drill press and using a milling machine to drill a hole, which operator is more promising depends not only on the operator itself but on the particular instance of drill press or milling machine used, and on the choice of tool, holding device, and orientation. Maybe using *milling-machine3* with *vise1* is better than using *drill-press1* with *vise1* because *vise1* is set on *milling-machine3*, and both of those alternatives are better than *milling-machine7* with *vise1* because that machine is holding a different vise. If multiple alternatives are available at

a cktree node, *all* of them must be explored to estimate the cost of the alternative at the root of the cktree.

Figure 4.41 described the basic algorithm for traversing the cktree. In Step 5 **match_cktree** is called to estimate the cost $cost_0$ of the first alternative explored alt_0. As multiple alternatives may be available at some of the nodes visited, **match_cktree** explores the first of those and stores the rest in a global variable all_alts (Steps 9 and 18 of Figure 4.42). Therefore the remaining alternatives must be explored now by calling **match_all_alts** (Step 8 of Figure 4.41), which is described in Figure 4.46.

---

match_all_alts$(q, \beta_0, alt_0, cost_{best})$;; all_alts is a global variable

1. $alt_{prev} \leftarrow alt_0$
2. $alt \leftarrow$ **next_alt**(all_alts)
3. $\Delta \leftarrow$ **diffs_with_prev_alt**(alt_{prev}, alt)
4. **mark_diffs**(q, Δ)
5. $<alt', cost_{alt}> \leftarrow$ **match_cktree_marked**$(q, \beta_0, alt, cost_{best})$
6. **unmark_all_nodes**
7. $other\_cktree\_nodes \leftarrow \{<g_1, g_2, \beta> \in marked\_goals$ s.t. **in_other_cktree_p**$(g_1, q)\}$
8. **store_n_best**$(alt', cost_{alt}, marked\_goals, deleted\_goals, other\_cktree\_nodes)$
9. if $cost_{alt} < cost_{best}$ then $<alt_{best}, cost_{best}> \leftarrow <alt', cost_{alt}>$
10. if $all\_alts = \emptyset$
    then
11. return$(<alt_{best}, cost_{best}>)$
    else
13. $alt_{prev} \leftarrow alt'$
14. goto **2**

Figure 4.46: Matching the available alternatives. Steps 7 and 8 will be explained in Section 4.5.9.2.

The matcher iterates over the available alternatives to estimate their cost. The best cost so far (initially $cost_0$) is used as an upper bound to stop matching when the current alternative is going to be worse than the best one so far. This is another way to limit the matching effort.

The set of all possible alternatives *all_alts* is not completely computed upfront. For example, if an operator is stored pending to be explored, the possible instantiations for it have not been yet generate. They will only be generated when the operator becomes part of the current alternative alt. As there may be more than one instantiation, the first one is explored and the rest kept in *all_alts*. Therefore *all_alts* is managed dynamically: when an alternative is explored, it may prompt the generation of new alternatives that are added to *all_alts*.

4.5.7.1 Cktree Matching in Relationship to Planning

We have just described how if multiple alternatives are available at a cktree node, the cktree matcher must explore *all* of them to estimate the cost of the alternative at the root of the cktree. Obviously this process can be expensive; however *"all"* can be relaxed to make the cktree matching process efficient. Using binding constraints as described in Section 4.5.4 prunes the alternatives explored. Other methods are described below. In addition, the choices that lead to the best alternative during the cktree exploration can be stored and used at planning time when the planner faces the corresponding decision. In this way the control knowledge captured by the cktrees is used *globally*, to guide a sequence of planner's decisions, instead of *locally* as was the case when simply using control rules.

It is worth clarifying the distinction between planning and cktree matching. When the planner must make an operator or bindings decision, it calls the cktree matcher with the goal g for which the decision must be made. The cktree matcher uses the cktree to estimate the cost of each alternative available to the planner. To do that the cktree matcher traverses the cktree(s) predicting, based on past planning experience, the decisions that the planner will make and the subgoals that it will explore. The matcher returns guidance for the planner's decision, that is, an operator, or a set of bindings, for g. As a side product the matcher stores guidance for future decisions that the planner will face. Then the planner commits to the cktree matcher's advice for g and continues planning. Each time it faces a choice point, it checks, prompted by the control rules described in Section 4.5.2, whether the previous call to the cktree matcher stored a suggestion for the new choice point. If so, the suggestion is taken.

4.5.8 Reusing Computation Among Alternatives

The cost of some subtrees may remain the same for different alternatives and therefore can be reused. For example, if the difference between two alternatives for drilling a hole in a drill press is the tool used, the cost of holding the part is still the same for both alternatives, and only the cost of holding the tool needs to be recomputed to estimate the cost of the second alternative. Step 3 of Figure 4.46 computes the differences Δ between the current and previous alternatives. Then Δ is used to mark the nodes in the cktree such that the cost of the subtrees rooted on them needs to be recomputed (Step 4). Therefore if a node is not marked the cost stored in it is reused. (Recall that the cost at each node was stored in Step 29 of Figure 4.42.)

Thus **match_cktree_marked** (Step 5 of Figure 4.46) is slightly different from **match_cktree**. If a node is marked, it recursively computes the cost of the subtree, as **match_cktree** does. However if the node is *not* marked, the cost stored at the node is returned (a test that would occur at a goal node before Step 3 of **match_cktree**).

Figure 4.47 describes how the nodes whose cost has to be recomputed are marked. All the marks are wiped out after exploring each alternative. The differences Δ between the current

mark_diffs(q, Δ)

```
1.  if Δ_is_bnds_p(Δ)                          ;; Δ is the instantiation of some variable(s)
    then
2.      for each <var, val> ∈ Δ
3.          for each touched ∈ var_hash[var]    ;; cf Figure 4.22 (Step 10)
4.              mark(touched,q)
    else                                        ;; Δ is an operator for some subgoal
5.      q ← parent(Δ)
6.      mark_children(Δ, q)
7.  marked_p(q) ← t
8.  cost(q) ← unknown
9.  M ← push(q, M)
10. for each m ∈ M
11.     delete_from(m,marked_goals,deleted_goals,visited_cknodes)
```

mark(p, q)

```
1.  if ¬ marked_p(p) ∧ p ≠ q
    then
2.      marked_p(p) ← t
3.      cost(p) ← unknown
4.      M ← push(p, M)
5.      case type(p)
6.        goal cknode:     for each dep ∈ achieves_too(p) ⋃ how_achieved(p) ⋃ deleted_by(p)
7.                             mark(dep, q)
8.        binding cknode: for each dep ∈ applied(p)
9.                             mark(dep, q)
10.     mark(parent(p),q)
```

mark_children(p, q)

```
1. mark(p, q)
2. for each child ∈ children(p)  mark_children(child,q)
```

Figure 4.47: Marking the cktree nodes whose estimated cost needs to be recomputed.

alternative and the previous alternative are used to determine which nodes to mark. This difference can be the instantiation of some variables, or the operator chosen to achieve a goal.

- If the difference is some variable binding, all the nodes that are affected by that variable are marked since their cost estimate must be recomputed (Steps 1-4 of **mark_diffs** in Figure 4.47). For example, if the affected node is a goal node, the cost of achieving it may vary as the goal is instantiated in a different way.

  The nodes affected by the change of a variable binding are computed efficiently (Step 3), since they are stored in a hash table when the cktree is built, as described in Section 4.4.2.2. The hash table is indexed on the variables that appear in the cktree. Figure 4.48 partially shows the contents of the hash table for the cktree learned in Figure 4.28.

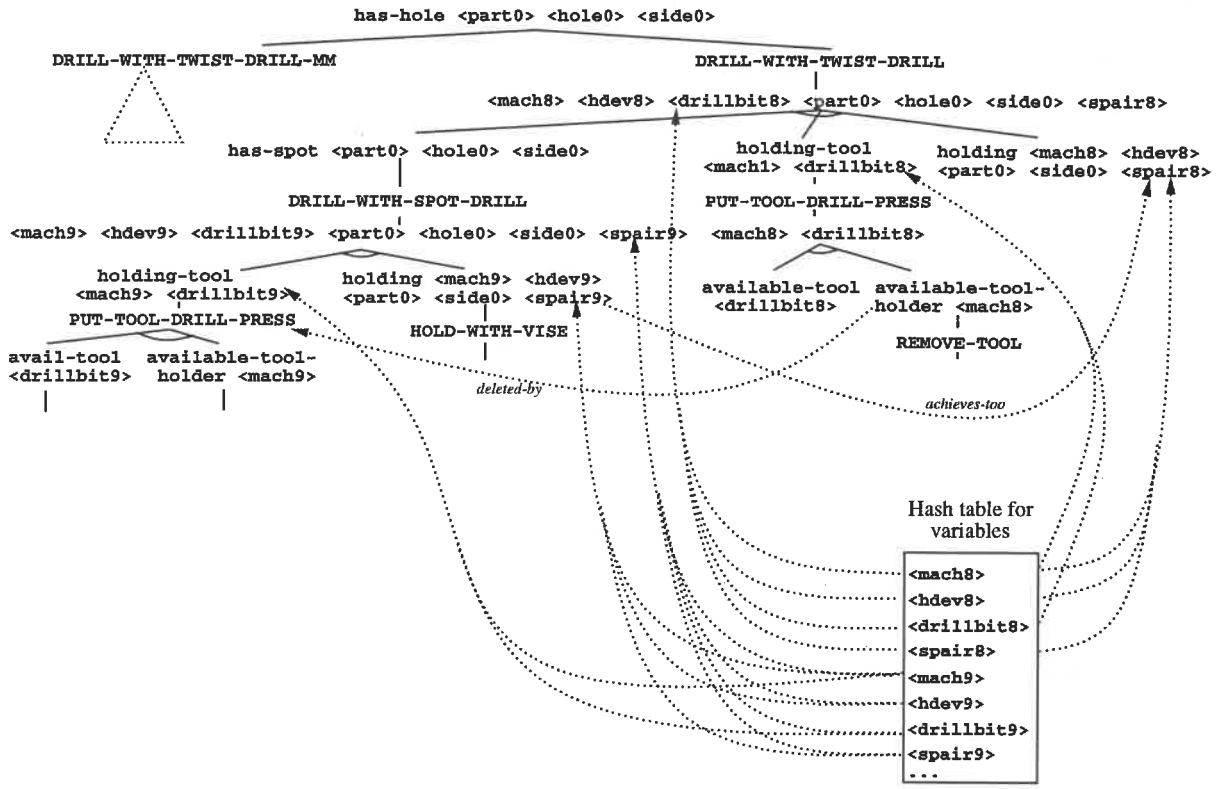


Figure 4.48: The `has-hole` cktree of Figure 4.28. This figure shows the contents of the hash table of pointers from the variables to the nodes that use them. The pointers are represented by the unlabeled dotted arcs with origin in the hash table. They are used to mark the nodes whose value may have changed with respect to the previous alternative explored, thus speeding up the cktree matching by limiting the number of nodes whose estimate must be recomputed.

- If the difference between the two alternatives is an operator, all the operator node descendants are marked (Steps 5-6 of **mark_diffs** in Figure 4.47). Note that since the operator

is different, those nodes will not be visited and so their cost will not be needed. However the nodes should be removed from the lists of marked, deleted, and visited nodes, and the nodes that depend on them via achievement and deletion links need to be marked as well since their cost may be different for the current alternative.

The marks are stored in the **marked_p** slot of the node. The middle part of Figure 4.47 (**mark**) gives the details of how a node n is marked. First a mark is put in the **marked_p** slot of node n and its cost is set to a dummy value. Then all the nodes linked to n by achievement and deletion links are marked recursively. Also, n will be removed from the lists of marked goals, deleted goals, and visited nodes, as it will be visited again (if needed). Finally, all the ancestors of n are marked because n's cost estimate may change and the ancestors' costs depend on the cost of n.

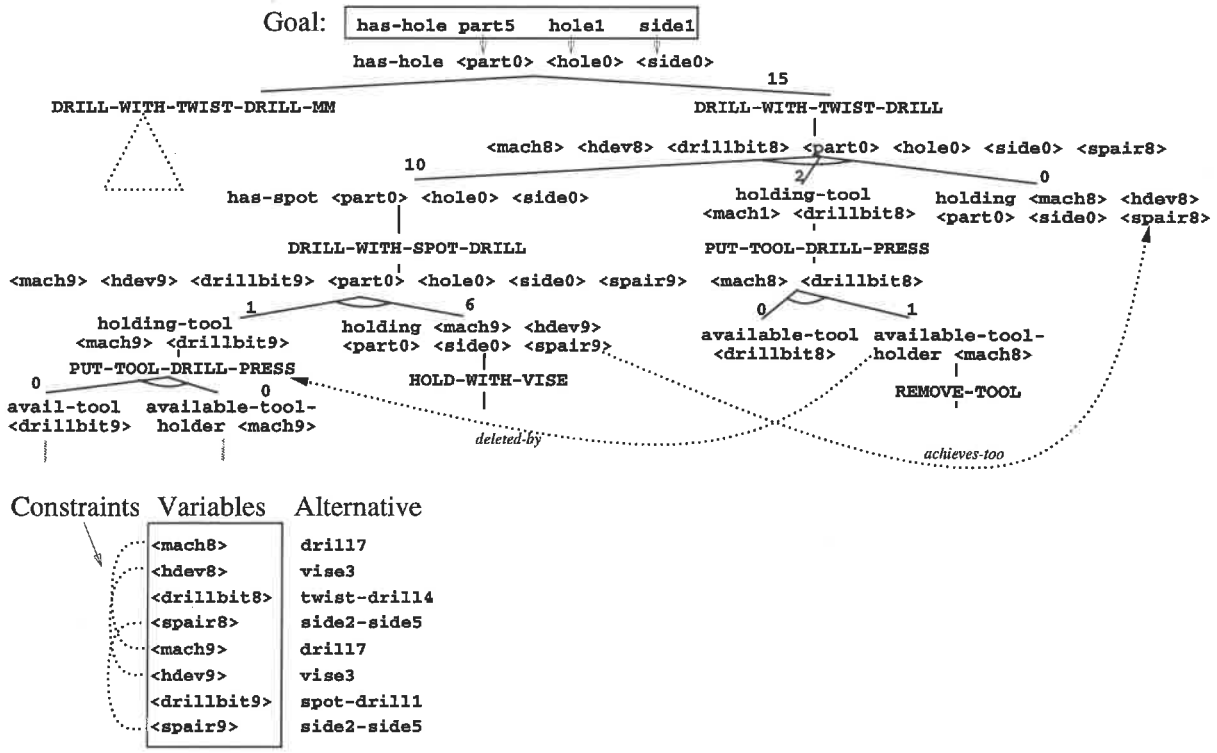


Figure 4.49: Using the cktree to estimate the cost of the first alternative. The problem goal is at the top. A number next to a node indicates the estimated cost of the subtree rooted at that node. The bottom part of the figure shows the relevant variables, the constraints among them, and the alternative currently being explored. The constraints are stored at binding node for *drill-with-spot-drill* (cf Section 4.5.4).

To illustrate how cktrees can be used efficiently by reusing the cost estimates among alternatives we now go back to the problem described in Section 4.5.4 in which the goal is to drill a hole

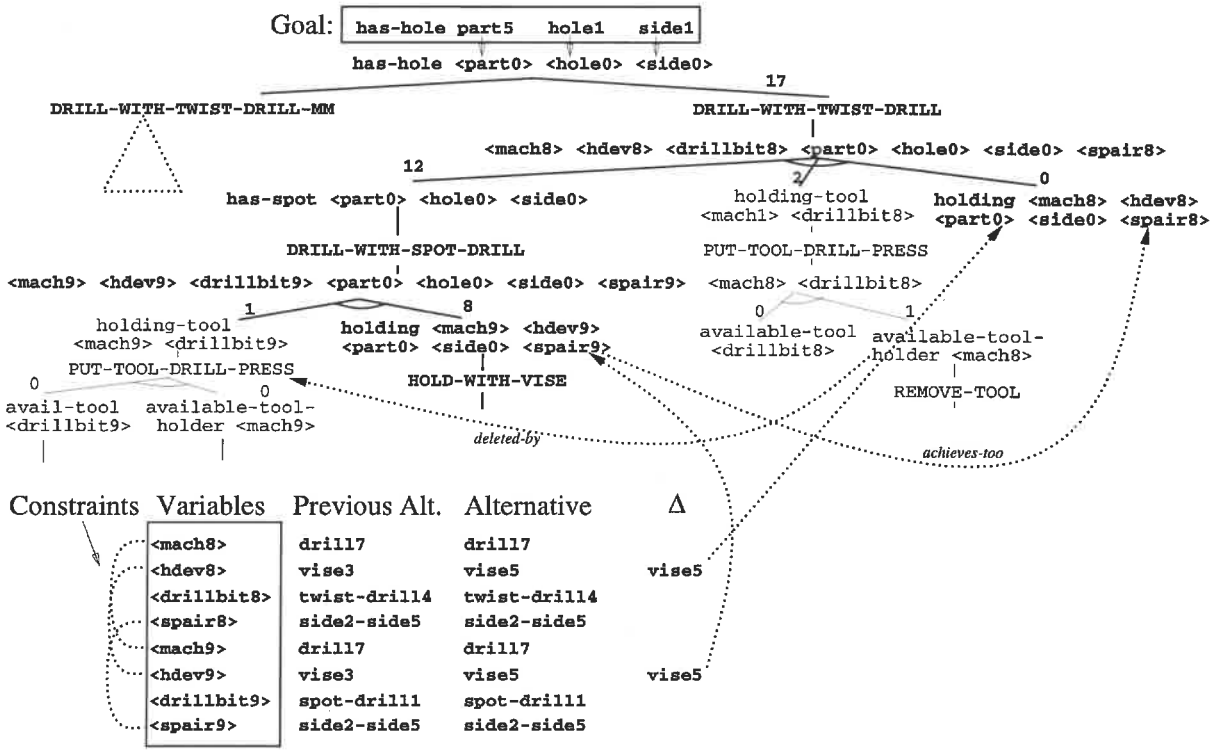


Figure 4.50: Reusing previous estimates to estimate the cost of the second alternative. The bottom part shows the previous and current alternatives and the differences (Δ) among them. The differences are used to determine which parts of the cktree need to be explored because their estimate may have changed. Subtrees depicted in a lighter font represent parts of the cktree not explored because the estimate for the current alternative could be reused.

`hole1` on `side1` of `part5` and the available machines and tools are a drill press `drill7`, two holding devices `vise3` and `vise5`, and two drill bits (a twist drill `twist-drill4` and a spot drill `spot-drill1`). Figure 4.48 shows part of the available cktree and the contents of the hash table of variable pointers. Assume that the cktree is used to estimate the cost of operator *drill-with-twist-drill*. The first alternative tried is depicted in Figure 4.49 with the estimated cost. A lighter font is used in this figure and the ones that follow to indicate parts of the cktree that are not explored when estimating the cost of the current alternative. Note that the constraints stored at the binding node for *drill-with-spot-drill* are used to prune the possible bindings for that node. Section 4.5.4 described this step in detail. The second alternative is explored in Figure 4.50. The difference Δ between the two alternatives is the holding device (namely `<hdev8>` and `<hdev9>`) used, as shown in column Δ at the bottom of the figure. Recall that tha hash table of variable pointers stores pointers to the nodes whose estimate may change when their instantiations change. The arrows in the figure represent those pointers. Those nodes are marked so their cost estimates are recomputed. Their ancestors are also marked since

their estimates will vary when the new costs are propagated up. In the example, the cost of `holding` is higher for the current alternative because the holding device `vise5` is not on the drill press (but `vise3`, in the previous alternative, is). The figure shows how the estimates for the `holding-tool` subtrees for the first alternative can be reused when traversing the cktree for the second alternative.

4.5.8.1 Discussion: Caching and Reusing Quality Estimates

The algorithm described is an example of caching to exploit redundancy in the computation by using previously computed information. Our implementation was inspired by Perlin's description of network-based programs and of efficient control mechanisms for them [Perlin, 1988]. The *data* part of a network-based program is a network, or directed acyclic graph. The *control* mechanism is some node enumeration algorithm in which each node is visited exactly once, after all the node predecessors have been visited. The nodes in the network may contain memory for executable code and passive data. The node's code usually acts on the input data in the context of the node's memory state, and the result is often stored in the node's memory. The node's memory persists between cycles of data. Network-based programs are very efficient. If topological sorting is used to traverse the tree, propagating from all the leaves and keeping newly computed values in the node's memory guarantees that each node is recomputed only once. If changes are made to only a subset of the network leaves, by using the node's memory the computation needs only be propagated to the transitive closure of the nodes affected by the changes. This kind of state-saving incremental algorithms perform minimal recomputation and are directed from just the *changes* to their input. Further efficiency gains come by restricting the network traversal to the subgraph influencing only select node computations. The efficiency gains are greater when only a small fraction of the input data (and hence the partial computations) changes in every cycle. Programs for conjunctive matching and constraint reasoning among others are frequently cast in this network form for efficiency.

These features of network-based programs match those of the control knowledge trees and their use to estimate the cost of planning alternatives. The leaves correspond to the pointers from the hash table of variables to the nodes that use those variables. The data correspond to the values assigned to those variables in each alternative. The changes on values are propagated bottom-up in order to mark the nodes that need recomputation. Therefore the incremental changes in the variable values between alternatives lead to reducing the computation effort. Each cycle of the computation corresponds to estimating the value for an alternative by traversing the cktree. There are differences though between the description of network-based programs and cktrees. The cktrees are not DAG's, and the alternatives (e.g. the data) are generated incrementally as the cktree is traversed. Still the ideas of state-saving (storing the cost of the node for the previous alternative and reusing it if the node is not marked) and incremental computation (exploring only the parts of the trees where recomputation is needed) lead the development of

our algorithm.

4.5.9 Cktree Matching When There Are Interacting Goals

The description of the cktree matcher in previous sections has assumed that estimating the cost of an alternative operator or binding for a goal was independent of whether other goals were present in the problem. However in many cases several goals must be achieved and they are not independent, that is, achieving a goal may have positive or negative effects for achieving other goal(s). This section describes how cktrees can be used efficiently when there is more than one goal. Section 4.4.4 described how the learmer builds the relevant cktree in this case.

The fact that two goals are not independent is captured in the cktree formalism by having achievement or deletion links from a node in a cktree to one or more nodes in another cktree. Therefore to estimate the cost of a given alternative the nodes in other cktrees should also be considered. The cktree matcher avoids the full exploration of the cktrees for the other interacting goals. It selectively traverses only the parts of the other cktrees that are relevant to the current alternative.

4.5.9.1 An Example

This section illustrates with an example how the matcher works when other goals interact with the goal for which an alternative is being chosen. Assume that the problem at hand is to solve g_1 and g_2 and the planner is making a choice about which operator to use in order to achieve g_1. For illustration purposes we start by showing in Figure 4.51 the plan trees that would result of pursuing two different alternative operators to achieve g_1, namely op_1^A and op_1^B. The quality metric assigns costs 3 and 5 respectively to these operators. Note that in spite of the smaller cost of op_1^A, the plan that uses op_1^B is better because of the savings in achieving subgoal g_{22}. However if only the part of the plan tree corresponding to achieving g_1 (that is, the subtrees rooted at g_1) is considered the plan that uses op_1^A looks better, as the costs of those subtrees are 5 and 14 respectively. Note that when the planner is making the choice of operator for g_1 the plan trees in the figure are not available. We have presented them here only for illustration purposes to describe the plans that would result of each alternative.

Assume that the learner has seen similar problems before, i.e. problems in which the goal was the conjunction of g_1 and g_2, and has learned the cktrees in Figure 4.52. In order to make the operator decision for g_1 the planner calls the cktree matcher. With the algorithms described in previous sections, the cktree matcher would only look at the g_1 cktree and estimate the cost of the two alternatives, op_1^A and op_1^B, as 5 and 14 respectively, suggesting to the planner to choose op_1^A. But this is the incorrect choice, as Figure 4.51 showed. The cktree matcher should consider the need to achieve a second, interacting goal g_2 and use g_2's cktree as well when

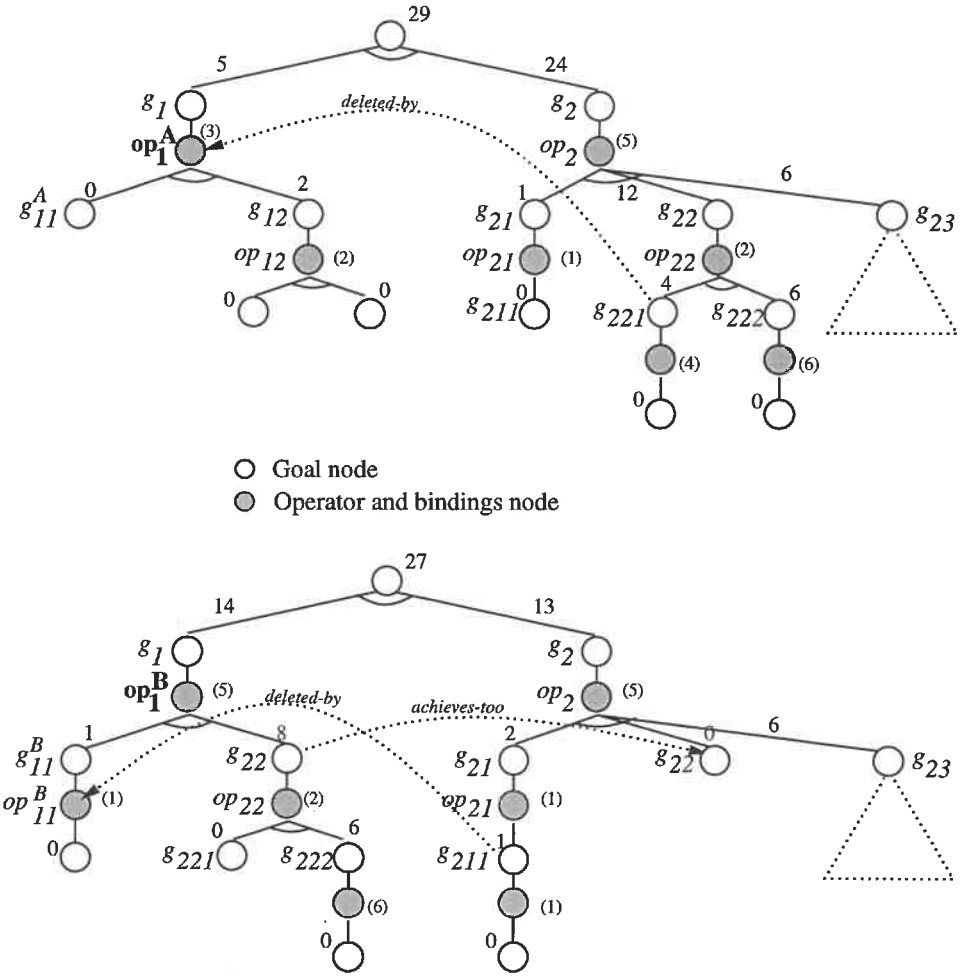


Figure 4.51: Two plan trees to solve goals g_1 and g_2. A number in brackets next to an operator is the cost of the operator given the quality metric. A number without brackets next to a goal node is the cost of the subtree rooted at that node. Operator and binding nodes have been merged together in the figure for clarity.

exploring each alternative for g_1. Exploring the whole g_2's cktree for each alternative would be expensive and is not necessary. For example, it is useless to explore the g_{23} subtree when estimating the cost of op_1^A and op_1^B, as g_{23} does not interact with g_1's subgoals (there are no achievement or deletion links between the subtrees), and the cost of g_{23} is independent of the alternative operator for g_1 explored. In other words, knowing the cost of achieving g_{23} would not help to decide between op_1^A and op_1^B. However the cost of achieving the g_{22} subgoal of g_2 is relevant. That cost should be added to the estimate of using op_1^A, because g_{22} will need to be achieved eventually, and it would be achieved for free should op_1^B be the chosen alternative (as indicated by the *achieves_too* link in the figure).

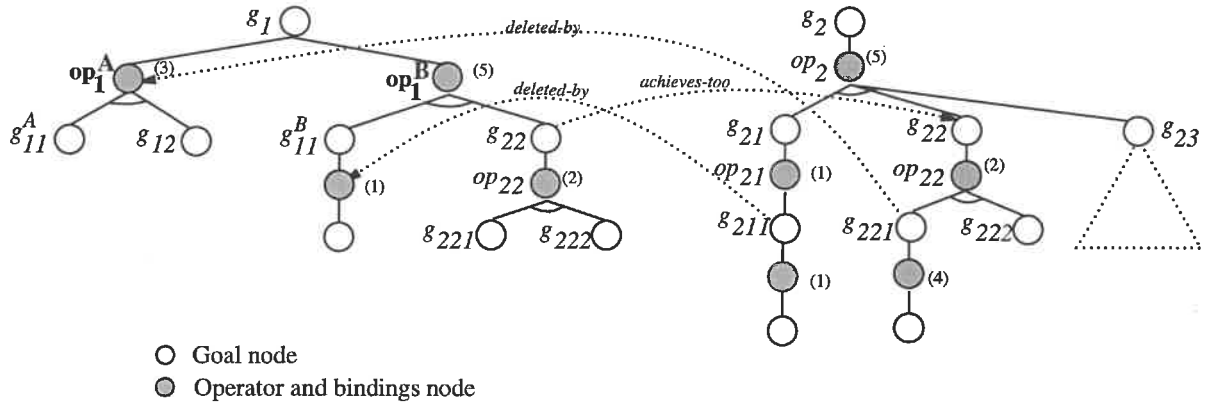


Figure 4.52: Two cktrees for goals g_1 and g_2 learned from previous planning experience. The number in parentheses next to an operator node correspond to the quality metric value for the operator. They are not part of the cktrees, as they may change with the operator instantiation and with the quality metric itself, but they are shown for illustration purposes.

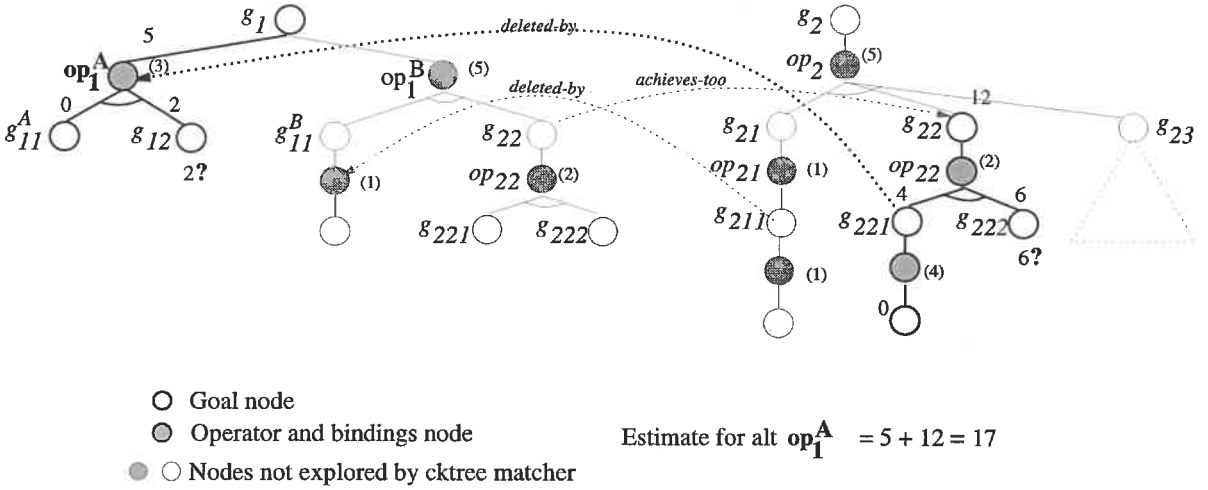


Figure 4.53: Illustration of the cktree matching process, that is, the cost estimates found and the parts of the cktree explored, for op_1^A, the first operator alternative for g_1. A number in brackets next to an operator is the cost of the operator given the quality metric. A number without brackets next to a goal node is the estimated cost of the subtree rooted at that node.

Similarly the cost of achieving g_{221} should be added to the estimate for op_1^A since it deletes g_{221} as a side effect. But as this cost is included in the cost of achieving its parent g_{22}, the matcher does not need to take any action.

Finally, operator op_{11}^B (a descendant of op_1^B) will delete g_{211} as a side effect. Therefore the estimate of cost of achieving g_{211} will be added to the estimate for op_1^B, but not to that of op_1^A

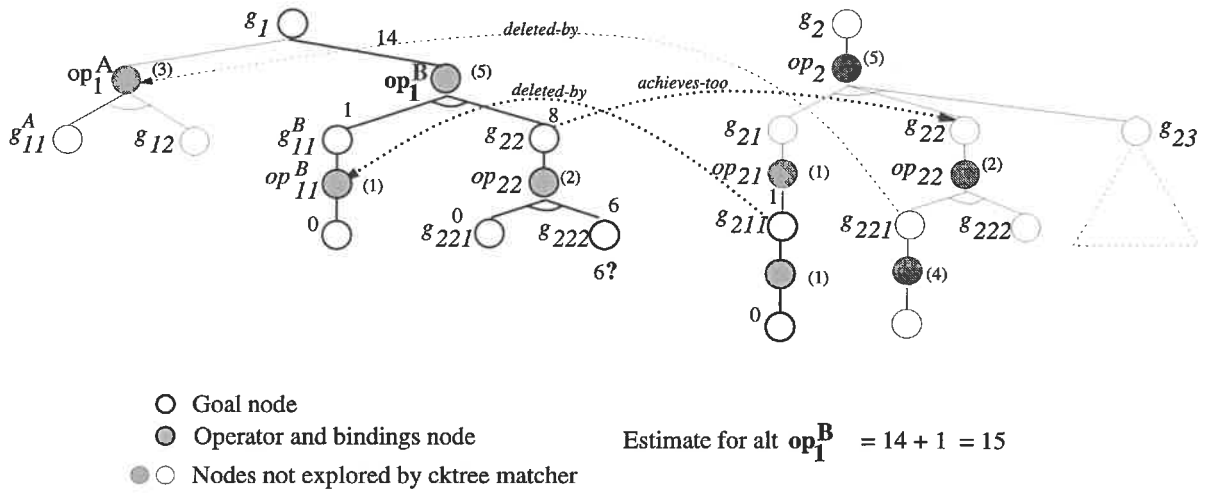


Figure 4.54: Illustration of the cktree matching process for op_1^B, the second operator alternative for g_1.

because it will not be deleted in that case. Figure 4.53 shows the estimates and the parts of the cktrees explored when estimating the cost of the first alternative, op_1^A. The estimated cost is the sum of the cost for the op_1^A subtree (5) and the cost of achieving g_{22} (12). Figure 4.54 shows those for the second alternative, op_1^B. In this case the estimated cost is the sum of the cost for the op_1^B subtree (14) and the cost of achieving g_{211} (1).

Thus which parts of the g_2 cktree are explored is determined by the achievement and deletion links from the parts of the g_1 cktree explored for both alternatives. For example, the cost of g_2 is added to the estimate for op_1^A because g_2 was linked with an *achieves_too* link to op_1^B subtree. Therefore to determine the estimate for op_1^A, information gathered during the exploration of the g_1 cktree for both op_1^A and op_1^B is used.

Given these results the matcher suggests the planner to use op_1^A to achieve g_1. Additionally the matching process will provide guidance for the bindings for op_1^A and op_2 that lead to that estimated cost, since those bindings were relevant for the 0-cost of g_{22}.

4.5.9.2 Cktree Matching Revisited

Section 4.5.3 introduced a first version of the meta-predicate `current-goal-and-pref-op` which invokes the cktree matcher for a given goal. That definition was a simplified one that assumed that estimating the cost of an alternative operator or instantiation of an operator for a goal did not depend on the existence of other possibly interacting goals. Therefore it only explored the cktree corresponding to the current goal. The example in the previous section illustrated how the matcher should explore, at least partially, the cktrees corresponding to other

current-goal-and-pref-op

1. $g \leftarrow$ **current_goal**
2. $ckroot \leftarrow$ **relevant_cktree**$(g, cktrees)$;;cf Figure 4.14
3. $\beta_0 \leftarrow$ **bnds**(g,**name**($ckroot$))
4. for each $op \in$ **children**($ckroot$)
5. $<alt_0, cost_0> \leftarrow$ **match_cktree**(op, β_0, β_0)
6. **store_n_best**(alt_0, $cost_0$,*marked_goals*,*deleted_goals*,*visited_cknodes*)
7. **match_all_alts**($op, \beta_0, alt_0, cost_0$)
8. $alt_{best} \leftarrow$ **update_with_other_cktrees**(*g*,*ckroot*,*n_best_alts*) ;; Figure 4.56
9. **store_prefs_for_subgoals**(alt_{best})
10. return(**choice**(alt_{best}))

Figure 4.55: The definition of **current-goal-and-pref-op** revisited to explore the cktrees of other goals. **store_n_best**($alt, cost$) stores the alternative alt, its cost and the context (marked and deleted goals and visited nodes) so they are used when the alternative's estimate is updated with the exploration of other cktrees.

update_with_other_cktrees(*g*,*ckroot*,*cost*,*data*,$data_{other}$)

1. $cost_{updated} \leftarrow cost$ + **update_alt**(*ckroot*,*cost*,*data*,$data_{other}$) ;; Figure 4.57
2. $cost_{updated} \leftarrow$
   $cost_{updated}$ + **update_with_same_cktree_goals**(*g*,*ckroot*,$cost_{updated}$,*data*) ;; Figure 4.59

Figure 4.56: Updating the estimate for an alternative by considering other goals. *data* refers to the information stored for the alternative by **store_n_best** (see Figure 4.55).

goals in the problem. It also illustrated how to determine which parts of other cktrees need to be explored, by using the information from more than one alternative for the first goal (g_1). Figure 4.55 shows the complete definition of `current-goal-and-pref-op` that considers the existence of other possibly interacting goals in the problem. First the matcher explores only the cktree for the current goal g, as explained in previous sections. The best alternatives explored are stored (Step 6 and also Step 8 of Figure 4.46) and then their estimates are updated by looking at cktrees that correspond to other problem goals *and* have been touched by achievement or deletion links from the visited nodes in g's cktree. Step 7 of **match_all_alts** (Figure 4.46) keeps track of those nodes in other cktrees.

Figure 4.56 describes how each of the alternatives is updated. For clarity we have decomposed it in two parts:

- In Step 1 the matcher explores the cktrees that correspond to other problem goals by using

the information stored with the current alternative $data$ and other alternatives $data_{other}$. Next section explains how this is done.

- In Step 2 the matcher considers other problem goals that would correspond to the same cktree as g, but obviously with a different instantiation. This is a separate step because the process exemplified above to determine which parts of the other cktrees are relevant cannot be used here, since it is actually the same cktree. Section 4.5.9.4 describes this case in detail.

4.5.9.3 Matching (Parts of) the Cktrees for Other Goals

Let g be the goal for which the matcher has been called to find a choice of operator or bindings, and $data$ be the alternative which is being explored for g, and whose estimate is being incremented by traversing parts of other cktrees. Figure 4.57 states how this traversal occurs. Its steps can be divided in three stages:

1. Set the context with the values of *marked_goals* and *deleted_goals* resulting from the exploration of that alternative. They are available as part of $data$ (Steps 1-2)
2. Determine which parts of other cktrees must be traversed (Steps 3-10).
3. Traverse the relevant parts of other cktrees and accumulate the resulting costs (Steps 11-14). This is not different from other calls to the cktree matcher (**match_cktree**).

We will concentrate now on how the relevant parts of the other cktrees are determined. Two categories of nodes in other cktrees are candidates for traversal:

- Nodes in other cktrees that have been marked as achieved during the exploration of the cktree corresponding to g when exploring other alternatives, but not when exploring the current one. The achievement cost of those nodes is relevant to finding the best alternative. The information about the other alternatives is captured in $data_{other}$. In particular these relevant nodes can be obtained from the nodes stored in **marked_goals**($data_{other}$) (Step 4).
- Nodes in other cktrees that have been deleted by the current alternative. Their reachievement cost must be considered. These nodes can be obtained from the nodes stored in **deleted_goals**($data$) (Step 7).

For each of these nodes in other cktrees the matcher tests whether they would actually correspond to subgoals in the current problem (Steps 5 and 8). Recall that the achievement links that pointed to those nodes were recorded from previous planning experience in which the additions and deletions actually occurred (cf Section 4.5.5). Whether the achievements will occur in the

update_alt($ckroot, cost, data, data_{other}$)

1. $marked\_goals \leftarrow$ **marked_goals**($data$)
2. $deleted\_goals \leftarrow$ **deleted_goals**($data$)
3. $\mathcal{G} \leftarrow \emptyset$
4. for each $<g_1, g_2, \beta> \in$ **marked_goals**($data_{other}$) s.t. **in_other_cktree_p**($g_1, ckroot$)
5. if $\neg$**wont_be_subgoal_p**(g_1, β)
   then
6. $\mathcal{G} \leftarrow$ push($<g_1, g_2, \beta>, \mathcal{G}$)
7. for each $<g_1, g_2, \beta> \in$ **deleted_goals**($data$) s.t. **in_other_cktree_p**($g_1, ckroot$)
8. if $\neg$**wont_be_subgoal_p**(g_1, β)
   then
9. $\mathcal{G} \leftarrow$ push($<g_1, g_2, \beta>, \mathcal{G}$)
10. $\mathcal{G} \leftarrow$ **prune_common_subtrees**($\mathcal{G}$)
11. for each $<g, g_2, \beta> \in \mathcal{G}$
12. $<alt, cost_g> \leftarrow$ **match_cktree**(g, β,**alt**($data$))
13. $cost \leftarrow cost + cost_g$
14. return($cost$)

Figure 4.57: Traversing other cktrees to update the estimate for the current alternative.

wont_be_subgoal_p(g, β) if one of these is true:

- g belongs to a subtree that will not be explored because neither g nor one of g's ancestors, instantiated with β, is a pending goal
- g will not be explored because one of g's ancestors is marked as achieved (in *marked_goals*)

Figure 4.58: Checking whether the goal corresponding to a node will actually become a goal in the current problem.

actual planning problem depends on how the variables in those nodes are instantiated, given the current problem goals and state objects. Figure 4.58 summarizes the criteria to discard goal nodes that will not correspond to actual subgoals in the current problem and therefore should not be used to increase the estimated cost of the current alternative. The test checks whether the nodes belong to subtrees that will not be explored because their roots do not correspond to goals that would become actual subgoals at planning time. For the node to become a subgoal first the instantiation of the node in the current problem (β in the figures) should be the same as that of the adder/deleter. Second, if p is an ancestor of q, and p is true or marked as achieved,

q will not become a subgoal and therefore the nodes in the subtree rooted at q should not be explored.

Once all the relevant nodes $\mathcal{G}$ have been determined, any node q in $\mathcal{G}$ that has an ancestor p also in $\mathcal{G}$ is discarded (Step 10 of Figure 4.57), as the cost of achieving q will be computed as part of the cost of achieving p. Finally the cktree matcher is called for each remaining node in $\mathcal{G}$ (Steps 11-13) and the resulting cost is added to the initial estimate (the one based only on the cktree for g). The total cost is returned.

4.5.9.4 Matching the Same Cktree for Other Problem Goals

When the cktree matcher updates the estimate of an alternative by looking at other possibly interacting goals, the case in which the other goals correspond to the same cktree as the current goal is dealt with separately (Step 2 of Figure 4.56). Figure 4.59 describes this case. An example of this case is the problem of drilling two spot holes on a part, that is, the goal is the conjunction of two instances of `has-spot`. The process described in the previous section to determine which parts of the other cktrees are relevant cannot be used here, since it actually is the same cktree. Here the achievement links of a node can be seen as pointing to the node itself. In this case the whole cktree is candidate for exploration again, but with the bindings corresponding to the other goal(s) (Steps 1 and 4-5) and considering the nodes visited in the initial exploration of the cktree as the *marked_goals* (which therefore will be assigned cost 0) (Steps 2-3).

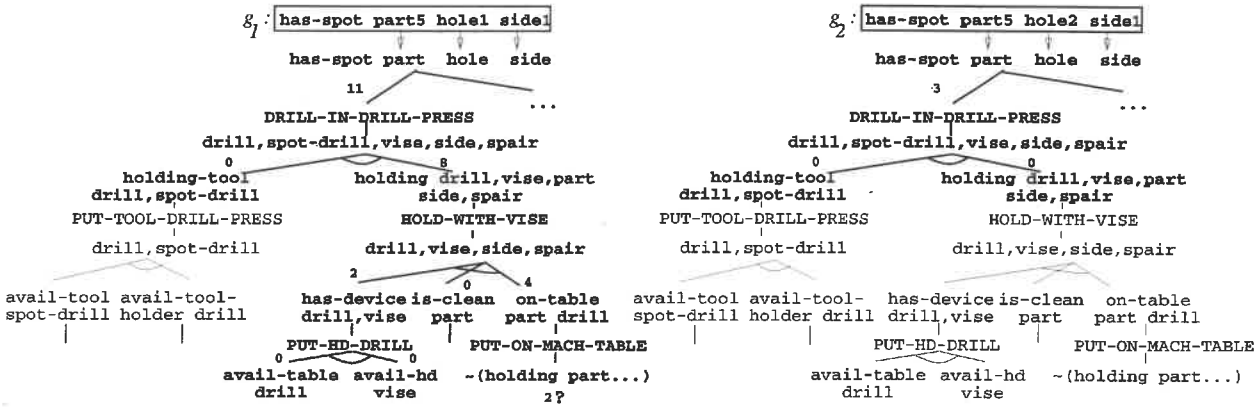
---

update_with_same_cktree_goals($g_0, ckroot, cost, data$)

1. $\mathcal{G} \leftarrow \{g \in top\_level\_goals - \{g_0\}$ s.t. **predicate**(g_0) = **predicate**(g)$\}$
2. $marked\_goals \leftarrow$ **visited_cknodes**($data$)
3. $deleted\_goals \leftarrow$ **deleted_goals**($data$)
4. for each $g \in \mathcal{G}$;; i.e. correspond to the same cktree
5. $\quad \beta_0 \leftarrow$ **bnds**(g,**name**($ckroot$))
6. $\quad$ for each $op \in$ **children**($ckroot$)
7. $\quad\quad <alt_0, cost_0> \leftarrow$ **match_cktree**(op, β_0, β_0)
8. $\quad\quad <alt_{best}, cost_{best}> \leftarrow$ **match_all_alts**($op, \beta_0, alt_0, cost_0$)
9. $\quad cost \leftarrow cost + cost_{best}$
10. return($cost$)

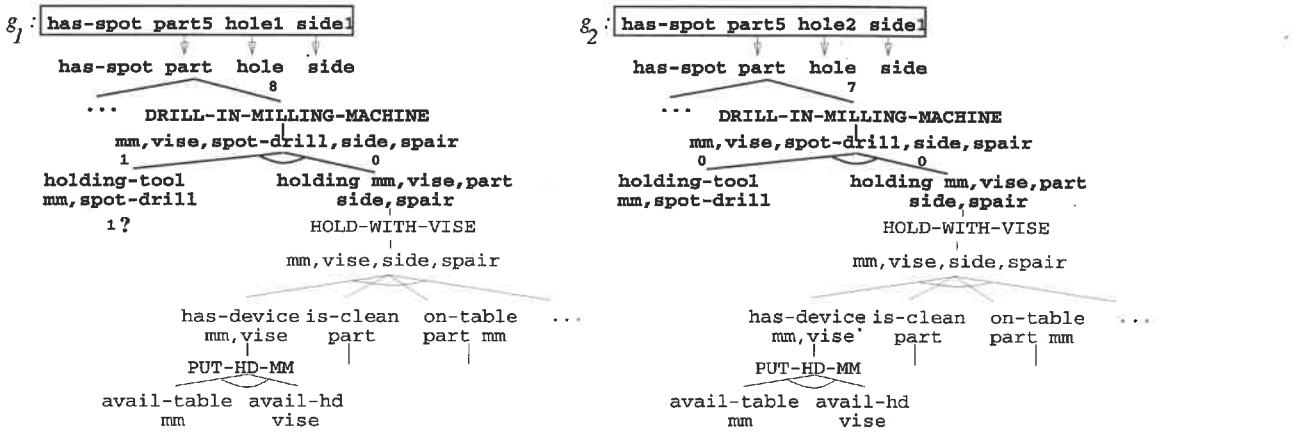
Figure 4.59: Traversing the same cktree again with an instantiation corresponding to a different goal.

Figure 4.60 shows how the cktree for `has-spot` introduced at the beginning of this section (Figure 4.36) is used as control knowledge to solve the problem of drilling two spot holes



Estimate for **DRILL-IN-DRILL-PRESS** = 11 + 3 = 14

Figure 4.60: Using the `has-spot` cktree of Figure 4.36 as control knowledge for the problem of drilling two spot holes on the same side of a part. The figure shows how the cost of the first alternative, *drill-in-drill-press*, is estimated.



Estimate for **DRILL-IN-MILLING-MACHINE** = 8 + 7 = 15

Figure 4.61: Using the `has-spot` cktree to estimate the cost of the second alternative, *drill-in-milling-machine*. In this example the quality metric assigns cost 7 to that operator.

`hole1` and `hole2` on the same side of a part. In this example the part is being held on the milling machine and the available spot drill is ready in the drill press. There is a second, free, holding device available on the drill press. The quality metric used assigns values 3 and 7 to operators *drill-in-drill-press* and *drill-in-milling-machine* respectively. Assume that the planner starts working on goal g_1, `(has-hole part5 hole1 side1)`, and the cktree matcher is invoked to suggest an operator to achieve that goal. The left side of the figure indicates the estimated cost. The nodes in parts of the cktree not explored by the cktree matcher are shown in a lighter font. Then **update_with_same_cktree_goals** is called with argument $cost = 11$. In Step 1 the set $\mathcal{G}$ contains the second goal g_2 `(has-hole part5 hole2 side1)` and the matcher uses the same cktree to estimate the cost of achieving that goal. This is shown on the right-hand side of Figure 4.60. Note that the `holding` and `holding-tool` subtrees have cost 0 because they were visited in the first pass and now are part of *marked_goals* (Step 1 of **update_with_same_cktree_goals**). That is, the part and tool are set only once for both holes. The final estimate is obtained by adding the results of both traversals, namely 11 and 3, as shown at the bottom of the figure. The cktree matcher then proceeds to estimate the cost for the second alternative for g_1, namely *drill-in-milling-machine*. Figure 4.61 describes this step.

The interesting point here is that should the matcher have considered only the first goal g_1, it would have suggested using the milling machine (with value 8 for g_1 only, versus 11 for drilling in the drill press, as shown on the left sides of Figures 4.60 and 4.61). However considering the presence of a second `has-spot` goal and exploring the cktrees appropriately, the correct choice was made, namely using the drill press. The cktree matcher suggests that choice for g_1 and stores the choice of operator and bindings for g_2, which that will be used when the planner subgoals on it. Table 4.5 shows the *real* cost of solving the 1- and 2-goal problems with each of the alternative operators. The real costs in the case of the milling machine are higher than the estimates due to the lack of knowledge about achieving `holding-tool` in the cktree.

Table 4.4 at the beginning of this chapter presented similar examples to motivate the usefulness of the cktree formalism. The control knowledge needed to make the correct choice in the example presented is not easily captured in the form of control rules, especially when the number of goals is generalized.

Total Plan Cost	drill press	milling machine
1 spot hole	11	**10**
2 spot holes	**14**	17

Table 4.5: Comparative quality of the plans to drill one and two spot holes depending of the machine used. The quality metric used assigns values 3 to drilling with the drill press and 7 to drilling with the milling machine. The numbers in bold face indicate the best plans.

4.5.10 Using the Same Cktrees for Different Quality Metrics

The cktrees by themselves do not capture a particular quality metric. The quality metric is parameterized and stored independently. It is the cktree matcher the one that uses the quality metric to find the corresponding value for the individual instantiated operators, and uses it to estimate the quality of the alternatives available (Step 28 of Figure 4.42). Therefore if the metric changes the same learned cktrees can still be used as control knowledge for the domain. Thus the cost of learning them can be amortized over the use of different quality metrics as the learned knowledge is robust to changes in the metric. (This was not the case with the control rules learned in Chapter 3.)

On the other hand the learned knowledge, in particular the choice of which nodes to add to a cktree, depends on the training episodes, that is, the problems in which the existing control knowledge lead to a suboptimal plan. And this in turn depends on the quality metric. If the planner's default choices lead to good plans, learning is not invoked. Therefore if not much is learned for a given metric, when the metric changes not much knowledge can be reused. An extreme example is to use at training time a metric that assigns cost 0 to all operators. Therefore any plan is as good as the others and no learning is needed. However if the metric changes no reuse across metrics is possible.

Sections 4.6 and 4.8.4 show how the cktrees are successfully reused with different quality metrics in a transportation domain and in the process planning domain respectively. Section 4.7.1 justifies the use of different quality metrics.

4.5.11 Goal Ordering Control Knowledge

In the discussion of the previous sections we have not mentioned goal choices. The current cktree matching algorithms only provide guidance for operator and bindings choices. They do not deal with choosing the next goal to work on at some point. In our implementation we have used the goal ordering control rules learned with the algorithms described in Chapter 3. Those rules have served our purposes and worked well in the experiments, slightly increasing the quality-improvement performance obtained by using the cktrees alone. Note that they were learned from a more restricted class of quality metrics, as discussed in Section 3.12.

4.6 An Example in a Transportation Domain

The previous sections have described in detail the use of control knowledge trees as a formalism to represent search control knowledge. The algorithms for learning and using cktrees have been illustrated with examples from the process planning domain and an artificial domain. In this

section we demonstrate the generality of the approach by illustrating the usefulness of the cktrees in the transportation domain introduced at the beginning of this chapter. Figure 4.1 and Table 4.2 respectively showed the domain description and a quality metric for the domain. This domain is different from the process planning domain in two aspects related to plan quality. First the value assigned by the metric to an operator depends not only on the operator but also on the operator bindings. Second, the quality of a plan does not depend so much on the ability to share the work among operator subgoals, but on finding a good route to send the packages given the different cost of moving the packages depending on the kind of transportation and the distance traveled.

```
 2 n2 (done)
 4 n4 <*finish*>
 5   n5 (at-obj package1 new-hampton)
 7   n7 <unload-van package1 ebro new-hampton> [1]
 8     n8 (inside-van package1 ebro)
10     n10 <load-van package1 ebro monroeville> [3]
11     n11 <LOAD-VAN PACKAGE1 EBRO MONROEVILLE>
12     n12 (at-van ebro new-hampton)
14     n14 <drive-van ebro monroeville new-hampton> [1]
15     n15 <DRIVE-VAN EBRO MONROEVILLE NEW-HAMPTON>
15   n16 <UNLOAD-VAN PACKAGE1 EBRO NEW-HAMPTON>

Solution:
  1. <load-van package1 ebro monroeville>
  2. <drive-van ebro monroeville new-hampton>
  3. <unload-van package1 ebro new-hampton>

cost = 1485
```

Figure 4.62: Initial solution and problem solving trace obtained by the planner for the problem in Figure 4.2.

Assume the planner is given the problem in Figure 4.2 and the quality metric in Table 4.2. Figure 4.62 shows the the initial plan obtained by the planner, and the trace of that planning episode. Figure 4.63 shows the improved plan suggested by a human expert, and the trace generated (or sequence of decisions made) by the planner in order to obtain that improved plan. Note that the crucial decision that distinguishes the two plans is the choice of bindings for the location for operator *load-van* at `n12` (`monroeville`) and `n14` (`philly`). From these two plans, the plan trees are built and the cktree in Figure 4.64 is learned. The figure shows the constraints on the operator instantiations that were learned during the construction of the cktree. These constraints greatly reduce the space of alternatives explored by the cktree matcher.

The learned cktree can be used to provide guidance in a variety of problems. First, the cktree provides the correct guidance in spite of changes in the quality metric. Table 4.6 shows the quality values of the two plans for a range of quality metrics. The metrics differ on the cost of the *drive-van* operator. The purpose of the table is to show how different quality metrics require that the cktree suggests different choices. Using the cktree in Figure 4.64, learned from the

```
  1. <load-van package1 ebro monroeville>
  2. <drive-van ebro monroeville pittsburgh>
  3. <unload-van package1 ebro pittsburgh>
  4. <load-train package1 loco pittsburgh>
  5. <ride-train loco pittsburgh philly>
  6. <unload-train package1 loco philly>
  7. <load-van package1 avia philly>
  8. <drive-van avia philly new-hampton>
  9. <unload-van package1 avia new-hampton>
cost = 480

  2 n2 (done)
  4 n4 <*finish*>
  5   n5 (at-obj package1 new-hampton)
  7   n7 <unload-van package1 ebro new-hampton> [1]
Op was not in the solution proposed by the expert.
Backtracking to make new binding decision at node 7.

  6   n6 unload-van
  7   n9 <unload-van package1 avia new-hampton>
  8     n10 (inside-van package1 avia)
 10     n12 <load-van package1 avia monroeville> [3]
Op was not in the solution proposed by the expert.
Backtracking to make new binding decision at node 12.

  9     n11 load-van
 10     n14 <load-van package1 avia philly> [2]
 11       n15 (at-obj package1 philly)
 12       n16 unload-van
Op #<OP: UNLOAD-VAN> was not in the solution proposed by the expert.
Backtracking to make new operator decision at node 16.

 13       n19 <unload-train package1 loco philly>
 14         n20 (inside-train package1 loco)
 16         n22 <load-train package1 loco pittsburgh> [1]
 17           n23 (at-obj package1 pittsburgh)
 19           n25 <unload-van package1 ebro pittsburgh> [1]
 20             n26 (inside-van package1 ebro)
 22             n28 <load-van package1 ebro monroeville> [3]
 23             n29 <LOAD-VAN PACKAGE1 EBRO MONROEVILLE>
 24             n30 (at-van ebro pittsburgh) [2]
 26             n32 <drive-van ebro monroeville pittsburgh> [1]
 27             n33 <DRIVE-VAN EBRO MONROEVILLE PITTSBURGH>
 28           n34 <UNLOAD-VAN PACKAGE1 EBRO PITTSBURGH>
 29         n35 <LOAD-TRAIN PACKAGE1 LOCO PITTSBURGH>
 30         n36 (at-train loco philly) [1]
 32         n38 <ride-train loco pittsburgh philly>
 33         n39 <RIDE-TRAIN LOCO PITTSBURGH PHILLY>
 34       n40 <UNLOAD-TRAIN PACKAGE1 LOCO PHILLY>
 35     n41 <LOAD-VAN PACKAGE1 AVIA PHILLY>
 36     n42 (at-van avia new-hampton)
 38     n44 <drive-van avia philly new-hampton> [1]
 39     n45 <DRIVE-VAN AVIA PHILLY NEW-HAMPTON>
 39   n46 <UNLOAD-VAN PACKAGE1 AVIA NEW-HAMPTON>
Achieved top-level goals.
```

Figure 4.63: The solution provided by the human expert and the trace generated in order to obtain that plan.

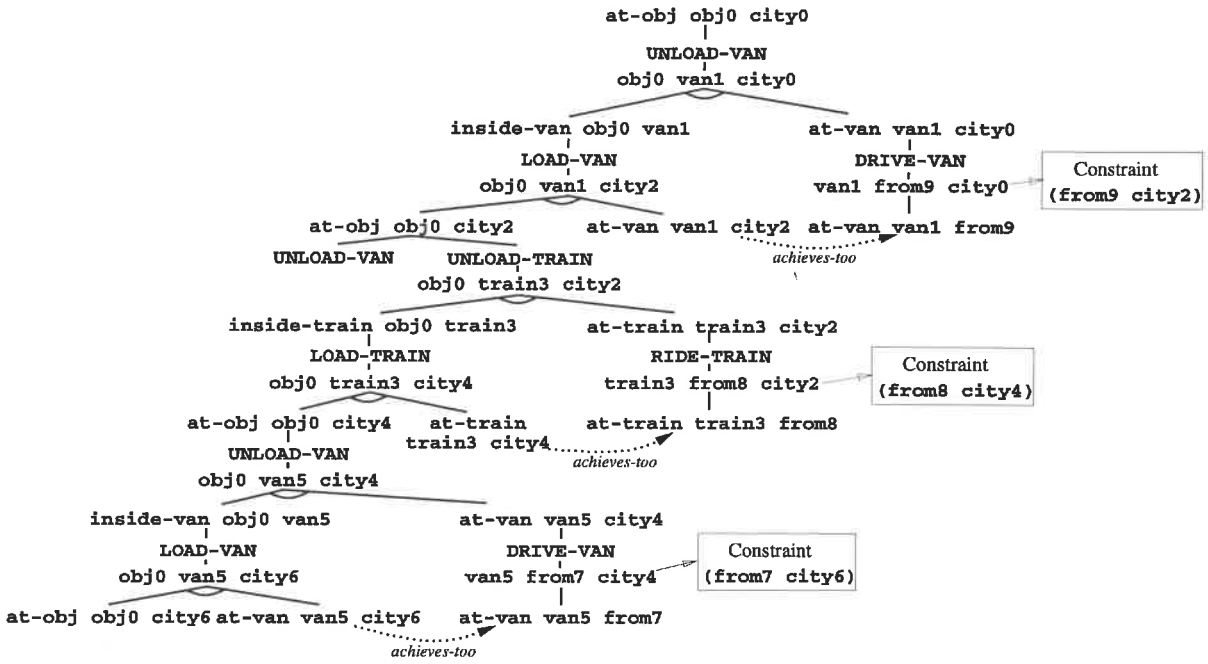


Figure 4.64: Control knowledge tree learned for the problem in Figure 4.2.

initial example and quality metric, the planner found the best plan in all the cases enumerated in the table. This result shows the robustness of the learned knowledge to changes in the quality metric, a desirable property in many domains.

Total Plan Cost	cost(drive-van(van,x,y))			
	5×d(x,y)	1.25×d(x,y)	1.2×d(x,y)	d(x,y)
Solution 1	1485	378	**364**	**305**
Solution 2	**480**	**367**	366	360

Table 4.6: Cost of two alternative plans for the problem of Figure 4.2 for several quality metrics. Bold face indicates the better plan in each case. The metrics differ on the cost of the *drive-van* operator. The planner was able to find the best plans in all the cases using the cktree learned from the 1-package problem and the initial quality metric (Figure 4.64).

The cktree of Figure 4.64 also provided correct guidance for a problem in which two packages need to be moved. Figure 4.4 outlined two plans for this problem. Which of the two plans is better depends on the particular metric used. Table 4.7 shows the quality of the plans against different quality metrics. Using the cktree learned for the 1-package problem with the first metric, the planner successfully obtained the best plan for all the metrics showed in the table.

Figure 4.5 plotted plan quality values for a variety of quality metrics as a function of the number of packages in the problem. PRODIGY4.0 is able to solve the problems corresponding to all the

Total Plan Cost	cost(drive-van(van,x,y))			
	5×d(x,y)	1.3×d(x,y)	1.25×d(x,y)	d(x,y)
Solution 1	1495	403	**388**	**315**
Solution 2	**510**	**399**	397	390

Table 4.7: Cost of two alternative plans for the 2-package problem for several quality metrics. The metrics differ on the cost of the *drive-van* operator. The planner was able to find the best plans in all the cases using the cktree learned from the 1-package problem (Figure 4.64).

cases in that figure (that is, varying the quality metric and the number of packages in the goal) and obtain the plan of better quality in each case.

As Section 4.5.11 discussed the control knowledge tree formalism does not currently address goal choices. In the experiments we have made use of a set of simple hand-coded goal preference control rules. Figure 4.65 shows one of these rules.

```
(control-rule ACQ4
  (if (and (candidate-goal (at-train <train> <loc>))
           (candidate-goal (at-obj <package> <other-loc>))))
  (then reject goal (at-train <train> <loc>)))
```

Figure 4.65: A goal preference control rule for the transportation domain.

We also experimented with a different representation of the domain, in which several objects may be loaded (or unloaded) with a single operation. In this case the number of packages is a parameter of the operator, and the cost of the load and unload operators depends on the number of packages loaded. Learning and using control knowledge trees also succeeded in generating the better quality plans on this version of the domain.

4.7 Discussion

This section discusses some of the properties of control knowledge trees, and of the mechanisms described to use them to represent quality-enhancing control knowledge. Some of the characteristics of the control rule learning algorithms described in Section 3.12 are shared by the cktree learning algorithms. In both cases **learning is triggered by failure**, when the available control knowledge does not lead to a good enough plan according to the quality metric and a human expert. Both approaches take advantage of feedback from a human domain expert in the form of improvements to an initial plan. Still the approaches differ along several important dimensions. The limitations of using control rules motivated the development of control knowledge trees. The following subsections explain those differences and other characteristics

of the control knowledge tree approach, including accuracy of the learned knowledge, and issues on efficiency and tradeoffs between plan quality and planning efficiency.

4.7.1 Using Cktrees versus Control Rules as a Control Knowledge Representation Formalism

This section presents a comparison of the characteristics of the algorithms for learning control rules and control knowledge trees described in thesis. The learned quality-enhancing control rules provide effective guidance when the quality metric does not require reasoning about complex global tradeoffs. They are highly operational and are efficiently used at planning time (Section 3.13.3.2). The performance in improving plan quality of the learned control knowledge trees is equivalent to that of control rules for simpler non-interacting situations, and superior for more complex interactions and tradeoff situations. However using control knowledge trees is computationally more expensive and may reduce planning efficiency, as the results reported in Section 4.8.3 will indicate. In addition control knowledge trees do not provide goal ordering control knowledge. In some of our experiments we have used them together with the goal preference rules learned by the first method achieving a synergistic effect. An important advantage of control knowledge trees is their robustness to changes in the quality metric, while the learned control rules are quality metric specific, are invalidated if the metric changes, and must be relearned. Note that the characteristics of both formalisms are somewhat complementary. An interesting issue, not explored in this thesis, is the design of an architecture in which both representations are combined, for instance by using simple, local control rules when the distinctions among choices are clear, and only global control knowledge trees when needed. (Related issues are explored in [Simmons, 1988a, Goodwin, 1994]). The rest of this section elaborates on some of the characteristics of the two formalisms.

Less constrained quality metrics. Section 3.12 pointed out the restrictions on the class of quality metrics suitable for our algorithms for control rule learning. In particular the algorithms relied on finding ways to share the work among parts of the plan by suggesting operator, binding, and goal alternatives that lead to common subgoals among the plan operators. Section 4.1 described how the extension of that class of metrics to account for differences in the quality of the alternative operators, and the tradeoffs originated between the quality of the operators themselves, and the cost of achieving those operators preconditions, required more complex control knowledge than the one learned by the previous algorithms. The need to capture such tradeoffs, that is, information which is more global in nature, prompted our development of the cktree learning algorithms.

The quality metrics suitable for cktree learning are still constrained to be linear in the cost of the plan operators. This limitation is due to the mechanisms used to assign costs to the nodes

of the plan trees and to match the cktrees. The cost of a node depends only on the cost of its children, and on the node itself if it is a binding node. The metrics are monotonically increasing in the cost attached to the subgoals in the plan tree, that is, the cost attached to a cktree node by the cktree matcher must be greater or equal than the cost attached to its children. The cktree matcher makes use of this property to produce estimates of the quality of each alternative.

Reusing cktrees across different quality metrics. The control rules learned by the algorithms in Chapter 3 capture in an implicit manner the quality metric for which they were built. Therefore if the metric changes, the rules may give incorrect preferences and need to be discarded and relearned. In contrast the cktrees do not capture a particular quality metric; instead the metric is parameterized. The cktree building algorithms do not use the quality metric. The metric is only used at cktree matching time. In general a change of the quality metric may lead to different planning decisions. The cktree matcher will suggest the appropriate decisions by using the current quality metric as it traverses the cktree and estimates the quality of each available alternative. Reusing the same control knowledge, i.e. the cktrees, when the quality metric changes amortizes the cost of learning along a larger number of problems in which that knowledge is useful. Section 4.5.10 further discusses cktree robustness to changes in the metric. Section 4.8.4 presents an empirical demonstration of the reuse of the learned cktrees across different quality metrics.

There are a number of situations in which the existence of control knowledge for a domain with a variety of quality metrics may prove useful. This list is not exhaustive but just suggests some of those situations. First, the planner can output multiple alternative plans according to different criteria captured in the quality metric, and let a human expert choose one or a combination of them. Section 3.12 suggested how this can be useful in mixed-initiative systems. Second, the quality metric may evolve over time or vary according to some criteria. For example, the cost of driving may vary in summer or winter; in factory scheduling the weight of different factors (e.g. tardiness) may be different depending on when the action is scheduled. Third, the quality metric may also change if it is being learned, possibly from the human expert, as the planner solves new problems. Learning control knowledge to capture a variety of quality metrics is an instance of the lifelong learning framework described in [Thrun and Mitchell, 1994]. In lifelong learning the system encounters a collection of related learning problems over its lifetime, and it may employ knowledge gathered in previous tasks to improve its performance.

Global guidance versus local guidance. The control rule learner studies the local decisions of the planning episode and produces rules. Those rules are used to make individual, local decisions at planning time. By contrast the cktree learner interprets and stores the relevant parts of the complete planning episode. The learned cktree is used to provide global guidance, that is, it can suggest a sequence of planning decisions, the combination of which will produce a good quality plan. Since control rules capture local decisions their number may grow with

the corresponding increase in matching cost, since matching must occur at all the decision points for which the planner must make a choice. On the other hand the effort of matching a cktree is amortized over the sequence of decisions for which a single matching episode provides guidance. Also the cost of matching may be made dependent on the *difficulty* of the decision, that is, on how close in quality the alternatives at the decision point are. The third example of Section 4.5.1 illustrated how the cktree matcher stops when it finds that the current alternative will be worse than the best so far. In the case of control rules all the rules available at each point are matched independent of the the difficulty of the choice.

Partial match versus total match. Control rules provide search guidance only when they match the current situation completely, that is, when the conjunction of preconditions in their left-hand side is satisfied in the planner's current state. Section 4.1.3 described the limitations of this total match with some examples. On the other hand a cktree provides guidance even when the information to estimate the alternatives in the current situation is incomplete, and only partially matches the current situation.

Partial match and global guidance are also characteristics of systems that store knowledge as a library of cases, as in the case of PRODIGY/ANALOGY [Veloso, 1994]. In those systems each relevant planning episode is parameterized and stored as an individual case. When confronted with a new similar problem the relevant case(s) is retrieved, instantiated, and possibly adapted to solve the new problem. A similarity metric decides which case should be retrieved, i.e. it measures how relevant a case is for the new problem. On the other hand, the cktree learner determines first which parts of the planning episode are relevant from the plan quality point of view (by looking at the decisions in which the initial plan and the improved plan differ). The cktree learner does not store each planning experience in a new structure (cktree) but merges it with the existing cktree(s). Therefore the cost of retrieval (finding the relevant cktree) is minimal. The cktree matcher only explores the relevant parts of the cktree, as the case replay mechanism is able to skip the irrelevant parts of the case being used.

The purpose of using cktrees is instead to lead the planner towards good plans. The cktree exploration (matching) algorithm is geared to that purpose. Generally cases in the context of planning systems have been successfully used to reduce the planner's search space. A case-based planner that would aim at generating good quality plans should be able to retrieve a case that leads to a good plan, according to the quality metric, for the current problem. Therefore some knowledge on estimating the quality of the case for the current problem should be captured by the similarity metric. Note that the case may need to be adapted to the current problem by adding or removing steps from it, or maybe merging it with other cases. Therefore estimating the quality of the suggested case should account also for the changes in quality due to the steps needed for the adaptation.

Cktrees and PSGs. Control knowledge trees bear similarities with the Problem Space Graphs (PSGs) developed by Etzioni [Etzioni, 1990]. PSGs represent all the possible paths in a backward-chaining search through a problem space. They are derived from the domain definition by unfolding or partial evaluation. A PSG is a directed, acyclic and/or graph, where nodes corresponding to goals are connected, via OR-links, to the operators that match it. The operators are partially instantiated by variable substitution and connected via and-links to their partially instantiated preconditions. The PSGs are built statically from the domain definition, that is, they do not require any actual planning episode. The PSGs are then used to compute the conditions under which subgoaling on each node would lead to a failure and would force the planner to backtrack. These conditions become the antecedents of control rules that guide the planner's choices, thus pruning the search space and *improving planning efficiency* by reducing the number of search nodes visited. The PSG traversal and control rule building algorithms are geared towards that aim, that is, finding proofs of success and failure. This is fundamentally different from finding the relevant reasons why a decision leads to *a good quality plan.*

The cktree building algorithm parameterizes the nodes found in the plan tree in a way similar to the partially instantiation used by the PSGs building algorithm. The two formalisms differ in some fundamental aspects. First, the PSGs are built statically, while the cktrees are built by translating a current planning episode which is stored in the plan trees. Etzioni [Etzioni, 1990] described (and demonstrated in the STATIC system) how the PSGs are built upfront, without any planning experience, by exploring the complete problem space. If the domain is large learning time is long. In addition the planner may suffer of the utility problem: many of the control rules generated may be never used if the problem distribution is not uniform. DYNAMIC [Pérez and Etzioni, 1992] showed how the use of planning experience may improve the performance of a PSG-based learner by pointing out the relevant parts of the problem space that the static analyzer (or PSG builder) should explore. For example, DYNAMIC only analyzes goal interactions that occur in training examples. Still once those parts are determined the PSGs are built without further reference to the planning episode. On the other hand building the cktrees based on planning experience increases cktree matching efficiency, because only the parts of the cktree (operator alternatives and goal interactions) that have been used in the past (either by searching with the existing knowledge or by using the expert's plans) are actually built and therefore explored at cktree matching time. Also, the cktree learning algorithm avoids building recursive explanations by limiting the depth of the explanation to that in the episodes seen so far. The cktree matching algorithm allows partial match and can use experience learned from an episode in which the depth of the recursive chain was different from that in the current problem. The cktrees could be built statically which could lead to high quality performance without using training examples, at the cost of losing cktree matching efficiency and increasing learning time.

The second crucial difference between the two formalisms is on how they are used. The purpose of building the PSGs is to translate them into control rules. On the other hand the cktrees are

used directly at planning time to provide guidance. We have discussed the advantages of doing so when trying to capture *plan quality* control knowledge. The third crucial difference has already been mentioned and concerns the purpose of the learning mechanism. PSGs are built to reduce the search space and so improve planning efficiency. They fall into the category of speed-up learning systems. The PSG traversal algorithms are designed with this performance goal in mind, as discused above. In contrast cktrees are built and explored to produce good plans given some quality metric that is used in the cktree matching process. The cktree learning and matching algorithms described in this chapter were designed for this performance goal.

Learning effort. The control rule learning algorithm of Chapter 3 invests effort deriving rules from the plan trees by explaining the differences in quality between the initial and improved plans. On the contrary the cktree learning algorithm spends a smaller effort at learning time, as it translates only the relevant parts of the plan trees into cktrees or subtrees that are incorporated to the existing cktrees. The rule learner can be characterized as an eager learner, while the cktree learner is a lazier one. The learning effort by the cktree learner is more comparable to that of PRODIGY/ANALOGY [Veloso, 1994] which stores annotated traces of planning episodes. These derivational traces are ellaborated further on an as-needed basis when the planner confronts new similar problems.

4.7.2 Efficiency Issues in Using Cktrees

Section 4.5 described the cktree matching algorithm. We now discuss some efficiency issues in that algorithm. In the worst case the number of nodes and alternatives explored by the cktree matcher can be as high as those explored by exhausting the whole search space of planning. However even in that worst case, exploring alternatives in the cktree is faster than in the planner's space: the cktree is already built and can be traversed fast, while the planner must build each node explored, with computational overhead in time and space. In addition the cktree matching algorithm uses several techniques to reduce the number of cktree nodes explored.

First, the cktree matcher explores only the parts of the cktree that are relevant to the decision for which guidance is sought. If several goals need to be achieved, the links between their corresponding cktrees determine whether the goals are independent. If they are, the matcher only considers the cktree for the current goal and ignores the others. A choice at that point leading to a good plan is independent of choices for other goals (given the available knowledge, which can be incomplete if no examples of an interaction have been seen already), given the characteristics of the quality metric (linear on the cost of the individual operators). By contrast, if the planner were to explore the complete space to find the best plan, in the absence of appropriate control knowledge, it would try all the possible ways of interleaving those

independent goals, even though those interleavings do not affect the quality of the resulting plan. In the case in which the goals are not independent, the number of cktree nodes explored is reduced by exploring only the relevant parts of the cktrees for the other goals, as described in Section 4.5.9.4.

Second, the cktree matcher is not invoked at every decision point. The cktrees are only used to provide guidance for a decision if the planner's choice for a similar decision was overriden to obtain an improved, expert-given plan in a past planning episode. In that episode learning was triggered and as a consequence one of the learned, currently existing cktrees is relevant to guide the current decision. For any given decision, if no cktree is relevant the planner chooses the default alternative.

Third, the previous section discussed how when the cktree is invoked at a decision point, it may generate global guidance, that is, guidance for other decisions that the planner will encounter later on in the search. For example, it may suggest both an operator to achieve a goal, and an instantiation of that operator. It may suggest as well operators to achieve the new subgoals. In this way matching cost is amortized along a number of planning decisions.

The cktree matching algorithm includes a number of other efficiency improvements that reduce the space of alternatives and cktree nodes explored. We review them here.

- Section 4.5.8 described how multiple alternatives can be explored efficiently by reusing the estimated costs of subtrees, if the matcher determines that those estimates do not change between alternatives. The nodes whose estimate may change are marked and their new estimate computed. On the other hand, compare this with the planner finding a good plan by exhaustively exploring the search space. When the planner backtracks to try a new alternative, it discards the whole search tree corresponding to it. Then although successive alternatives may require to expand some subtree repeatedly, the planner does not take advantage of it and repeats the work instead, as it backtracks to the point where the alternative was introduced and discards the search tree under it.

- Sections 4.4.2.4 and 4.5.4 explained how constraints on the bindings that an operator may take were learned and used. The purpose of those constraints is to make cktree matching more efficient by pruning out bindings that would lead to lower quality plans. Those constraints are stored in cktree binding nodes. The constraints stored in a node are used when the cktree matcher is exploring the node and generating bindings for the variables introduced by the node. The constraints reduce the set of possible instantiations to those that will cause sharing of subgoals with other operators in the cktree, therefore leading to a better quality plan. If no alternative satisfies the constraints, all the alternatives are explored.

- At each point during cktree matching the matcher maintains the best alternative found so far, and its estimated cost. If the estimate for the current alternative exceeds the best so

far, matching for the current alternative stops and the alternative is discarded. Discarding an alternative whose cost exceeds the best so far is possible because of the linear nature of the quality metric, which will monotonically increase the value of the estimate as more nodes are visited. Stopping when a limit is reached leads to savings in matching time that depend on how close in quality the alternatives are. If one alternative is far superior to the others and it is explored early on, the matcher will not invest a large effort on exploring the other alternatives as it will quickly find that they will be worse. The third example in Section 4.5.1 clearly illustrated this point. Additionally the matcher always has available an alternative, the best one so far, should it run out of time to make a decision. Note however that if an anytime solution is needed, the planner should allocate additional time to complete the search for the plan after that decision is made. Another limit to the matching effort could be set by putting a bound on the depth up to which the cktree is traversed. Once that depth bound is reached, the cost of the subtree below the current node could be estimated in some fixed way (that is, without exploring the subtree further) and backed up the cktree. The properties, regarding the accuracy of the estimate, of using such bound would vary from domain to domain. We have not implemented it.

Traversing the cktree to estimate the cost of an alternative is a search problem. The algorithm described is one way to search the cktree. Other traversal algorithms are possible. In particular the cktree may be traversed by associating a range (interval) with each node, that is, an upper and a lower bound on the cost estimate for the subtree rooted at the node, and then successively refining the estimates associated with the nodes: as new nodes are visited in the subtree their estimates can be backed up and the range of the root node may be refined until, if necessary, it converges in a single value. The traversal may stop earlier though, when the upper-bound of an alternative is no worse (higher) than the lower bound of any of the other alternatives. An example of such algorithms can be found in [Berliner, 1979]. The DRIPS planner [Haddawy and Suwandi, 1994] employs intervals to represent the expected utility of an abstract plan. The interval includes the expected utilities of all possible instantiations of that abstract plan. The plan is built by refinement, that is, instantiating one of the actions, which tends to narrow the interval. Again, when the intervals (which corespond in this case to abstract plans) do not overlap, the plan with the worst interval can be eliminated.

An interval-based search algorithm could be useful to implement cktree traversal. The upper and lower bounds on the subtree cost estimates can be recorded in the cktree nodes from previous planning experience. However some characteristics of the nature of the cktrees, and of the problems we are interested in, may limit the usefulness of the approach. Note that the search tree traversed by the search algorithm is not the cktree itself, but a tree with a higher branching factor: each binding node can be instantiated in multiple ways and the cktree matcher explores (a subset of) them. The intervals would enclose the range of estimated quality values for all the possible instantiations. Examples throughout this chapter showed how the choice of bindings may lead to very different quality values and therefore the intervals may be quite wide. Also, in

those examples, with the quality metrics used, different alternatives, e.g. operator alternatives, are usually close in quality; this means that the intervals will overlap heavily. Distinguishing among alternatives may require to traverse nodes close to the cktree leaves. Last, we have seen examples of how the choices of alternatives for the planning decisions are not independent; the dependences are captured by the achievement and deletion links. Traversing those links is needed to accurately estimate the quality of the alternatives. To our knowledge DRIPS assumes that the subgoals are independent. We have presented examples in which this is not the case, and how the cktree traversal algorithms described are able to handle them. An interval-based cktree matcher would have to deal with these cases. In summary, in Section 4.1.3 we argued that our research has concentrated in cases in which the control knowledge must make tight distinctions, and how those considerations lead to our development of control knowledge trees. In such cases the search reduction obtained by interval search may be limited. Still in some cases implementing cktree traversal using interval search shows promise and we plan to explore this approach in our future work.

In order to guarantee finding the best plan, a planner must explore the complete search space. This is computationally very expensive, as we have argued above. Search techniques such as A^* are able to find the best path in a graph given an admissible heuristic function that evaluates the goodness of each node in the path. However it is difficult to find admissible heuristics that capture the quality metrics in planning problems, frequently due to the existence of conjunctive goals. We have chosen to follow a learning approach instead, in which past experience guides the current choices. This approach does not guarantee that the solutions found are optimal, but given an adequate experience in solving similar problems, it has been able to obtain significant plan quality improvements in our experiments.

4.7.3 The Accuracy of the Learned Control Knowledge

The cktree matching algorithm is used at a given decision point to provide guidance on the available alternatives. Its aim is to estimate the quality of the plans to which each of the alternatives would lead should it be chosen by the planner. To generate that estimate the cktree matcher traverses the cktree simulating the choices that the planner would made given its past planning experience. It considers the operator and bindings alternatives available at each point. In addition it assumes that the planner will follow a depth-first, left-to-right ordering in exploring the pending subgoals (relevant to keep track of operator side effects and thus interactions among subgoals in the cktree). This corresponds to PRODIGY4.0's default search strategy. The cktree matcher estimates the quality of the best plan that the planner might find under these assumptions.

The effect of limited past experience. As in the case of any learner, the performance of the global system would depend on the problem solving experience accumulated by the learner.

If the experience captured by the control knowledge trees is small, the estimates they provide may not be accurate, in particular when the problem for which guidance is sought belongs to a class (problem distribution) different from that of the problems used in the learning phase. When the cktree matcher does not have knowledge to generate an estimate at a cktree goal node, it uses a default value (the minimum cost of achieving that goal by just applying a relevant operator). Therefore the value for the goal node itself is underestimated. Also, if achievement and deletion links are missing, the matcher estimates may be inaccurate. If a goal is achieved by some node that would be in the missing subtree, the quality value of that goal would be overestimated (as the matcher does not realize that the goal would have cost 0). On the other hand, if the goal is deleted by the missing subtree, the matcher underestimates its achievement cost assigning it a cost 0.

The effect of other available control knowledge. In addition to the control knowledge encoded in the cktrees, the planner may have at its avail other control knowledge in the form of control rules, possibly hand-encoded by the domain writer. When the planner is searching, if those rules are applicable in the current meta-state (which includes the world state, and the current state of the search process, i.e. the set of pending goals, the expanded operators, etc), they may change the default search strategy. Therefore in the presence of control rules the actual control strategy used during planning may be different from the search strategy that the cktree matcher follows. That is, the space of alternatives explored by the cktree matcher may be different from the space actually explored by the planner. The presence of those control rules may reduce the accuracy of the quality estimates found by the cktree matcher: the plan (that is, the sequence of operator and bindings decisions used to achieve the subgoals) that the cktree matcher predicts may be different from the plan actually found by the planner, and therefore have different quality.

PRODIGY4.0 allows several types of control rules. Operator and bindings preference rules change the order in which the planner explores the available alternatives. As the cktree matcher explores all of them, (or a subset of them, as explained above) the acuracy of the estimate obtained should not be affected. On the other hand, select and reject operator and bindings control rules are used to reduce the set of alternatives that the planner considers. Currently the cktree matcher ignores these rules and therefore may pursue alternatives that will in reality would not be available to the planner, leading to inaccurate quality estimates. To extend the cktree matcher so it considers those rules when it is traversing the cktree, it should be able to predict the planner's meta-state at the point where the control rules would be tested in order to test the applicability of their preconditions. [13]

[13] After the planner makes the decision suggested by the cktree matcher, it continues searching (thus updating its meta-state) including expanding some of the subgoals that appear in the cktree (for which it may have guidance as well) and applying whatever control rules match at that point.

However the purpose of some select and reject rules is to prune alternatives leading to dead-end paths, in order to speed up the planning process. This is the case of the control rules built by PRODIGY's speed-up learning mechanisms. The cktree matcher is able to detect whether an alternative will lead to a dead-end and discard it if so, effectively pruning the alternatives that the rules would prune. Therefore the accuracy estimate is not affected by the existence of these types of rules.

Section 3.9 presented examples of the relevance of goal decisions for plan quality, and Section 4.5.11 mentioned how the cktree matching algorithm only provides control knowledge for operator and bindings decisions. It leaves to goal control rules to provide guidance for goal decisions. Extending the cktree matcher to provide goal ordering guidance would increase its computational cost, as the number of alternatives to explore would increase. Using control rules instead has proved satisfactory in our domains and therefore we have not added that capability to the cktree matcher yet.

We started this section describing how the matcher assumes a default ordering of goal expansion and achievement. The planner's control strategy may change that default by using either control rules or other search heuristics [Stone *et al.*, 1994]. As goal ordering affects plan quality, a difference between the planner's strategy and the default assumed by the cktree matcher may reduce the accuracy of the estimates produced by traversing the cktrees. Different goal orderings may lead to different achievement and deletions of goals and therefore the lists of achieved and deleted goals maintained by the cktree matcher may be different from the actually achievement and deletions that the planner would face at planning time.

Quality of guidance and accuracy of the estimates. We have described a number of factors that may contribute to the inaccuracy in the estimates obtained by the cktree matcher of the quality of the alternatives available to the planner at a given decision point. However it is important to note that even when those estimates are not accurate, the guidance provided by the cktrees can be useful: the planner's decision is actually guided by the comparison of those estimates among the alternatives. Their relative value is what matters. Section 4.5.9.3 explained how several alternatives are used to estimate the quality of one of them when there are interacting goals to avoid exploring the whole cktrees for the interacting goals. The estimates obtained consider only the parts of the problem relevant for comparing the different alternatives. In the example of Section 4.5.9.4 the estimates for both alternatives are lower than the real quality values, as the cktree lacks knowledge to evaluate the quality of part of the plan, i.e. of achieving some of the subgoals. Still those estimates are good enough to point to the best alternative at the choice point at which the control knowledge is invoked.

4.7.4 Tradeoffs Between Plan Quality and Planning Efficiency

Optimality in plan quality, that is, obtaining always the best plan, can only be guaranteed when the planner explores the complete search space. This has a high computational cost (in terms of planning efficiency, i.e. time). Our approach, *learning quality-enhancing control knowledge*, aims at finding good plans in an efficient way by exploiting past problem solving experience, that is, knowledge of which plans were good in the past. This approach is not optimizing *plan quality*. We do not claim that the use of the learned quality-enhancing control knowledge will lead to optimal plans, as the quality values computed are obtained from incomplete knowledge, from past experiences that may only partially match the problem being current solved. Our learning approach is not optimizing the *efficiency of the search* either, as the cktree matcher explores (a subset of) all the alternatives.

Our focus has been in learning about plan quality and generating good quality plans. Much previous work in learning for planning systems has focused on improving planning efficiency (speed-up learning) [Etzioni, 1990, Gratch *et al.*, 1993, Knoblock, 1994, Minton *et al.*, 1989, Mitchell *et al.*, 1986, Pérez and Etzioni, 1992, Veloso, 1994]. Although the algorithms described in this thesis are able to efficiently provide guidance towards better plans, we have largely ignored the issue of planning efficiency. In some of the examples showed the difference in quality between the available alternatives was small. Despite that, the matcher invested computational effort to find a good alternative. We were interested in being able to generate the better plan in those cases. Those small differences may translate into large economic savings, for example in the case in which the plan is going to be executed multiple times. However, in some domains the tradeoff between planning efficiency (i.e. planning time) and plan quality is an issue worth considering. It may not be worth spending time producing accurate quality estimates in order to find the best alternative when the second best is almost as good. We plan to explore this tradeoff in future work.

The use of past experience stored in the form of cktrees to guide future problem solving causes an additional tradeoff. The cktrees capture the quality of plans in parts of the search space that have been explored as part of previous experience. The alternatives corresponding both to the planner's initial solutions and to the improved solutions provided by the human expert are stored in the cktree. Other alternatives available in the domain (e.g. other operators for a given goal) that have not been used in previous episodes are not captured in the cktree. The cktree matcher provides as guidance only the alternatives stored in the cktree. Therefore it would not suggest other alternatives yet unexplored that could turn to be better. To avoid this effect the learner should allow some *exploration* in which instead of preferring something that has worked well in the past, it would try something new and learn from it (during training, the human expert if available provides such guidance).

4.8 Experimental Results

We have fully implemented the algorithms described in these chapter to automatically *acquire* control knowledge trees from experience and *use* them during planning to obtain good plans. This section describes our empirical evaluation of the implemented algorithms on the process planning domain. Section 4.6 analyzed cktree performance in a small transportation domain. Here we first analyze the performance of cktrees in generating good plans. Then we compare them with using different default operator choices, and the quality-enhancing control rules learned in the experiments of Chapter 3. We end with a demonstration of the reusability of cktrees across metrics different from the one they were generated for.

4.8.1 The Performance of the Learned Cktrees

In the beginning of this chapter we motivated the use of cktrees as control knowledge when control rules could not suitably capture the desired effect of the domain quality metric for generating high quality plans requiring non-local tradeoffs. We chose a quality metric of those characteristics (see Section 4.1.3 for an explanation) and tested the performance of the learned cktrees.

4.8.1.1 The Setting

Table 4.8 shows the quality metric used in this experiment. This metric assigns costs to the drill-press machining operators different from those for the milling-machine operators (second

Type	Cost	Operators
Drill press operators	3	drill-with-spot-drill, drill-with-twist-drill, drill-with-high-helix-drill, tap, countersink, counterbore, ream
Milling machine operators	5	drill-with-spot-drill-in-milling-machine, drill-with-twist-drill-in-milling-machine, face-mill, side-mill
Machine and holding device set-up operators	2	put-holding-device-in-drill, put-holding-device-in-milling-machine, remove-holding-device-from-machine, put-on-machine-table, remove-from-machine-table, hold-with-vise, release-from-holding-device
Tool operators	1	put-tool-on-milling-machine, put-tool-in-drill-spindle, remove-tool-from-machine
Cleaning operators	2	clean, remove-burrs
Oil operators	1	add-soluble-oil, add-mineral-oil, add-any-cutting-fluid

Table 4.8: The quality metric used in the experiments described in this section.

and third rows of the table). Thus the choice of operation and machine depends not only on the set-up cost (as in the experiments of Chapter 3) but also on the cost of the operator itself. In particular there is a tradeoff between the savings in cost of the drilling operations per se, and the savings on setting up the work in each case. Sections 4.1 and 4.1.3 give examples of such tradeoffs. Note that the differences in cost between the operators are not large, and therefore in many cases the differences in quality between the plans that use those operators will not be large either. Still we are interested in producing the better plans, as small differences in quality may translate in huge economic or other savings when these plans are executed thousands of times (see Section 3.13.3 for a discussion on these issues of our experimental evaluation of plan quality improvement).

The exact same problems as in the previous experiments were used, namely 60 randomly-generated problems for the training phase and 180 randomly-generated problems for the test phase. Note that, since the quality metric changed, the desired solutions for them, and therefore the planner's choices, may be different from those in the experiment of Section 3.13. *At every time* (in the training phase and the test phase, except when indicated) the goal-preference control rules learned in the experiments of Section 3.13 were part of the domain (see Section 4.5.11).

4.8.1.2 The Training Phase

As in the experiment of Section 3.13.2 the cktree learner was initially given 60 randomly-generated problems. For each of the 60 problems, it called PRODIGY4.0 to solve it. However in this experiment we did not rely on the human expert to generate the improved solutions for the 60 training problems. Instead those solutions were produced automatically by PRODIGY4.0 finding successive plans until the space was exhausted or the bound of 20000 nodes was reached. In order to avoid uninteresting paths (with respect to quality) a branch-and-bound technique was used, namely: the current path was abandoned if the quality value (given the quality metric described) of the incomplete current plan exceeded a bound dynamically set to the quality value of the best plan found so far. Hand-written domain-dependent control knowledge was available to the planner to guide its backtracking choices to prune paths that looked unpromising with respect to improving the current best plan.[14] This was implemented using PRODIGY4.0's interrupt and signal mechanism [Carbonell *et al.*, 1992]: every time a complete plan was found or the quality bound was exceeded, PRODIGY4.0 backtracked to the most recent node in which a different choice of machining operator (one of *drill-with-spot-drill-in-milling-machine, drill-with-spot-drill, drill-with-twist-drill, drill-with-twist-drill-in-milling-machine, face-mill, side-mill, ream, counterbore, tap* or *countersink*) or binding for it was open. The rationale is that those are the relevant decisions for quality purposes in the training problems.

[14]This knowledge includes rules for planning efficiency to avoid dead-end paths, which are orthogonal to plan quality issues, and control knowledge to abandon paths leading to a plan worse than the best one found so far.

- Average **planning** time (initial plan + improved plan): 25.9 s.
- Average **learning** time: 3.5 s.
- Problems learned from: 13

Table 4.9: Summary of the training phase. The numbers shown were computed for the 13 training problems in which learning was actually invoked.

Table 4.9 summarizes the results of the training phase. For 13 of the 60 problems a plan better than the one initially obtained by PRODIGY4.0 was available; in those cases learning was automatically invoked. The learned knowledge after each problem was used to solve the subsequent problems. Table 4.10 shows those results in detail. The third and fourth rows indicate planning and learning times for the problems in which learning occurred (as shown in row 2). The bottom part of the table shows data on the quality of the plans before and after learning. Note that learning is incremental, and the quality increase represented compares the plan with the knowledge learned up to that point with the better plan available. Therefore the quality increase refers only to the knowledge added by each individual problems (the table shows the total of those increases for each 10-problem set). Thus the % increase is not very large.

4.8.1.3 The Test Phase

We tested the performance of the learned cktrees on a set of 180 problems, different of the training problems (see Section 3.13.3 for more details on the test set). Table 4.11 shows the effect of the learned knowledge in plan quality. Row 4 of the table indicates the number of problems in which the plan quality was actually improved. Row 5 shows the rate of plan quality improvement considering only those problems. Note that in some problems no improvement is possible, as the planner obtains a good solution by default, without using control knowledge. Also, given the nature of the quality metric, the quality values for the different alternative plans may be close, and therefore there is no room for a large improvement. Still the learned knowledge captured the relevant distinctions and obtain the better plan. Section 3.13.3.1 explained why plan *quality* cannot be improved at the exponential rate which is possible when learning search *efficiency* control knowledge.

Table 4.12 shows the effect of the learned cktrees in planning efficiency, that is in the time and number of nodes visited by the planner during search. The number of nodes is smaller when cktrees are present due to shorter plan lengths (see Section 3.13.3.2 for a discussion). However planning time increases due to the overhead of matching the cktrees. The time spent increases with the difficulty of the goal interactions. For example, the cktree matching cost is large in

Problem set (10 problems per set)	1	2	3	4	5	6
Number of problems learned from	2	3	1	2	2	3
Planning time (secs.)	19.3	35.9	5.5	37.7	70.9	167.4
Learning time (secs.)	3.0	6.4	0.5	8.5	8.3	19.0

Cost before learning	119	146	127	324	376	531
Cost after learning (incremental)	116	140	121	318	355	525
% Cost decrease (in improved problems only)	10%	11%	46%	12%	28%	11%

Table 4.10: Experimental results for the training phase. Each column corresponds to a set of 10 problems. The second row shows the number of problems of each problem set in which learning occurred, prompted by the availability of a better quality solution. The third and fourth rows show respectively the total time spent in planning and the total time spent in learning, considering only those problems in which learning occured. The bottom table shows plan quality data, namely the total cost of the solutions initially output by PRODIGY4.0 for the 10 problems in each set (row 6); the total cost of the better solutions for the 10 problems of the set (row 7); and finally the improvement in quality (or cost decrease) due to learning to obtain the better solutions, now considering only the problems for which a better plan was available (row 2 of the top of the table). Note that each problem was solved using the control knowledge learned for the previous problems, that is the improvement in quality in each problem is between the planner with the current knowledge before and after learning in that individual problem.

Problem set (30 problems per set)	1	2	3	4	5	6
Cost without learned control knowledge	350	620	764	1117	1261	1570
Cost with learned control knowledge	310	446	622	945	898	1448
Number of problems with improvement	8	25	18	22	27	21
% Cost decrease (in improved problems only)	31%	30%	25%	18%	32%	8%
Max % decrease	40%	44%	40%	46%	55%	23%

Table 4.11: Improvement on the quality of the plans for 180 randomly-generated problems obtained by using the learned cktrees. The numbers refer to the quality metric of Table 4.8. The second row of the table shows the total quality of the plans obtained for the 30 problems in the each set without using quality-enhancing control knowledge (except for the goal-ordering rules learned in the experiments of previous chapter). Row 3 shows the corresponding values using the learned cktrees. Row 4 shows the number of problems in each set in which the solution obtained using the learned cktrees was better than that obtained without quality-enhancing control knowledge. Row 5 shows the rate of improvement considering only those problems. The last row shows the maximum rate of improvement in a single problem of the set. Note that these values are not comparable to those in Table 3.4 because they correspond to a different quality metric.

problem set 5 because some of the problems have the goal of cutting the same part along three different dimensions (three goal conjuncts) which can be achieved with only two set-ups if the planner chooses appropriate operators (*side-mill* or *face-mill*), bindings (machine instance, orientation, and tool), and order of goal achievement. Note that in the case of that problem set the increase in search time corresponds with higher rate of quality improvement (Table 4.11).

Problem set	Without learned control knowledge			With learned cktrees		
	Qual	Time	Nodes	Qual	Time	Nodes
1	350	83	1405	310	96	1148
2	620	143	2196	446	139	1616
3	764	153	2111	622	314	1770
4	1117	275	3576	945	303	3089
5	1261	287	3641	898	1264	2467
6	1570	519	5571	1448	566	4781

Table 4.12: Effect of the learned control knowledge in the planning time and in the number of nodes searched. Each row displays the total quality, planning CPU time (seconds), and nodes for the 30 problems in the test set. The values shown are averaged over 5 runs for each problem. Columns 1 to 3 show the numbers without quality-enhancing control knowledge. Columns 4 to 6 show the values when the cktrees automatically generated in the training phase are used during planning.

This fact raises issues on the tradeoffs between plan quality and planning efficiency. In some cases it may not be worth spending time producing accurate quality estimates in order to find the best alternative when the second best is almost as good. It is interesting to note that the overhead of using control rules (Section 3.13.3.2) is significantly less than using cktrees, though the latter are able to capture nuances and tradeoffs leading to better plan quality.

4.8.2 Effect of the Default Operator Choice

The quality metric used shown in Table 4.8 gives a better quality to the machining operations performed in drill presses over operations performed in milling machines. Therefore planning choices that prefer using drill presses will frequently lead to better plans, unless the part is partially set on a milling machine and resetting it would overcome the savings by using the drill press. Thus the default planner's heuristic to choose operators (prefer the first relevant operator that appears in the domain definition file) may have an effect in the quality of the plans obtained by the planner in the absence of other control knowledge.

We performed the following experiment to see the effect of this operator choice heuristic and analyze more accurately the quality improvements obtained by the learned knowledge. Note

that operator choices are not the only reasons for differences in plan quality (binding and goal choice are also relevant). We compared the quality of the plans for 80 of the test problems obtained (a) using the learned cktrees and (b) using different default operator orderings and no other control knowledge. Figure 4.66 shows the results. No single operator ordering achieves a better performance than the cktrees. There are two reasons for that. First, even though some operators are better than others (Table 4.8) in some problems the "locally" worse operator should be preferred because the savings in set-up cost are greater. Second, other types of choices, in particular bindings choices, influence quality. The cktrees provide guidance for those choices as well, while in the other scenarios the PRODIGY4.0's default bindings heuristics are the only guidance.

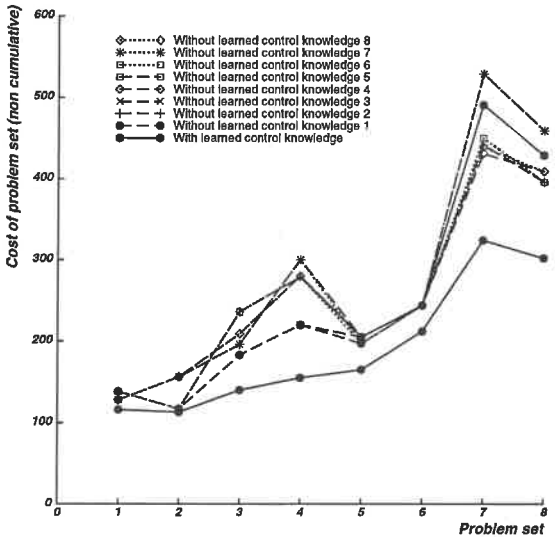


Figure 4.66: Analyzing the effect of operator ordering. Each data point represents the total cost of the 10 problems in a problem set. The solid line represents plan quality when using the learned cktrees. Each of the other lines represents a different default operator ordering.

4.8.3 Comparing Learned Control Rules and Learned Cktrees Performance

We have discussed how the quality metrics captured by the control knowledge trees are more complex than those to which the control rules learned by the algorithms of Chapter 3 are applicable. This section presents an experimental comparison of the performance of both approaches, given a quality metric suitable for control rule learning. The metric used is in Table 3.1. We showed in Section 3.13 how automatically acquired control rules are able to improve considerably the quality of the planner's solutions for that metric. The same training and test sets are used in this experiment, namely 60 randomly-generated problems for training

Problem set	No learned ctrl knowl			Learned control rules			Learned cktrees		
	Qual	Time	Nodes	Qual	Time	Nodes	Qual	Time	Nodes
1	886	84	1405	689	71	1135	689	97	1135
2	1528	141	2196	755	99	1594	771	155	1608
3	1716	152	2111	1330	204	1852	1218	313	1770
4	2765	272	3633	2056	249	3083	2043	410	3087
5	3421	313	3915	1472	214	2432	1583	3630	2536
6	3834	509	5546	3245	426	4590	3432	653	4874

Table 4.13: An empirical comparison of the two learning approaches: learning control rules and learning control trees. The table shows the quality of the solution obtained, CPU time (in seconds) to obtain that solution, and the number of nodes explored. The values showed are cumulative over the 30 problems in each set. The time results are averaged over 5 runs for each problem. The performance of the planner without any quality-enhancing control knowledge is also shown. The quality metric used is in Table 3.1. Higher values mean worse quality.

and 180 problems for testing divided in 6 problem sets. Table 4.13 shows the performance of the learned rules and the learned control trees for the test set described there. The table also shows the performance of the planner when no quality-enhancing control knowledge is available.

The performance of the learned cktrees and of the learned control rules is very similar and considerably better than that of the base planner (i.e. without quality control knowledge). This is not surprising since the quality metric is one suitable for the control rule learner, as the experiment in Section 3.13 proved. As we argued there, the learned knowledge was able to guide the planner toward virtually optimal solutions, i.e. solutions that could not be improved by the human expert.

This experiments also shows that the learned control rules outperform the cktrees in the time spent finding those problems. This occurs even though both systems explore a similar number of nodes. The cktree matching process is more costly than the control rule matching. Only 36 rules were learned (see Section 3.13) leading to a small matching overhead.

In all the experiments with cktrees reported so far the *goal-preference control rules* learned in the experiments of Section 3.13 were part of the domain, since the cktree learning algorithms do not learn goal-ordering control knowledge. The last row of Table 4.14 shows the quality values (according to the metric in Table 3.1) when the learned goal-preference rules are *not* used for problem solving and therefore the effect in plan quality performance is only due to the learned cktrees. Although some quality performance is lost, these results show that most of the improvement is indeed coming from the use of the learned cktrees.

Problem set (30 problems per set)	1	2	3	4	5	6
No learned control knowledge	886	1528	1716	2765	3421	3834
Learned control rules	689	755	1330	2056	1472	3245
Learned control knowledge trees with learned goal-ordering rules	689	771	1218	2043	1583	3432
Learned control knowledge trees without learned goal-ordering rules	689	771	1218	2175	1617	3578

Table 4.14: An experiment to factor out the effect of the learned goal-preference control rules from the effect of the cktrees. The numbers shown are the total quality of the plans obtained for the 30 problems in each problem set given the metric in Table 3.1. The last row shows the performance of the learned cktrees when the learned goal-preference rules are not used. For comparison the quality totals when no learned control knowledge, learned control rules, and learned cktrees (with goal-preference rules) are shown in rows 2-4. Those values come from Table 4.13. Although some quality performance is lost by using the cktrees without the learned goal-preference rules (row 4), these results show that most of the improvement is indeed coming from the guidance provided by the learned cktrees.

4.8.4 Reusing Learned Cktrees across Quality Metrics

Section 4.7.1 discussed the ability of the cktrees learned for a given quality metric to transfer and generate good plans for other metrics, as they do not capture a particular one. The metric is not used when the cktrees are built. It is only used at cktree matching time: the cktree matcher suggests planning decisions by using the current quality metric as it traverses the cktree and estimates the quality of each available alternative. Table 4.15 presents experimental results of this transfer.

Problem set (10 problems per set)	1	2	3	4	5	6
Cktrees learned for metric 1 (Table 3.1)	309	451	622	936	929	1431
Cktrees learned for metric 2 (Table 4.8)	310	446	622	945	898	1448

Table 4.15: Effects in quality of reusing cktrees learned for different metrics. The table shows the quality, *given the quality metric of Table 4.8,* of the 60 problems in the test set using two different sets of cktrees. The second row shows the results when the cktrees learned for the metric in Table 3.1 are used. (Those are the cktrees learned during the experiment of Section 4.8.3). Thus those cktrees were learned for a metric different from the one used for testing in this table. For comparison, the third row shows the results when the cktrees learned for the metric in Table 4.8 are used (Section 4.8.1.3). The quality values obtained in both cases are comparable, denoting transfer of the learned knowledge across quality metrics.

In this experiment the 60 problems of the test set were solved twice using the quality metric of Table 4.8 (metric 2). In the first pass, we used the cktrees learned in the experiment of the previous section, that is, for the metric of Table 3.1 (metric 1). Thus the cktrees were learned for a different metric than that they are being tested on now. Row 2 of Table 4.15 shows the quality of the plans obtained. In the second pass we used the cktrees learned for the current metric (metric 2, Table 4.8) and the quality values obtained are shown in row 3. These results show that the quality of the plans generated for the same quality metric with those two different control knowledge strategies (cktrees) is comparable.

This is a useful feature of the cktree formalism. Reusing the same control knowledge, i.e. the cktrees, when the quality metric changes amortizes the cost of learning along a larger number of problems in which that knowledge is useful.

4.9 Summary

This chapter has introduced control-knowledge trees, a formalism to represent control knowledge. The chapter describes domain-independent algorithms to automatically build cktrees from planning experience, and to use them during planning in order to generate good quality plans. The motivation behind this new representation is to allow a more global view of the planner's decisions than in the case of control rules, in order to make globally better choices. Some quality metrics that capture non-local tradeoffs are difficult to capture by control rules that make only local decisions.

The experimental results show that cktrees succeed in improving plan quality under those kinds of quality metrics. Although the overhead of using cktrees is higher than that of using control rules, they are able to captures nuances and tradeoffs that are relevant for plan quality. In addition cktrees transfer across quality metrics, since the metric is not captured in them, but only considered when the cktrees are used during planning.

Chapter 5

Related Work

This chapter describes other research related to this dissertation. Our work is a contribution to the area of machine learning for AI planning systems. The first section of this chapter reviews other work in that area. QUALITY can take advantage of the problem solving experience of a human domain expert, and thus is related to the systems called learning apprentices. Section 5.2 discusses some of that work. Finally Section 5.3 examines other approaches to the problem of generating good plans in the planning community. QUALITY differs from them in its *learning* view.

5.1 Learning Search-Control for Planning

The approach to acquiring quality-enhancing control knowledge described in this thesis falls in the broader category of systems that learn problem-solving expertise [Mitchell, 1983]. Most of the research to date on learning search-control knowledge for planning in particular has focused on making planning more efficient, i.e. on speed-up learning. Several techniques have been used in this framework, including learning search control knowledge [Mitchell *et al.*, 1986, Minton, 1988, Etzioni, 1990, Pérez and Etzioni, 1992, Borrajo and Veloso, 1994a, Katukam and Kambhampati, 1994], macro-operators [Fikes *et al.*, 1972, Korf, 1985, Cheng and Carbonell, 1986, Segre *et al.*, 1993], chunking [Laird *et al.*, 1986], abstraction hierarchies [Knoblock, 1994, Christensen, 1990], and problem-solving cases [Veloso, 1994]. Many of these systems report considerable reductions in the amount of search, but in general they do not pay attention to plan quality issues. Search reductions in speed-up learning systems frequently are due to taking advantage of or minimizing the effects of goal interactions. When this happens, search time is typically reduced and better solutions tend to be found. These solutions are generally shorter in length, and more direct [Minton, 1988, Ryu and Irani, 1992, Veloso, 1994]. Knoblock [1994] reports some small reductions in solution length by using automatically built

abstraction hierarchies to guide search in a hierarchical planner. However there is no guarantee that short plans at high abstraction levels lead to short plans at the ground level or that the best abstract solution will be the most useful for refinement [Bergmann and Wilke, 1995]. Veloso [1994] also reports smaller solution lengths obtained by the analogical problem solver in PRODIGY due to the strategy chosen (random interleaving) for merging multiple cases during replay.[1] The quality metric used in all these cases is plan length.

Some recent research has started to focus on learning about plan quality. In work independent and simultaneous with ours, Iwamoto [1994] has developed an extension to PRODIGY to solve optimization problems, and an explanation-based learning (EBL) method to learn control rules to find near-optimal solutions in LSI design. The quality goals are represented explicitly and based on the quality of the final state. QUALITY instead represents the quality of the plan.[2] The learning method is similar to QUALITY's control-rule learning algorithm in that it compares two solutions of different quality. It builds an explanation by backpropagating the weakest conditions, but excluding the conditions expressed in terms of the predicates related to quality. QUALITY however makes use of the quality evaluation function to determine the relevant conditions, which we believe leads to more succinct rules. Iwamoto's system learns operator preference rules. QUALITY can learn in addition bindings and goal preference rules. Both systems build incomplete explanations, which lead to over-general rules. Iwamoto's system deals with them by constructing a generalization hierarchy of the learned rules, and giving priority to the most specific rule (according to the hierarchy) among the matched ones. QUALITY instead learns priorities among the conflicting rules, as it may not be possible to order them in terms of a specificity hierarchy (see the examples in Section 3.11.4 and 3.11.5). Iwamoto's method does not allow user guidance and uses exhaustive search until the quality goal is satisfied to find the best solution. This is possible because of the relatively small size of the search space of the LSI examples used.

HAMLET [Borrajo and Veloso, 1994b, Veloso *et al.*, 1995] learns PRODIGY4.0 control rules that improve both planning efficiency and also the length of the plans generated, by a combination of bounded explanation and induction. Instead of relying on a comparison of pairs of complex plans as in QUALITY, HAMLET assumes that the learner is trained with simple problems for which the planner can explore the space of all possible plans to find the optimal one(s), and does not take advantage of human expert guidance. The quality metric is not used during the explanation process. Over-general rules in HAMLET are refined by looking at episodes in which they lead to wrong decisions and modifying their applicability conditions. QUALITY instead learns priorities among rules that match simultaneously and give conflicting preferences.

Since both HAMLET and Iwamoto's system learn control rules, they may suffer of the limitations

[1] Section 4.7.1 compares further this thesis work with PRODIGY/ANALOGY.

[2] In some domains (robot control, assembly, and organic synthesis) the quality of the sequence of operators is important. In contrast, in some design domains (scheduling and circuit design) the quality of the final state is what matters [Kibler, 1993].

of using control rules as representation formalism for control knowledge that we discussed in Section 4.1 when quality metrics grow in complexity. Our cktrees address those limitations and can be reused across different quality metrics.

HAMLET, Iwamoto's system, and QUALITY address the issue of the intractability of constructing complete explanations [Mitchell *et al.*, 1986]. Tadepalli [1990] notes that in general, intractability can arise from many sources, such as missing information, need for optimal solutions, or the presence of an active adversary. The solution followed by all these approaches is building incomplete explanations. As a consequence they may learn over-general rules. Early work by Minton in the context of learning for game playing [Minton, 1984, Minton, 1985] suggested that it may be better to learn rules that recommend plausible good moves rather than provably optimal moves [Tadepalli, 1989]. LEBL [Tadepalli, 1989, Tadepalli, 1990] confronts the intractable theory problem in game domains also by generalizing incomplete explanations and incrementally refining the over-general knowledge thus learned when met with unexpected plan failures. In contrast with our work, refinement occurs by introducing exceptions. Other systems deal with the intractability of complex theories by making simplifying assumptions and refining the learned knowledge upon failure [Chien, 1989, Bhatnagar, 1992, Ellman, 1988]. To our knowledge none of these systems included plan quality concerns as a target for learning or worried specifically about plan quality issues. QUALITY focuses precisely on those. We agree with Kibler [1993] in that "the major concern for real-world problems is the quality of the solution and not the speed at which the solution is reached. The value of EBL should not be measured by how much efficiency is improved, but by how much solution quality is improved."

SteppingStone [Ruby and Kibler, 1991] learns sequences of subgoals to to deal with interactions among subproblems. These sequences have the mixed properties of macro-operators and control rules. It allows soft constraints which measure the quality of a solution. These constraints are treated as subgoals and ordered along with the other problem subgoals by the learned knowledge. In its application to VLSI design SteppingStone is able to optimize multiple constraints (critical path delay and number of gates) simultaneously in spite of the tradeoffs among them. SteppingStone does not learn operator or binding choices, but uses local search guided by the learned goal sequences, which performs quite well in the VLSI domain chosen. It relies in training problems that are small enough for local search alone to produce optimal designs. EASe [Ruby and Kibler, 1992] is a generalization of SteppingStone, in which problem solving knowledge about goal sequences is learned and stored in the form of episodes (or cases). The episode indexing mechanism orders them by the amount they can improve upon the subgoal they are relevant for, thus using the quality metric to retrieve the best relevant episode.

R1-Soar's task [Rosenbloom *et al.*, 1985] is computer configuration. In some of R1-Soar's problem spaces the goal includes optimizing over some criterion and therefore the best solution is found given enough computational resources. Search control knowledge learned by chunking

prunes considerably the search space allowing the system to satisfy the optimization goals. (It gets rid of search altogether in some problem spaces.) The learned chunk contains conditions for the aspects that were accessed when solving the subgoal. It makes no distinction between decisions to reduce search and decisions to prefer better solutions. QUALITY instead explains the difference in quality between pairs of paths (solutions), rather than unioning what was relevant along the two paths. Therefore it may be able to build more general rules.

It is interesting to note that although our work focuses only on learning about plan quality, there is an implicit relationship with efficiency. If planning consumed few computational resources, the planner could explore the whole search space and find the optimal solution. However in practice this is not possible within reasonable computational bounds and motivates the need for control knowledge, both efficiency-improving and quality-improving. Efficiency control knowledge prunes portions of the search space and may allow a planner without quality-improving knowledge to further explore the space and improve the quality of its solutions within a given limit of computational resources [Williamson and Hanks, 1994, Rosenbloom *et al.*, 1985]. On the other hand, the knowledge learned by QUALITY improved not only solution quality but also the planner's efficiency by leading it to a solution faster (Section 3.13.3.2).[3] Kibler [1993] proposes a framework in which planning efficiency, probability of solving a problem, and quality of the solution are instances of the definition of the quality of a problem solver.

Some efforts in other areas of machine learning have analyzed the tradeoffs between different goals for learning, namely learning time, accuracy of the learned knowledge, and cost of making mistakes if that knowledge is not correct [Provost and Buchanan, 1992, desJardins, 1991]. MAX [Provost, 1993] judges learned potential concept descriptions based on a linear polynomial that factored in both accuracy and cost.

5.2 Interacting with a Human Expert

Because of the large search spaces in complex domains, finding good enough plans from which to learn can be computationally expensive. On the other hand, human expertise is available in many domains, and can be advantageously used to help the system find useful strategies to obtain good plans. For this reason our architecture is designed so it can benefit from human guidance. The general idea of systems taking advice was proposed by McCarthy [1968]. Systems that learn from different types of human expert guidance fall are known as *learning apprentice systems* [Mitchell *et al.*, 1990]. Some of these systems learn by observing expert actions [Mitchell *et al.*, 1990, Martin and Redmond, 1989, Wilkins, 1988, Dent *et al.*, 1992, Wang, 1995]. Others attempt to solve problems and then allow the expert to critique their

[3] Discussions with Jon Gratch and Steve Chien pointed out this way of looking at quality control knowledge.

decisions [Laird *et al.*, 1990, Tecucci, 1992, Huffman and Laird, 1994, Porter and Kibler, 1986]. QUALITY falls in the latter category, although the expert does not critique problem solving decisions directly but only the final plan in order to make oblivious the planning algorithm and representation language to the expert. Learning apprentices rely in different degrees on the expert's interaction, from a non-intrusive observation (for example [Mitchell *et al.*, 1990, Dent *et al.*, 1992, Wang, 1995]) to direct advice at decision points [Golding *et al.*, 1987, Laird *et al.*, 1990] or the expert supplying illustrative examples within the system's current knowledge grasp [Golding *et al.*, 1987]. QUALITY can work autonomously (without human expert interaction) as the quality metric is known to the planner and it can always solve the problem from first principles.

Control knowledge used in LEAP [Mahadevan, 1990] to rank alternative circuit implementations (regarding power or delay) of a boolean specification is learned by asking a user to select one operator from a list of those that are applicable to refine the current specification. Thus the user directly provides the problem solver with an example of a control decision. Domain theories to estimate circuit cost are incomplete at the lower levels of design. LEAP uses determinations to form simple partial theories of circuit cost; they express approximate correlations between attributes that can be computed at the abstract levels. They are used to discriminate among alternative operators, and also to further refine the theory by explaining subsequent decisions. A form of explanation-based learning is used to propagate the information in the user's control decisions to refine the partial cost theory. The learned rules are over-specific and cannot guide high-level abstract decisions because the explanations built use very simple terms; thus it is restricted to learning simple boolean-level control rules.

Robo-Soar [Laird *et al.*, 1990] (see also [Golding *et al.*, 1987]) actively seeks guidance while solving problems. The problem solver does not build plans but picks the next action given the current state and outside advice. When the system is not able to select the next operator it asks for guidance among the acceptable operators at that point. Therefore guidance is at the level of individual decisions instead of complete plans. Guidance acts as a heuristic and is verified by the internal problem solver, which then generates chunks [Laird *et al.*, 1986] using only those elements of the working memory necessary to derive the result. By being *situated* in the state in which the decision must be made, chunking can learn the relevant conditions for the guidance to apply in the future. PRODIGY's glass-box approach (Appendix A) makes any problem-solver state during search available at any time, including at the end of problem solving. Therefore QUALITY does not need to learn while problem solving is happening. The goal of producing good plans and PRODIGY4.0's problem solving algorithm make the operationalization of the outside guidance a more involved process. QUALITY uses the quality metric to guide learning instead of just gathering the weakest preconditions for the application of the guidance. Robo-Soar's approach to correcting the learned control knowledge is to increase the interaction between the expert and the system so the expert can point out the relevant features that caused the success or failure of the system's choice. QUALITY relies in very simple, if none, expert guidance.

Search control knowledge can also be acquired by knowledge acquisition methods [Gruber, 1989, Joseph, 1992], as extracting from the human experts justifications for their choices. QUALITY instead has fully automated the acquisition task by using a purely machine-learning approach. The expert, if at all present, does not need to make explicit the reasons for the choices of plan steps. In addition, its focus is on quality-enhancing control knowledge.

Recent research has pointed out the role of planning systems as personal assistants to human experts as a requirement in real-world domains [Chien *et al.*, 1994, Muscettola and Pell, 1994, desJardins, 1994, Gutknecht *et al.*, 1991]. There is a growing interest for mixed-initiative systems in which both humans and machines can make contributions to a problem solution, often without being asked explicitly (Jaime Carbonell, Sr., cited in [Burstein and McDermott, 1994]). Although QUALITY is still far from that description, it is able to interact with an expert and incorporate her/his expertise to use in future problem solving. Guided by QUALITY's learned knowledge, the planner can offer different alternative plans for a given problem, each of them good according to some quality metric.

In the domain of scheduling two interesting pieces of work, CABINS [Sycara and Miyashita, 1994] and the system described in [Hamazaki, 1992], acquire user preferences to generate good schedules.[4] Both pieces of work are motivated by the fact that which is a better schedule depends on user preferences, which balance conflicting objectives and tradeoffs, and are difficult to express as a cost function. CABINS acquires those preferences in the form of repair tactics to improve sub-optimal schedules, and of ways to evaluate those repairs. The acquired preferences are stored as cases and used to guide the iterative solution optimization of job shop schedules. The knowledge acquired by CABINS is on how to fix a sub-optimal complete schedule, while QUALITY learns knowledge to guide the generation of a good plan in the first place. In Hamazaki's system the expert specifies directly quality factors, their characteristics (hard/soft, local/global) and priorities among them in as much detail as possible. These quality factors are used during scheduling. The resulting schedule is then critiqued by the expert. If it is not satisfactory, the expert specifies which factors should be modified. QUALITY assumes instead that a quality metric is available and therefore can function autonomously without the human expert. In future work we would like to loosen that assumption, given what Sycara, Miyashita, and Hamazaki suggest about how quality knowledge is represented in certain domains, and explore how the quality metric can be built from the expert's interaction.

5.3 Planning Approaches to Generating Good Quality Plans

This section overviews planning research on generating good plans. Much of the work on planning to obtain good plans has aimed at taking advantage of goal interactions. We describe

[4]Thanks to Austin Tate for the pointer to Takashi Hamazaki's work.

some of that work. Then we overview some research on decision theory and planning related to this dissertation. Next we review some domain-specific approaches to generating quality plans, and we end with a brief overview of different types of quality metrics.

5.3.1 Plan Quality and Goal Interactions

Section 1.3.1 described the effect of the interactions between conjunctive goals in the quality of the plans that solve them [Wilensky, 1983, Pollack, 1991, Nau, 1993, Pérez and Veloso, 1993]. Wilensky [Wilensky, 1983] analyzes the different types of goal interactions and develops meta-planning mechanisms to deal with them. When a goal overlap, or positive goal interaction between a planner's goals, occurs, his planner is able to carry out an action that is in the service of a number of goals at once. Goal overlap situations provide opportunities to achieve goals more economically than they could be achieved otherwise, and the planner prefers efficient plans over inefficient ones. Nau [1993] refers to these situations as enabling-condition interactions. This principle also seems to be the underlying justification for a number of processes incorporated in other planning systems. Pollack uses a related strategy called overloading [Pollack, 1991, Pollack, 1992]. Several of NOAH's critics [Sacerdoti, 1977] including "use existing objects," "eliminate redundant preconditions," and "optimize disjuncts" are motivated by this idea and correspond to particular kinds of goal overlap situations. Although these strategies are rather general, they can lead to suboptimal plans in some cases (e.g. using existing objects can be bad if robustness is a major plan quality concern, as in the space shuttle domain). In the case of QUALITY the learned knowledge takes advantage of goal interactions. But those goal interactions are learned and exploited insofar as they lead to good plans according to the domain-specific quality metric (see for example Section 3.9.1).

The LCOS planning strategy (Least Commitment to Operator Selection) described in [Hayes, 1994] is related to our use of control knowledge trees. The motivation for LCOS is to take a global view of the plan in order to make globally optimal operator choices. Commitments to particular operators are made only when operator choices for all goals have been explored. When solving a problem with multiple goals, an AND/OR tree is constructed containing all goals and one or more candidate operators for them; that tree is searched to find both positive (cost sharing) and negative interactions, and then a set of operators is chosen that satisfies all the goals and maximizes the quality criteria for the domain. In order to limit the search, heuristics used by human planners (machinists) are used, such as which alternatives tend to be more useful. Hayes proposes the use of abstraction or the modification of the operators preconditions to reduce the search space and indicates that these are good candidates to be learned automatically. In contrast, QUALITY's cktrees are learned from past experience and reused (instead of built for each problem solving episode) and search in them is limited by the alternatives that proved useful in the past. Instead of exploring alternatives for *all* the goals, as in LCOS, past experience captured in the cktrees dictates which cktrees, and thus which goal

interactions, to explore. The cktrees provide not only operator guidance but also instantiations of those operators (which are also relevant for generating good plans).

Some planners solve multiple-goal problems by developing separate plans for the individual goals, combining these plans to form a naive plan for the conjoined goal, and then performing optimizations to yield a better combined plan [Nau *et al.*, 1990, Yang *et al.*, 1992]. However they restrict the types of goal interactions that may happen. In this context, the quality of a plan is only considered as far as dealing with and taking as much advantage as possible of goal interactions. A similar mechanism is also used by some domain-dependent planners [Hayes, 1990, Nau, 1987, Karinthi *et al.*, 1992].

Several systems perform plan debugging as their problem solving strategy [Sussman, 1975, Hammond, 1987, Simmons, 1988b]. They employ heuristic rules to generate an initial hypothesis and then debugging if the hypothesis is incorrect. Therefore they fix planning failures (not execution failures). An example is the Generate, Test and Debug paradigm [Simmons, 1988b] in which the debugger analyzes causal explanations for why a bug arises and fixes it by replacing those assumptions. The debugger is only used if the heuristic generator produces an incorrect hypothesis. In contrast our planner generates correct plans and our goal is not to fix them but to improve their quality. Our approach does not perform post-facto modification of the plans, but analyzes the problem solving process to extract knowledge that will guide the problem solver towards better solutions.

5.3.2 Decision Theoretical Planning

Recent research has focused in the application of decision theory to planning. This section overviews some of that work relevant to this dissertation. In general optimization techniques in operations research require pre-specifying the available alternatives and thus are not directly applicable to knowledge-based symbolic problem-solving. Symbolic planning provides methods for representing planning problems and generating alternative plans for goals. On the other hand decision theory provides a method for choosing among alternatives and a language that allows reasoning about utility and uncertainty. However it provides no guidance in structuring planning knowledge, no way of generating alternatives, and no computational model [Haddawy and Hanks, 1993]. Utility functions can be arbitrarily expressive but may not be amenable at effective problem solving. At worst a problem-solver is forced to generate complete plans, apply the utility function, and choose the plan with the highest expected value. To plan effectively the planner must be able to evaluate the (potential) quality of (incomplete) plans as they are generated in order to discard unpromising ones early in the process. The approach followed by [Feldman and Sproull, 1977] was to add restrictions (probability and utility models) to classical planning algorithms. SUDOPLANNER [Wellman, 1988] reasons about the relative value (dominance) of plans or plan classes. These two pieces of work did not provide empirical investigations of tractability and did not exploit domain-specific heuristics.

DRIPS [Haddawy and Suwandi, 1994] structures actions into an abstraction hierarchy and refines plans whose expected utility is known within an interval. In general our quality metrics are simpler than the utility functions proposed by the systems above and focus just on quality (utility) and not uncertainty (see [Blythe, 1994] for probabilistic planning in the context of the PRODIGY architecture).

The PIRRHUS planner [Williamson and Hanks, 1994] is an interesting extension of the UCPOP least-commitment planner [Penberthy and Weld, 1992] with a utility model that allows goals whose value is a function of their satisfaction time, and plan cost as a function of consumption and replenishment of resources. PIRRHUS uses branch-and-bound search in the space of partial plans. Each time a complete plan is generated it computes its exact utility and compares it to the best found so far. If it is better, it is kept and the bound updated. Partial plans with worse utility are discarded and planning terminates when the plan queue is empty. Therefore it guarantees to find optimal plans (given enough computational resources). Williamson and Hanks empirical tests on a truck world show how heuristic search-control knowledge (inspired by that of PRODIGY) allows PIRRHUS to find a complete plan more quickly and therefore make it tractable. They plan to explore other domains, including more realistic ones.

This thesis chooses instead a learning approach. We confronted the problem of making operational at problem solving time the domain-specific definition of the quality (utility) of a plan. In the case of learned control rules, they capture the quality metric and directly guide the planner in the path of a good solution. In the case of cktree learning, the range of alternatives explored during cktree matching is guided and constrained by knowledge of previous planning experience (which actions produced good plans in the past, and how goals interacted). In both cases it is the *learned* knowledge what makes the problem tractable and produces plans of good quality. The learned control knowledge could be integrated with the approach described above as Williamson and Hanks's results suggest. We agree with Williamson and Hanks in that, since finding an optimal plan is "at least as hard as classical planning, [...] the best we can hope for is an algorithm [aimed at finding an optimal plan] that is *heuristically tractable*, that is, one that can perform well on a class of problems in a particular domain, given an adequate body of domain-specific knowledge." In our approach that "given" domain-specific knowledge is automatically acquired from planning experience.

Simon introduced the idea of "satisficing" [Simon, 1981] arguing that a rational agent does not always have the resources to determine what the optimal action is, and instead should attempt only to make good enough, to satisfice. Some current work on planning (for example [Pollack, 1992]) is about the tradeoff between getting around to acting, and spending enough time thinking. Such resource-bounded reasoning leads to suboptimal behavior. In our work we do not consider the tradeoff between acting and planning time. We acknowledge the computational cost of finding the optimal behavior and do not claim that the acquired control knowledge will necessarily guide the planner to optimal plans, but that plan quality will improve incrementally over experience as the planner sees new interesting problems and interacts with the human

expert.

Another body of work consists of methods to choose and/or learn optimal policies of action. Some examples are reinforcement learning (e.g. [Lin, 1992]), dynamic programming, and real-time A^* [Korf, 1988]. However these methods have been applied to more reactive models and not so much to solve complex planning problems.

The complexity of the problem of finding optimal solutions in the blocks world is analyzed in [Gupta and Nau, 1991] and [Chenoweth, 1991]. In both cases optimal solutions are shortest-length plans.

5.3.3 Domain-Dependent Approaches

In this section we briefly discuss some domain-dependent approaches to generate quality plans, in particular focusing in the process planning domain.

Hayes' MACHINIST program [Hayes, 1990] generates plans in a machining process planning domain. Human machinists often spend a large amount of time in the early planning stages looking over the part specification for feature interactions and exploring the limitations that those features impose on the plan. Machinists have specialized knowledge which helps them to quickly focus on the situations in which interactions are likely. This knowledge is acquired through experience. Hayes analyzed the way features interact and encoded this specialized knowledge in form of rules in the MACHINIST program. This program first constructs a plan that deals with feature interactions, and retrieves from memory a plan to square the part (squaring is getting the raw material into a square and accurate shape with the minimum waste of material). Then these two plans are merged to produce a final plan as short as possible. Although plan length and plan cost are not the same, machinists and MACHINIST use it as an estimator of plan cost, because setup cost is almost always a much larger cost than all the other costs [Hayes, 1995a].

SIPP [Nau and Chang, 1985] is a process planning system that produces plans for the creation of metal parts. It utilizes a frame hierarchy to represent problem solving knowledge. In particular, actions have cost slots that contain relative costs derived from actual process costs and shop preferences. The problem solving strategy utilizes a least-cost-first branch and bound algorithm to find the least-cost sequence of processes for making each of the part's machinable surfaces. SIPP selects the least cost manufacturing method for an individual feature, in isolation from considerations about other features, and therefore it does not care to find an overall low-cost plan. SIPP is domain-dependent although they anticipate it will be useful in other domains as well.

GARI [Descotte and Latombe, 1985] is a process planner for metal cutting that uses a constraint satisfaction algorithm. Knowledge is represented by manufacturing rules: the left-hand side consists of conditions about the desired part, the available machines, and/or the machining

plan. The right-hand side contains pieces of advice representing technological and economical preferences, and so it encodes knowledge about the quality of the plans. In contrast with PRODIGY, there is no separation in the representation between domain knowledge and search control knowledge. Experts have to assign a weight to each piece of advice according to the importance of its satisfaction. These weights are an extremely condensed representation of a large body of knowledge, and human experts have difficulty in expliciting them. The initial state of a problem contains global pieces of information such as the quality desired for the part. To our knowledge this system has not been applied to other domains.

In contrast to these approaches QUALITY avoids the knowledge acquisition effort to capture quality control knowledge by *learning* it from problem solving experience when a quality metric is available. QUALITY's learning mechanisms are fully independent of the application domain.

IRS [Sun and Weld, 1992] is an interesting approach to diagnosis and repair that combines partial-order planning with model-based diagnosis. IRS uses a cost function that accounts for both the eventual need to repair broken parts and the cost of probing to make the diagnosis. The top-level of IRS is a diagnostic reasoner, and the planner (UCPOP) is called as a subroutine to generate actions sequences (given a fault suggested by the diagnoser) and estimate their cost. That cost is used by the diagnoser as part of the larger cost function to eventually find the best diagnostic operation. However the planner does not use guidance or the cost function to generate a good plan. All the quality decisions are made by the diagnoser. Thus the problem is different from the one QUALITY attacks.

5.3.4 Different Quality Metrics

Section 1.2 briefly described there groups of quality metrics. Figure 5.1 shows the taxonomy in more detail. We refer the reader to [Pérez and Carbonell, 1993], from where the taxonomy is taken, for details. Of the classes of metrics shown, QUALITY has concentrated on reducing the plan execution cost. Note that the taxonomy only captures plan quality metrics, and not planning efficiency metrics.

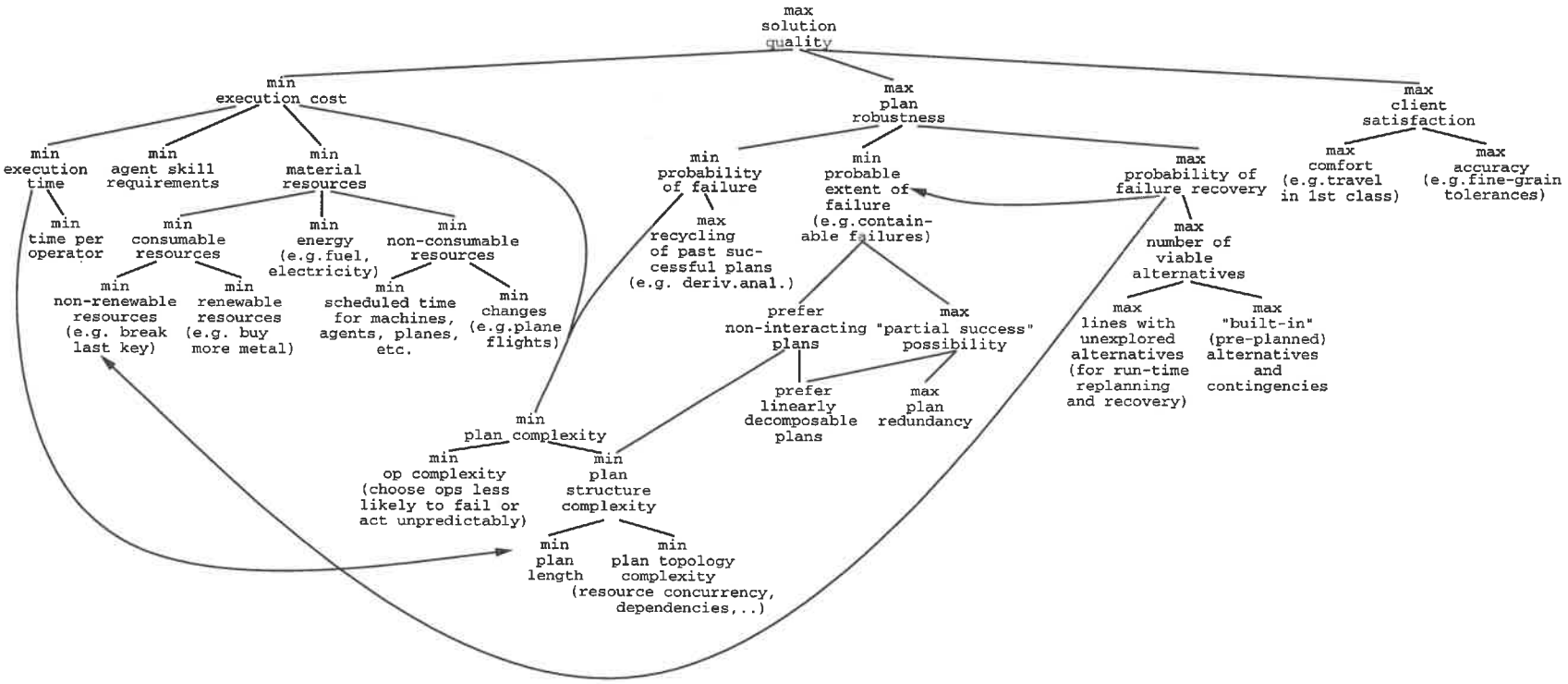


Figure 5.1: A taxonomy of quality metrics (from [Pérez and Carbonell, 1993]). These metrics can be classified in three broad categories, namely those regarding execution cost, those related to robustness of the plan obtained, and those considering the degree of satisfaction of the client with the solution obtained. Note that this taxonomy does not consider planning efficiency metrics.

Chapter 6

Conclusion

In this final chapter we summarize this thesis and its contributions, and outline some directions for future research.

6.1 Summary of the Thesis

This thesis has explored a general framework to solve the problem of generating good plans in AI planning systems. The approach chosen has two fundamental characteristics. The first one is representing knowledge about plan quality as operational, planning-time search-control knowledge. The second one is automatically acquiring such control knowledge from planning experience in a machine *learning* approach. Our motivation has been that typically the kind of knowledge about quality that is available takes the form of a cost function or quality metric. However such knowledge is non-operational, as it cannot be used until after a plan is produced. The problem thus is how to translate such quality knowledge into planning-time decision control guidance. Automating this mapping based on planning experience and the (optional) interaction with a human domain expert has been the objective of this thesis. This general framework has been implemented in QUALITY, an architecture built on top of PRODIGY4.0, the most recent nonlinear planner of the PRODIGY architecture. PRODIGY has proved a suitable vehicle for our investigation because it has clear explicit decision points that permit the infusion of automatically, or manually, acquired control knowledge to improve plan quality.

In this general framework we have developed two distinct learning mechanisms to efficiently acquire quality control knowledge. Both mechanisms are domain independent and require only a domain definition, a metric of the quality of the plans specific to that domain, and problems from which to draw problem solving experience. They differ in the language used to represent the learned knowledge, namely quality-enhancing search control rules and control knowledge trees. They also differ in the kinds of quality metrics they are suited for.

The learned quality-enhancing control rules provide effective operator, bindings, and goal ordering guidance when the quality metric does not require reasoning about complex global tradeoffs. They lead to near-optimal plans in many cases. They are highly operational and very efficiently used at planning time. The performance in improving plan quality of the learned control knowledge trees is equivalent to that of control rules for simpler non-interacting situations, and superior for more complex metrics that require reasoning about tradeoffs and taking a global view of the plan to make a set of globally optimal choices. However using control knowledge trees is computationally more expensive and may reduce planning efficiency, since cktrees are less operational than control rules. Control knowledge trees do not provide goal ordering control knowledge. In our experiments we have used them together with the goal preference rules learned by the first method and achieved a synergistic effect. An important advantage of control knowledge trees is their robustness to changes in the quality metric, which is parameterized. On the other hand the learned control rules are quality metric specific, are invalidated if the metric changes, and must be relearned. Integrating the two learning mechanisms to take advantage of their complementary characteristics remains an open issue.

In addition to acquiring control knowledge from planning experience, QUALITY can benefit from the interaction with a human expert in the application domain. This interaction is at the level of plan steps. Our objective was that the expert could remain oblivious to the planning algorithm and representation language, thus reducing the knowledge engineering effort of acquiring quality-enhancing control knowledge.

QUALITY has been fully implemented and its empirical evaluation has shown that the learned knowledge significantly improves the quality of the plans without a considerable loss of planning efficiency. In fact, in some of our experiments the learned knowledge *improves* planning efficiency, in addition to plan quality. Although the approach, the learning mechanisms, and the learned knowledge representations have been developed for the PRODIGY4.0 planner, the framework is general and addresses a problem that any planner that treats planning as search, as a constructive decision-making process, must confront. Therefore our framework is suitable for other planners.

Generating good plans is an essential step in moving our current AI planners from research tools towards real-world applications. This thesis is a step in that direction.

6.2 Future Research Directions

This thesis has opened some lines of future research. They are discussed in this final section.

6.2.1 Improvements to the Learning Architecture

The implementation of the learning algorithms described in this thesis has left room for improvements. Some of them are mentioned here. One is the use of other search techniques to traverse the cktrees, for example interval-based search as we discussed in Section 4.7.2.

It is interesting to explore the effect of the training problems and the training sequence for learning cktrees. Which are good training examples? Since cktrees are used to estimate the cost of different alternatives, problems that require a deeper exploration of the search space may lead to cktrees that provide better estimates. In our experiments we have noticed that if the "right" example is seen first (e.g. one in which the complete set-up must be built for machining a hole) the learned knowledge is enough to provide guidance in a whole set of related problems. A teacher can be useful in providing examples that explore parts of the space for which the learned knowledge is not able yet to produce good quality estimates. The cktrees do not capture alternatives that have not been seen in past problems, and thus they are not suggested as good alternatives. It would be nice to have an *exploratory* mode in which the learner, instead of preferring something that has worked well in the past, would try something new and learn from it.

The interaction of the learner with the domain expert through the Interactive Plan Checker (Section 3.2) has not been at the core of our research, and in fact QUALITY can learn without it. The interaction can be improved in different directions: offering further default alternatives, flexibility to retract steps, a better display of the current state, and the integration with PRODIGY's graphical user interface. As planners address more realistic problems and move towards more interactive, mixed-initiative approaches, the role of user interfaces increases. Quoting [Lieberman, 1994], "the machine learning problem is really one of interaction. The key is to establish an interaction language for which the human teacher finds it easy to convey notions of interest, and at the same time, for which the computer as a student is capable of learning the appropriate inferences." QUALITY closes the gap between the expert advice and the level of inferences required by the learner. Machine learning for planning systems can get increased leverage by a careful interaction with the experts to whom they support, especially when solving problems closer to the real world. We would like to further explore that interaction in the larger context of PRODIGY as an integrated planning and learning system.

6.2.2 Other Quality Metrics and Other Domains

As planning and learning research moves towards real-world applications, plan quality issues are becoming increasingly important [Chien *et al.*, 1994]. A challenging future direction for our work is to apply the quality-improving control-knowledge learning techniques to real-world domains. The process planning domain we have described can easily contemplate more realistic quality factors. For example the choice of a shaping, milling, or turning operation to reduce

a part's size depends on many variables such as price, desired accuracy, or tool life [Zhang and Lu, 1992]. Database query optimization [Siegel *et al.*, 1991, Arens *et al.*, 1993] is a knowledge-intensive domain that has been cast out as an interesting planning problem that we plan to explore. In a realistic transportation domain [Strom, 1994] factors such as weather forecasts, vehicle availability, or seasonal increases in package traffic affect plan quality.

More realistic domains may require to extend the class of quality metrics allowed by our algorithms. Section 1.2 briefly presented different factors that, in addition to plan execution cost, influence plan quality. These may also include maximizing a plan's robustness, reliability, or possibility of recovery, minimizing its uncertainty, or maximizing other factors that are not easily quantifiable. Even more challenging is the fact the these factors can appear combined and lead to tradeoffs. Additionally in some domains user preferences are a more natural way than cost functions to represent plan quality [Sycara and Miyashita, 1994, Hamazaki, 1992]. We plan to further explore the gamut of quality metrics. Our control knowledge trees formalism can deal with tradeoffs in the metric. Although we have only considered plan cost factors, the tradeoffs could be due to other kinds of factors. However due to the way the quality metric is currently used by the cktree matcher, it must be additive on the operator costs, or at least the cost of the plan must be monotonically increasing in the costs attached to the subgoals in the plan trees or cktrees. Currently the metric is an input to the learner, and it is consistent with the expert's advice. Acquiring the metric itself from the interaction with the expert is another open problem.

6.2.3 Quality and Planning Efficiency Tradeoffs

Although we have concentrated on improving plan quality, we have generally ignored the tradeoffs between finding good plans and finding them quickly, that is, between the cost of plans and the cost of planning. This is an open avenue for our work. If the best and the second best solutions are close in cost, is it worth the time spent finding the best one? These issues are especially interesting when plan generation and plan execution are interleaved. Some questions to address are: Are there tradeoffs between response time and quality? Is it better to find the best solution or to find quickly a good enough solution? Can we generate good, robust plans? How to recover from execution failures without compromising (too much) quality?

The two learning algorithms subject of this thesis have different characteristics that can be seen as complementary. Control rules are applied efficiently to make local choices. Control knowledge trees are computationally more expensive to use but provide global guidance in the presence of non-local tradeoffs. An interesting research direction is to explore an architecture in which both representations are combined. A reasonable architecture could use simple, local control rules when the distinctions among choices are clear, and only use global control knowledge trees when needed. Other work [Simmons, 1988a, Goodwin, 1994] addresses these

meta-level issues on how to achieve good performance without consuming too many resources in the process.

6.2.4 Other Planning Techniques

Another line of future work is to apply our quality learning techniques to other classical planning architectures such as least-commitment planners. We want to analyze what aspects of the techniques are relevant to the particular planning algorithm used (PRODIGY4.0) and how they can improve the performance of other planners.

In Section 4.7.1 we compared cktrees with cases. With the ideas we have gathered about quality metrics and the relevance for plan quality of control decisions during planning, we want to explore the use of cases to provide that control guidance. What would be needed to extend PRODIGY/ANALOGY [Veloso, 1994] to deal with plan quality? Which would a good similarity metric be? It would have to estimate the quality of the case's solution for the current problem given that the case may need to be adapted.

Bibliography

[Arens *et al.*, 1993] Yigal Arens, Chin Y. Chee, Chun-Nan Hsu, and Craig A. Knoblock. Retrieving and integrating data from multiple information sources. *International Journal on Intelligent and Cooperative Information Systems*, 2(2):127–159, 1993.

[Barrett and Weld, 1994] Anthony Barrett and Daniel S. Weld. Partial-order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67(1):71–112, 1994.

[Bergmann and Wilke, 1995] Ralph Bergmann and Wolfgang Wilke. Building and refining abstract planning cases by change of representation language. *Journal of Artificial Intelligence Research*, Forthcoming, 1995. Also as technical report: Report LSA-95-07E, Centre for Learning Systems and Applications, Univ. of Kaiserslautern, Kaiserslautern, Germany.

[Berliner, 1979] Hans Berliner. The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12:23–40, 1979.

[Bhatnagar, 1992] Neeraj Bhatnagar. Learning by incomplete explanations of failures in recursive domains. In D. Sleeman and P. Edwards, editors, *Machine Learning: Proceedings of the Ninth International Conference, ML92*, pages 30–36. Morgan Kaufmann, San Mateo, CA., 1992.

[Blythe and Veloso, 1992] Jim Blythe and Manuela Veloso. An analysis of search techniques for a totally-ordered nonlinear planner. In *Proceedings of the First International Conference on AI Planning Systems*, College Park, MD, June 1992.

[Blythe, 1994] Jim Blythe. Planning with external events. In Ramon López de Mantaras and David Poole, editors, *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 94–101, Seattle, WA, July 1994. Morgan Kaufmann.

[Borrajo and Veloso, 1994a] Daniel Borrajo and Manuela Veloso. Incremental learning of control knowledge for nonlinear problem solving. In *Proceedings of the European Conference on Machine Learning, ECML94*, Sicily, Italy, 1994. Springer Verlag.

[Borrajo and Veloso, 1994b] Daniel Borrajo and Manuela Veloso. Incremental learning of control knowledge for improvement of planning efficiency and plan quality. In *Working Notes of the AAAI 1994 Fall Symposium Series, Symposium on Planning and Learning: On to Real Applications*, New Orleans, November 1994.

[Burstein and McDermott, 1994] Mark H. Burstein and Drew McDermott. Mixed-initiative military planning: Directions for future research and development. In Mark H. Burstein, editor, *ARPA/Rome Laboratory Knowledge-Based Planning and Scheduling Initiative, Workshop Proceedings*, pages 467–483, Tucson, AZ, February 1994.

[Carbonell *et al.*, 1992] Jaime G. Carbonell, and the PRODIGY Research Group: Jim Blythe, Oren Etzioni, Yolanda Gil, Robert Joseph, Dan Kahn, Craig Knoblock, Steven Minton, Alicia Pérez, (editor), Scott Reilly, Manuela Veloso, and Xuemei Wang. PRODIGY4.0: The manual and tutorial. Technical Report CMU-CS-92-150, School of Computer Science, Carnegie Mellon University, June 1992.

[Chang and Wysk, 1985] Tien C. Chang and Richard A. Wysk. *An Introduction to Automated Process Planning Systems*. Prentice Hall, Englewood Cliffs, NJ, 1985.

[Chapman, 1987] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378, 1987.

[Cheng and Carbonell, 1986] Patricia Cheng and Jaime Carbonell. The FERMI system: Inducing iterative macro-operators from experience. In *Proceedings of the National Conference on Artificial Intelligence*, pages 490–495, Philadelphia, PA, 1986.

[Chenoweth, 1991] Stephen V. Chenoweth. On the NP-hardness of blocks world. In *Proceedings of Ninth National Conference on Artificial Intelligence*, pages 623–628, Anaheim, CA, July 1991.

[Chien *et al.*, 1994] Steve Chien, Randall W. Hill, and Kristina Fayyad. Why real-world planning is difficult. In *Working Notes of the AAAI 1994 Fall Symposium Series, Symposium on Planning and Learning: On to Real Applications*, pages 28–33, New Orleans, November 1994.

[Chien, 1989] Steve A. Chien. Using and refining simplifications: Explanation-based learning of plans in intractable domains. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 590–595, Detroit, MI, 1989.

[Christensen, 1990] Jens Christensen. A hierarchical planner that creates its own hierarchies. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1004–1009, Boston, MA, 1990.

[de Silva, 1995] Rujith de Silva. Reasoning about goal-interactions in non-linear planners. Thesis proposal. School of Computer Science, Carnegie Mellon University, 1995.

[Dent *et al.*, 1992] Lisa Dent, Jesús G. Boticario, John McDermott, Tom Mitchell, and David Zabowski. A personal learning apprentice. In *Proceedings of the National Conference on Artificial Intelligence*, pages 96–103, San Jose, CA, 1992.

[Descotte and Latombe, 1985] Yannick Descotte and Jean-Claude Latombe. Making compromises among antagonist constraints in a planner. *Artificial Intelligence*, 27:183–217, 1985.

[desJardins, 1991] Marie desJardins. Probabilistic evaluation of bias for learning systems. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 495–499. Morgan Kaufmann, 1991.

[desJardins, 1994] Marie desJardins. Knowledge development methods for planning systems. In *Working Notes of the AAAI 1994 Fall Symposium Series, Symposium on Planning and Learning: On to Real Applications*, New Orleans, November 1994.

[Doyle, 1969] Lawrence E. Doyle. *Manufacturing Processes and Materials for Engineers.* Prentice-Hall, Englewood Cliffs, NJ, second edition, 1969. Third edition available, 1985.

[Doyle, 1985] Lawrence E. Doyle. *Manufacturing Processes and Materials for Engineers.* Prentice-Hall, Englewood Cliffs, NJ, third edition, 1985.

[Ellman, 1988] Thomas Ellman. Approximate theory formation: An explanation-based approach. In *Proceedings of the National Conference on Artificial Intelligence*, pages 570–574, St. Paul, MN, 1988.

[Etzioni, 1990] Oren Etzioni. *A Structural Theory of Explanation-Based Learning.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, 1990. Also appeared as Technical Report CMU-CS-90-185.

[Feldman and Sproull, 1977] Jerome A. Feldman and Robert F. Sproull. Decision theory and artificial intelligence II: The hungry monkey. *Cognitive Science*, 1:158–192, 1977.

[Fikes *et al.*, 1972] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4), 1972.

[Foulser *et al.*, 1992] David E. Foulser, Ming Li, and Qiang Yang. Theory and algorithms for plan merging. *Artificial Intelligence*, 57:143–181, 1992.

[Gil and Pérez, 1994] Yolanda Gil and M. Alicia Pérez. Applying a general-purpose planning and learning architecture to process planning. In *Working Notes of the AAAI 1994 Fall Symposium Series, Symposium on Planning and Learning: On to Real Applications*, pages 48–52, New Orleans, November 1994.

[Gil, 1991] Yolanda Gil. A specification of process planning for PRODIGY. Technical Report CMU-CS-91-179, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, August 1991.

[Gil, 1992] Yolanda Gil. *Acquiring Domain Knowledge for Planning by Experimentation.* PhD thesis, Carnegie Mellon University, School of Computer Science, August 1992. Available as technical report CMU-CS-92-175.

[Golding *et al.*, 1987] Andrew Golding, Paul S. Rosenbloom, and John E. Laird. Learning general search control from outside guidance. In *Proceedings of the Tenth International Conference on Artificial Intelligence*, pages 334–337, Milan, Italy, 1987.

[Goodwin, 1994] Richard Goodwin. Reasoning about when to plan and what to plan. Thesis proposal. School of Computer Science, Carnegie Mellon University, February 1994.

[Gratch *et al.*, 1993] Jonathan Gratch, Steve Chien, and Gerald DeJong. Learning search control knowledge for deep space network scheduling. In *Machine Learning. Proceedings of the Tenth International Conference*, pages 135–142, Amherst, MA, June 1993. Morgan Kaufmann.

[Gruber, 1989] Thomas R. Gruber. Automated knowledge acquisition for strategic knowledge. *Machine Learning*, 4:293–336, 1989.

[Gupta and Nau, 1991] Naresh Gupta and Dana S. Nau. Complexity results for blocks-world planning. In *Proceedings of Ninth National Conference on Artificial Intelligence*, pages 629–633, Anaheim, CA, July 1991.

[Gutknecht *et al.*, 1991] Matthias Gutknecht, Rolf Pfeifer, and Markus Stolze. Cooperative hybrid systems. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI91*, pages 824–829, Sydney, Australia, 1991.

[Haddawy and Hanks, 1993] Peter Haddawy and Steve Hanks. Utility models for goal-directed decision-theoretic planners. Technical Report 93-06-04, Department of Computer Science and Engineering, University of Washington, June 1993.

[Haddawy and Suwandi, 1994] Peter Haddawy and Meliani Suwandi. Decision-theoretic refinement planning using inheritance abstraction. In *Proceedings of the Second International Conference on AI Planning Systems, AIPS-94*, pages 266–271, Chicago, IL, June 1994.

[Haigh *et al.*, 1994] Karen Zita Haigh, Jonathan Richard Shewchuk, and Manuela Veloso. Route planning and learning from execution. In *Working Notes of the AAAI 1994 Fall Symposium Series, Symposium on Planning and Learning: On to Real Applications*, pages 58–64, New Orleans, November 1994.

[Hamazaki, 1992] Takashi Hamazaki. High quality production scheduling system. In *Proceedings of SPICIS 92*, pages 195–200, 1992.

[Hammond, 1987] K. Hammond. Explaining and repairing plans that fail. In *Proceedings of the Tenth International Conference on Artificial Intelligence*, Milan, Italy, 1987.

[Hammond, 1994] Kristian Hammond, editor. *Proceedings of the Second International Conference on AI Planning Systems, AIPS-94, Chicago, IL.* The AAAI Press, Menlo Park, CA, June 1994.

[Hayes, 1990] Caroline C. Hayes. *Machining Planning: A Model of an Expert Level Planning Process.* PhD thesis, The Robotics Institute, Carnegie Mellon University, December 1990.

[Hayes, 1994] Caroline C. Hayes. Planning using least-commitment to operator selection. In *Working Notes of the AAAI 1994 Fall Symposium Series, Symposium on Planning and Learning: On to Real Applications*, pages 65–71, New Orleans, November 1994.

[Hayes, 1995a] Caroline C. Hayes. Personal communication, June 1995.

[Hayes, 1995b] Caroline C. Hayes. QUEM: A method for measuring the solution quality and experience level of knowledge-based systems. *IEEE Transactions on Data and Knowledge Engineering*, forthcoming, 1995.

[Hendler *et al.*, 1990] James Hendler, Austin Tate, and Mark Drummond. AI planning: Systems and techniques. *AI Magazine*, 11(2):61–76, Summer 1990.

[Hendler, 1992] James Hendler, editor. *Proceedings of the First International Conference on AI Planning Systems, AIPS-92, College Park, MD.* Morgan Kaufmann, San Mateo, CA, June 1992.

[Huffman and Laird, 1994] Scott B. Huffman and John E. Laird. Learning from higly flexible tutorial instruction. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 506–512, Seattle, WA, July 1994. AAAI Press/The MIT Press.

[Huffman *et al.*, 1993] S. Huffman, D. Pearson, and J. Laird. Correcting imperfect domain theories: A knowledge-level analysis. In Susan Chipman and Alan L. Meyrowitz, editors, *Foundations of Knowledge Acquisition: Cognitive Models of Complex Learning*. Kluwer Academic Publishers, Boston, 1993.

[Iba, 1993] Glenn A. Iba. Speedup and scale-up in experiential learning. In *Proceedings of 3rd International Workshop on Knowledge Compilation and Speedup Learning, in ML93*, pages 90–95, Amherst, MA, June 1993.

[Iwamoto, 1994] Masahiko Iwamoto. A planner with quality goal and its speedup learning for optimization problem. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 281–286, Chicago, IL, 1994.

[Joseph, 1992] Robert L. Joseph. *Graphical Knowledge Acquisition for Visually-Oriented Planning Domains*. PhD thesis, Carnegie Mellon University, School of Computer Science, August 1992. Also appeared as Technical Report CMU-CS-92-188.

[Kambhampati *et al.*, 1993] Subbarao Kambhampati, Mark R. Cutkosky, Jay M. Tenenbaum, and Soo Hong Lee. Integrating general purpose planners and specialized reasoners: Case study of a hybrid planning architecture. *IEEE Transactions on Systems, Man and Cybernetics, Special Issue on Planning, Scheduling, and Control*, 23(6), 1993.

[Karinthi *et al.*, 1992] Raghu Karinthi, Dana S. Nau, and Qiang Yang. Handling feature interactions in process planning. *Applied Artificial Intelligence*, 6(4):389–415, October-December 1992. Special issue on AI for manufacturing.

[Katukam and Kambhampati, 1994] Suresh Katukam and Subbarao Kambhampati. Learning explanation-based search control rules for partial order planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 582–587, Seattle, WA, July 1994. AAAI Press/The MIT Press.

[Kibler, 1993] Dennis Kibler. Some real-world domains for learning problem solvers. In *Proceedings of KCSL93, 3rd International Workshop on Knowledge Compilation and Speedup Learning (in ML93)*, Amherst, MA, 1993.

[Knoblock, 1994] Craig A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68, 1994.

[Korf, 1985] Richard E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.

[Korf, 1988] R. E. Korf. Real-time heuristic search: New results. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, St Paul, MN, 1988. Morgan Kaufmann.

[Kushmerick *et al.*, 1994] Nicholas Kushmerick, Steve Hanks, and Danield Weld. An algorithm for probabilistic least-commitment planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1073–1078, Seattle, WA, July 1994. AAAI Press/The MIT Press.

[Laird *et al.*, 1986] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[Laird *et al.*, 1990] John E. Laird, Michael Hucka, Eric S. Yager, and Christopher M. Tuck. Correcting and extending domain knowledge using outside guidance. In Bruce W. Porter and Ray J. Mooney, editors, *Machine Learning: Proceedings of the Seventh International Conference, ML90*, pages 235–243, Austin, TX, June 1990. Morgan Kaufmann.

[Leckie and Zukerman, 1993] Christopher Leckie and Ingrid Zukerman. An inductive approach to learning search control rules for planning. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI93*, 1993.

[Lieberman, 1994] Henry Lieberman. A user interface for knowledge acquisition from video. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 527–534, Seattle, WA, July 1994. AAAI Press/The MIT Press.

[Lin, 1992] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8, 1992.

[Mahadevan, 1990] Sridhar Mahadevan. *An Apprentice-Based Approach to Learning Problem-Solving Knowledge*. PhD thesis, Rutgers, The State University of New Jersey, Department of Computer Science, May 1990. Technical Report ML-TR-30.

[Marcus, 1990] Sandra Marcus, editor. *Knowledge Acquisition: Selected Research and Commentary*. Kluwer Academic Publishers, 1990.

[Martin and Redmond, 1989] Joel D. Martin and Michael Redmond. Acquiring knowledge for explaining observed problem solving. *SIGART Newsletter, Knowledge Acquisition Special Issue*, 108:77–83, April 1989.

[McAllester and Rosenblitt, 1991] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 634–639, Anaheim, CA, 1991.

[McCarthy, 1968] John McCarthy. Programs with common sense. In Marvin Minsky, editor, *Semantic Information Processing*, pages 403–418. MIT Press, Cambridge, MA, 1968.

[Minton *et al.*, 1989] Steven Minton, Jaime G. Carbonell, Craig A. Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence*, 40:63–118, 1989.

[Minton, 1984] Steven Minton. Constraint-bases generalization: Learning game-playing plans from single examples. In *Proceedings of the National Conference on Artificial Intelligence*, pages 251–254, Austin, TX, 1984.

[Minton, 1985] Steven Minton. A game-playing program that learns by analyzing examples. Technical Report CMU-CS-85-130, School of Computer Science, Carnegie Mellon University, May 1985.

[Minton, 1988] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-based Approach*. Kluwer Academic Publishers, Boston, MA, 1988. PhD thesis available as Technical Report CMU-CS-88-133, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[Mitchell *et al.*, 1986] Tom M. Mitchell, Richard Keller, and Smadar Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1), 1986.

[Mitchell *et al.*, 1990] Tom M. Mitchell, Sridhar Mahadevan, and Louis I. Steinberg. LEAP: A learning apprentice system for VLSI design. In Yves Kodratoff and Ryszard Michalski, editors, *Machine Learning: An Artificial Intelligence Approach*, volume III, pages 271–289. Morgan Kaufmann, San Mateo, CA, 1990.

[Mitchell, 1983] Tom M. Mitchell. Learning and problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI83*, volume 2, pages 1139–1151, Karlsruhe, Germany, 1983. Computers and Thought Lecture.

[Mostow, 1983] D. Jack Mostow. Machine transformation of advice into a heuristic search procedure. In R. S. Michalsky, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning, An Artificial Intelligence Approach*. Tioga Press, Palo Alto, CA., 1983.

[Muscettola and Pell, 1994] Nicola Muscettola and Barney Pell. Toward real-world science mission planning. In *Working Notes of the AAAI 1994 Fall Symposium Series, Symposium on Planning and Learning: On to Real Applications*, New Orleans, November 1994.

[Nau and Chang, 1985] Dana S. Nau and Tien-Chien Chang. Hierarchical representation of problem-solving knowledge in a frame-based process planning system. Technical Report TR-1592, Computer Science Department, University of Maryland, November 1985.

[Nau *et al.*, 1990] Dana S. Nau, Qiang Yang, and James Hendler. Optimization of multiple-goal plans with limited interaction. In *Proceedings of the Darpa Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 160–165, San Diego, CA, November 1990.

[Nau, 1987] Dana S. Nau. Automated process planning using hierarchical abstraction. *Texas Instruments Technical Journal*, Winter:39–46, 1987.

[Nau, 1993] Dana S. Nau. Enabling-condition interactions and finding good plans. In *Working Notes of the AAAI 1993 Spring Symposium Series, Symposium on Foundations of Automatic*

Planning: The Classical Approach and Beyond, pages 93–97, Stanford University, CA, March 1993.

[Penberthy and Weld, 1992] J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference, KR92*, pages 103–114, San Mateo, CA, October 1992. Morgan Kaufmann.

[Pérez and Carbonell, 1993] M. Alicia Pérez and Jaime G. Carbonell. Automated acquisition of control knowledge to improve the quality of plans. Technical Report CMU-CS-93-142, School of Computer Science, Carnegie Mellon University, April 1993.

[Pérez and Etzioni, 1992] M. Alicia Pérez and Oren Etzioni. DYNAMIC: A new role for training problems in EBL. In D. Sleeman and P. Edwards, editors, *Machine Learning: Proceedings of the Ninth International Conference, ML92*, pages 367–372. Morgan Kaufmann, San Mateo, CA., 1992.

[Pérez and Veloso, 1993] M. Alicia Pérez and Manuela M. Veloso. Goal interactions and plan quality. In *Working Notes of the AAAI 1993 Spring Symposium Series, Symposium on Foundations of Automatic Planning: The Classical Approach and Beyond*, pages 117–121, Stanford University, CA, March 1993.

[Pérez, 1994] M. Alicia Pérez. The goal is to generate better plans. In *Working Notes of the AAAI 1994 Spring Symposium Series, Symposium on Goal-Driven Learning*, pages 88–93, Stanford University, CA, March 1994.

[Perlin, 1988] Mark W. Perlin. Transforming programs into networks: Call-Graph Caching, applications, and examples. Technical Report CMU-CS-88-202, School of Computer Science, Carnegie Mellon University, December 1988.

[Pollack, 1991] Martha E. Pollack. Overloading intentions for efficient practical reasoning. *Noûs*, 25(4):513–536, 1991. Also as Technical Note 497 of SRI International.

[Pollack, 1992] Martha E. Pollack. The uses of plans. *Artificial Intelligence*, 57:43–68, 1992.

[Porter and Kibler, 1986] Bruce Porter and Dennus Kibler. Experimental goal regression: A method for learning problem-solving heuristics. *Machine Learning*, 1:249–286, 1986.

[Provost and Buchanan, 1992] Foster John Provost and Bruce Buchanan. Inductive policy. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 255–261, San Jose, CA, July 1992. AAAI Press/The MIT Press.

[Provost, 1993] Foster John Provost. Goal-directed inductive learning: Trading off accuracy for reduced error cost. In *Working Notes of the AAAI 1993 Spring Symposium Series, Symposium on Foundations of Automatic Planning: The Classical Approach and Beyond*, pages 94–100, Stanford University, CA, March 1993.

[Rosenbloom *et al.*, 1985] Paul S. Rosenbloom, John E. Laird, John McDermott, Allen Newell, and Edmund Orciuch. *R1-Soar*: An experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(5):561–569, September 1985.

[Ruby and Kibler, 1990] David Ruby and Dennis Kibler. Learning steppingstones for problem solving. In *Proceedings of the Darpa Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 366–373, San Diego, CA, November 1990.

[Ruby and Kibler, 1991] David Ruby and Dennis Kibler. Steppingstone: An empirical and analytical evaluation. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 527–532, Anaheim, CA, 1991.

[Ruby and Kibler, 1992] David Ruby and Dennis Kibler. Learning episodes for optimization. In D. Sleeman and P. Edwards, editors, *Machine Learning: Proceedings of the Ninth International Conference, ML92*, pages 379–384. Morgan Kaufmann, San Mateo, CA., 1992.

[Ryu and Irani, 1992] Kwang R. Ryu and Keki B. Irani. Learning from goal interactions in planning: Goal stack analysis and generalization. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 401–407, San Jose, CA, July 1992. AAAI Press/The MIT Press.

[Sacerdoti, 1977] Earl Sacerdoti. *A Structure for Plans and Behavior*. Elsevier, North Holland, New York, 1977.

[Segre *et al.*, 1993] Alberto M. Segre, David Sturgill, and Jennifer Turney. Neoclassical planning. In *Preprints of the AAAI 1993 Spring Symposium Series, Symposium on Foundations of Automatic Planning: The Classical Approach and Beyond*, Stanford University, CA, March 1993.

[Shen, 1989] Wei-Min Shen. *Learning from the Environment Based on Percepts and Actions*. PhD thesis, Carnegie Mellon University, School of Computer Science, June 1989. Available as technical report CMU-CS-89-184.

[Siegel *et al.*, 1991] Michael Siegel, Edward Sciore, and Sharon Salveter. Rule discovery for query optimization. In George Piatetsky-Shapiro and William J. Frawley, editors, *Knowledge*

Discovery in Databases, pages 411–427. AAAI Press/The MIT Press, Menlo Park, CA., 1991.

[Simmons, 1988a] Reid G. Simmons. Combining associational and causal reasoning to solve interpretation and planning problems. Technical Report 1048, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 1988. PhD thesis.

[Simmons, 1988b] Reid G. Simmons. A theory of debugging plans and interpretations. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 94–99, St Paul, MN, 1988. Morgan Kaufmann.

[Simon, 1981] Herbert A. Simon. *The Sciences of the Artificial*. The MIT Press, Cambridge, MA, second edition, 1981.

[Stone *et al.*, 1994] Peter Stone, Manuela Veloso, and Jim Blythe. The need for different domain-independent heuristics. In *Proceedings of the Second International Conference on AI Planning Systems, AIPS-94*, pages 164–169, Chicago, IL, June 1994.

[Strom, 1994] Stephanie Strom. A wild sleigh ride at Federal Express. *The New York Times*, December, 20. Pages C1-C2, 1994.

[Sun and Weld, 1992] Ying Sun and Daniel S. Weld. Beyond simple observation: Planning to diagnose. In *Proceedings of the Third International Workshop on Principles of Diagnosis, DX92*, pages 67–75, October 1992.

[Sussman, 1975] Gerald J. Sussman. *A Computer Model of Skill Acquisition*. American Elsevier, New York, 1975. Also available as technical report AI-TR-297, Artificial Intelligence Laboratory, MIT, 1975.

[Sycara and Miyashita, 1994] Katia Sycara and Kazuo Miyashita. Case-based acquisition of user preferences for solution improvement in ill-structured domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 44–49, Seattle, WA, July 1994. AAAI Press/The MIT Press.

[Tadepalli, 1989] Prasad Tadepalli. Lazy explanation-based learning: A solution to the intractable theory problem. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 694–700, Detroit, MI, 1989.

[Tadepalli, 1990] Prasad Tadepalli. *Tractable Learning and Planning in Games*. PhD thesis, Rutgers, The State University of New Jersey, Department of Computer Science, May 1990. Technical Report ML-TR-31.

[Tecucci, 1992] Gheorghe D. Tecucci. Automating knowledge acquisition as extending, updating, and improving a knowledge base. *IEEE Transactions on Systems, Man and Cybernetics*, 22(6), November/ December 1992.

[Thrun and Mitchell, 1994] Sebastian Thrun and Tom M. Mitchell. Learning one more thing. Technical Report CMU-CS-94-184, School of Computer Science, Carnegie Mellon University, September 1994.

[Veloso and Blythe, 1994] Manuela Veloso and Jim Blythe. Linkability: Examining causal link commitments in partial-order planning. In *Proceedings of the Second International Conference on AI Planning Systems, AIPS-94*, pages 170–175, Chicago, IL, June 1994.

[Veloso and Carbonell, 1993] Manuela M. Veloso and Jaime G. Carbonell. Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization. *Machine Learning*, 10:249–278, 1993.

[Veloso *et al.*, 1990] Manuela M. Veloso, M. Alicia Pérez, and Jaime G. Carbonell. Nonlinear planning with parallel resource allocation. In *Proceedings of the Darpa Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 207–212, San Diego, CA, November 1990. Morgan Kaufmann.

[Veloso *et al.*, 1995] Manuela Veloso, Jaime Carbonell, M. Alicia Pérez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1), January 1995.

[Veloso, 1989] Manuela M. Veloso. Nonlinear problem solving using intelligent casual-commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University, 1989.

[Veloso, 1994] Manuela M. Veloso. *Planning and Learning by Analogical Reasoning*. Springer Verlag, Berlin, Germany, 1994. PhD thesis available as technical report CMU-CS-92-174, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[Wang, 1995] Xuemei Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the Twelfth International Conference on Machine Learning*, Tahoe City, CA, 1995.

[Wellman, 1988] Michael P. Wellman. Formulation of tradeoffs in planning under uncertainty. Technical Report MIT/LCS/TR-427, Massachusetts Institute of Technology. Laboratory for Computer Science, August 1988. PhD Thesis.

[Wilensky, 1983] Robert Wilensky. *Planning and Understanding*. Addison-Wesley, Reading, MA, 1983.

[Wilkins, 1988] David C. Wilkins. Knowledge base refinement using apprenticeship learning techniques. In *Proceedings of the National Conference on Artificial Intelligence*, pages 646–651, St. Paul, MN, 1988.

[Williamson and Hanks, 1994] Mike Williamson and Steve Hanks. Optimal planning with a goal-directed utility model. In *Proceedings of the Second International Conference on AI Planning Systems, AIPS-94*, pages 176–181, Chicago, IL, June 1994.

[Yang *et al.*, 1992] Qiang Yang, Dana S. Nau, and James Hendler. Merging separately generated plans with restricted interactions. *Computational Intelligence*, 8(4), 1992.

[Zhang and Lu, 1992] Guangming Zhang and Stephen C-Y. Lu. An expert system framework for economic evaluation of machining operation planning. In F. Famili, S. Kim, and D. S. Nau, editors, *AI Applications in Manufacturing*, pages 133–156. AAAI/MIT Press, Menlo Park, CA, 1992. Also Technical Research Report, University of Maryland Systems Research Center, TR 89-87.

Appendix A

The PRODIGY Problem Solver

PRODIGY is a domain-independent problem solver that serves as a testbed for planning and machine learning research. Given an initial state and a goal expression, PRODIGY searches for a sequence of operators that will transform the initial state into a state that matches the goal expression. The current version of PRODIGY, PRODIGY4.0, is a nonlinear and complete planner. It follows a means-ends analysis backward chaining search procedure reasoning about multiple goals and multiple alternative operators relevant to achieving the goals. Detailed descriptions of PRODIGY4.0 appear in [Carbonell *et al.*, 1992, Veloso *et al.*, 1995].

PRODIGY4.0 provides a rich action representation language coupled with an expressive control language. A planning *domain* is defined by a set of types of objects, i.e., classes, used in the domain, and a library of operators and inference rules that act on these objects. Each operator is defined by its *preconditions* and *effects*. The description of preconditions and effects of an operator can contain typed variables. In addition variable bindings (i.e. values a variable can take) can be constrained by arbitrary Lisp functions. Preconditions in the operators can contain conjunctions, disjunctions, negations, and both existential and universal quantifiers. The *effects* of an operator consist of a list of predicates to be added or deleted from the state when the operator applies. An operator may also have *conditional* effects that are to be performed depending on particular state conditions. Inference rules deductively change a particular planning state by adding semantically redundant information to the state, in constrast to operators which specify *real* changes to the state. They have the same syntax as operators. PRODIGY4.0 allows two types of inference rules: *eager* inference rules fire automatically every time there is a change in the state whenever their preconditions are satisfied; they are used only in a forward-chaining manner. *Lazy* inference rules are used for backward chaining; they only fire on demand and PRODIGY4.0 subgoals on their preconditions if they are not true. A truth-maintenance system (TMS) keeps track of all the inference rules (both eager and lazy) that are fired. When an operator is applied, the effects of inference rules whose preconditions are no longer true are undone.

A *planning problem* is defined by (1) a set of available objects of each type, (2) an *initial state* I, and (3) a *goal statement* G. The initial state is represented as a set of literals. The goal statement is a logical formula equivalent to a preconditions expression, i.e. it can contain typed variables, conjunctions, negations, disjunctions, and universal and existential quantifications. A solution to a planning problem is a sequence of operators that can be applied to the initial state, transforming it into a state that satisfies the goal. A sequence of operators is called a *total-order plan*. A *partial-order plan*, that is, a partially ordered set of operators, can be obtained efficiently from the total-order plan [Veloso, 1989].

---

1. Check if the goal statement is true in the current state, or there is a reason to suspend the current search path.

   If yes, then either return the final plan or backtrack.

2. Compute the *set* of *pending goals* $\mathcal{G}$, and the set of possible *applicable operators* $\mathcal{A}$.
   A pending goal is a precondition of an operator previously expanded (in Step 4) that is not true in the current state. An applicable operator is an operator whose preconditions are true in the state.
3. Choose a goal G from $\mathcal{G}$ or select an operator A from $\mathcal{A}$ that is directly applicable.
4. If G has been chosen, then
   - get the set $\mathcal{O}$ of *relevant operators* for the goal,
   - choose an operator O from $\mathcal{O}$,
   - get the set $\mathcal{B}$ of possible *bindings* for O,
   - choose a set B of bindings from $\mathcal{B}$,
   - go to step 1.
5. If an operator A has been selected as directly applicable, then
   - *apply* A,
   - go to step 1.

Figure A.1: A skeleton of PRODIGY4.0's nonlinear planning algorithm (adapted from [Veloso, 1989]). Problem solving decisions, namely selecting which goal/subgoal to address next, which operator to apply, what bindings to select for the operator, or where to backtrack in case of failure, can be guided by control knowledge. PRODIGY's trace provides all the information about the decisions made during problem solving so it can be exploited by machine learning methods.

Table A.1 describes the basic search cycle of PRODIGY4.0's nonlinear planner [Veloso, 1989]. This search algorithm involves several decision points, namely:

- Which goal to subgoal on, from the set of pending goals.
- Which operator to choose in order to achieve a given goal.

- Which bindings to choose in order to instantiate the selected operator.
- Whether to apply an applicable operator (and which one) or defer application and continue subgoaling.

Control knowledge may direct the choices in each of these decision points. In PRODIGY, there is a clear division between the declarative domain knowledge (operators and inference rules) and the more procedural control knowledge. This simplifies both the initial specification of a domain and the incremental learning of the control knowledge. Control knowledge can take the form of control rules (usually domain-dependent), complete problem solving episodes to be used by analogy [Veloso, 1994], and domain-independent heuristics [Blythe and Veloso, 1992, Stone *et al.*, 1994].[1]

Control rules are productions (*if-then* rules) that indicate which choices should be made (or avoided) depending on the current state and other meta-level information based on previous choices or subgoaling links. They can be hand-coded by the user or automatically learned. They are divided into these three groups: *select*, *reject*, and *prefer* rules. *Select* and *reject* rules are used to prune parts of the search space, while *prefer* rules determine the order of exploring the remaining parts. Alternatives pruned by select and reject control rules are not tried should the planner backtrack to the node where the rule fired. Control rules choose goals, operators, bindings, or subgoaling versus apply. They can also choose nodes to backtrack to.

Chapter 3 of this dissertation presents algorithms to learn *prefer* control rules. Therefore it is worth describing those rules here in some detail. The right-hand side of a *prefer* rule indicates two alternatives, namely the one preferred, should the rule fire, and the one to be tried upon backtracking should the preferred alternative fail. A prefer rule is considered for matching only if the preferred-over alternative matches the current preference. If a prefer rule matches, it fires, that is, its preference is chosen as the current preference; then all the prefer rules are considered for matching again. Several prefer rules (or instantiations of the same rule) match giving conflicting preferences. A cycle in the rule preferences occur when there is a chain of rule firings $r_1, r_2, ...r_n$ such that $r_i, i = 1, ...n-1$ prefers alt_i over alt_{i-1} and r_n prefers alt_n over alt_1. As all the rules are considered again after one of them matches, the current preference is continually overriden by some rule. We have slightly modified PRODIGY4.0 so the cycle is detected, the set of preferred alternatives $\{alt_i, i = 1, ...n\}$ built, and one of them is chosen by some heuristic (see Section 3.11.2).

PRODIGY is designed with a "glass-box" approach: all the decisions made by the search engine and all the information available to make those decisions are captured in a problem's trace. This provides an information context in which learning can take place. Figure A.2 shows the learning modules developed in PRODIGY, according to their learning goal, namely: learn control knowledge to improve the planner's *efficiency* in reaching a solution to a problem [Minton, 1988,

[1] Chapter 4 of this thesis presents a fourth form, control knowledge trees.

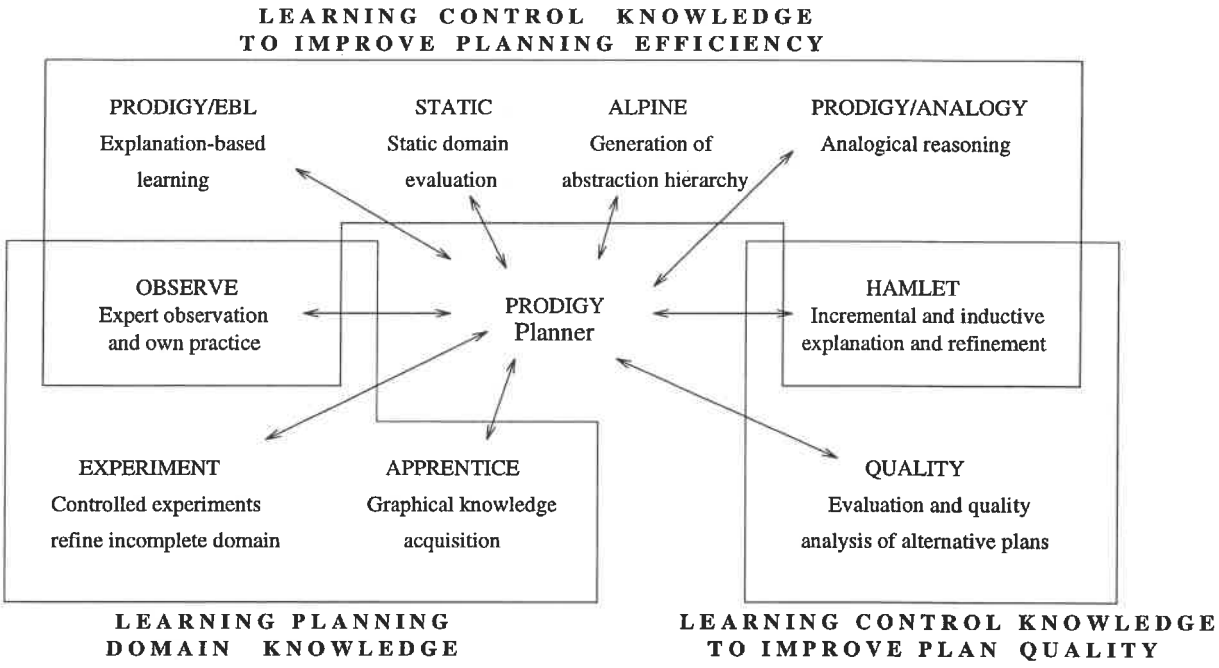


Figure A.2: The learning modules in the PRODIGY architecture (from [Veloso *et al.*, 1995]).

Etzioni, 1990, Pérez and Etzioni, 1992, Knoblock, 1994, Veloso, 1994, Borrajo and Veloso, 1994b]; learn control knowledge to improve the *quality* of the solutions produced by the planner([Borrajo and Veloso, 1994b, Iwamoto, 1994] and this thesis); and learn *domain* knowledge, i.e., learn or refine the set of operators specifying the domain [Gil, 1992, Wang, 1995], or acquire them through a graphical apprentice-like dialog [Joseph, 1992].

The machine learning and knowledge acquisition work supports PRODIGY's casual-commitment method[2], as it assumes there is *intelligent* control knowledge, exterior to its search cycle, that it can rely upon to make decisions, both to make planning more efficient and to obtain good quality plans. PRODIGY has been applied to a wide range of planning and problem-solving tasks: robotic path planning [Haigh *et al.*, 1994], the blocksworld, several versions of the STRIPS domain, matrix algebra manipulation, discrete machine-shop planning [Gil and Pérez, 1994] and scheduling, computer configuration, logistics transportation planning, and several others. Other research in the PRODIGY project has focused in studying different planning techniques and heuristics [Blythe and Veloso, 1992, Veloso and Blythe, 1994, Stone *et al.*, 1994, Blythe, 1994].

[2]In a casual-commitment strategy at each decision point the planner commits to a particular alternative, and backtracks upon failure. This is in contrast to a least-commitment strategy where decisions are deferred until all possible interactions are recognized.

Appendix B

The PRODIGY4.0 Process Planning Domain

This appendix provides the details of the process planning domain. A domain in PRODIGY4.0 is described by the type hierarchy (an ontology of the classes of objects in the domain) and a set of operators and inference rules. Figure B.1 shows the type hierarchy for the process planning domain. Then we list all the operators and inference rules of the domain relevant to the examples used throughout this thesis. We also list the hand-coded control rules for planning *efficiency*, that is, to prune search paths leading to dead-ends, that we used during the experiments. Those control rules are not relevant to plan quality and their effect is orthogonal to that of the learned quality-enhancing control knowledge.

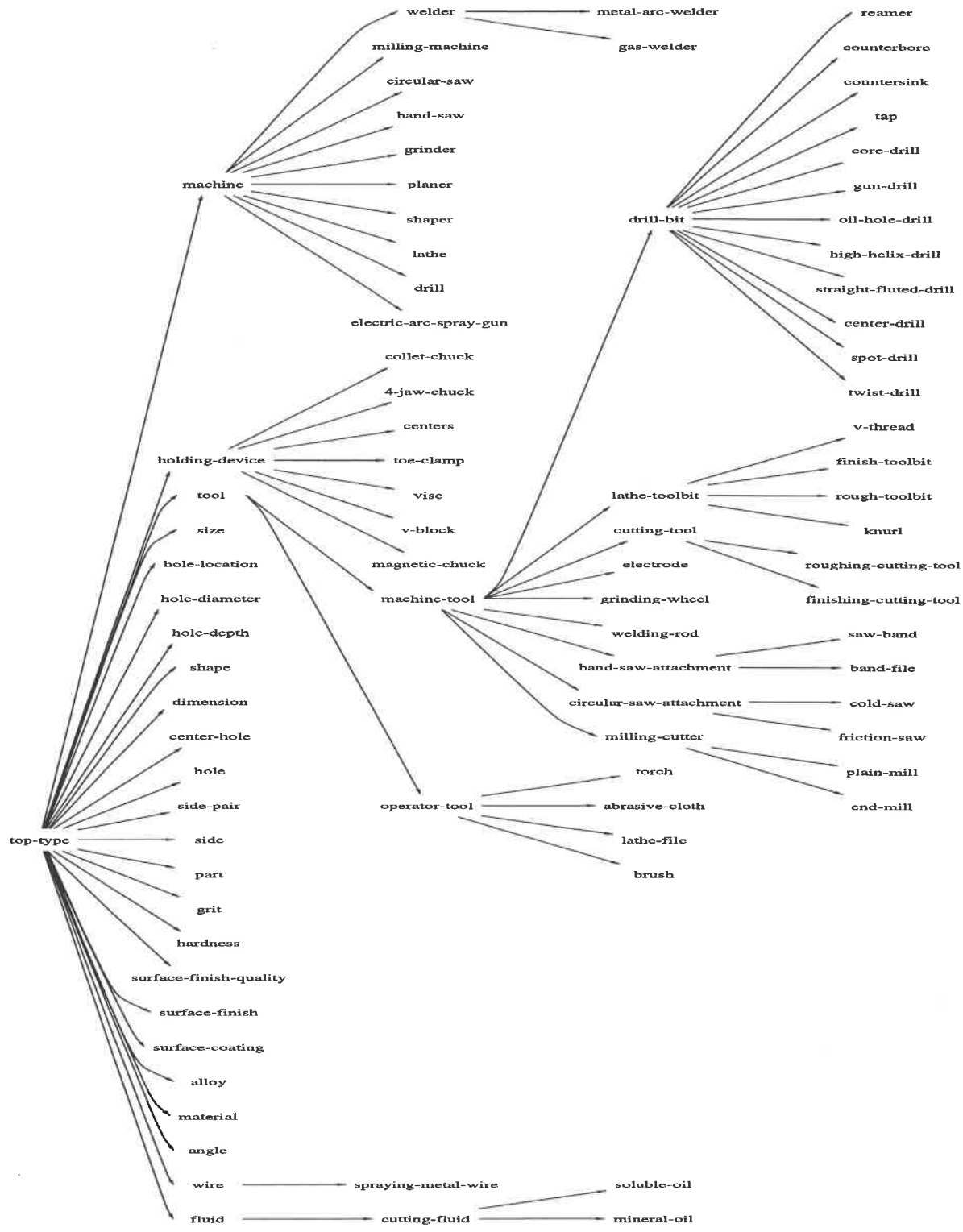


Figure B.1: The type hierarchy for the process planning domain.

```
;;; Reference: "A Specification of Manufacturing Processes for Planning", Yolanda Gil.
;;; Technical report CMU-CS-91-179. School of Computer Science, Carnegie Mellon Univ.
;;;
;;; Translated for PRODIGY4.0 by Alicia Perez, May 1992
;;; Only the operators, inference rules, and types used in the examples throughout the
;;; thesis are shown here.
;;;*************************************************************
(create-problem-space 'hd-machining :current t)

(setf (getf (p4::problem-space-plist *current-problem-space*) :depth-bound) 500)

;;;*************************************************************
;;; Types
;;;*************************************************************

(ptype-of Machine :Top-Type)        ;;other machines:  Lathe, Shaper, Planer, Grinder,
(ptype-of Drill Machine)            ;;Band-Saw, Circular-Saw, Welder (Metal-Arc-Welder,
(ptype-of Milling-Machine Machine)  ;;Gas-Welder), Electric-Arc-Spray-Gun

(ptype-of Holding-Device :Top-Type) ;;other holding devices: V-Block, Toe-clamp, Centers
(ptype-of Vise Holding-Device)      ;;4-Jaw-Chuck, Collet-Chuck, Magnetic-Chuck

(ptype-of Tool :Top-Type)
(ptype-of Machine-Tool Tool)
(ptype-of Operator-Tool Tool)

;;;Kinds of Machine-Tools
(ptype-of Drill-Bit Machine-Tool)      ;;other machine-tools: Electrode, Welding-Rod
(ptype-of Cutting-Tool Machine-Tool)   ;;Lathe-Toolbit, Grinding-Wheel,
(ptype-of Milling-Cutter Machine-Tool) ;;Band-Saw-Attachment, Circular-Saw-Attachment

;;;Drill-Bits
(ptype-of Spot-Drill Drill-Bit)        ;;other drill-bits: Center-Drill, Straight-Fluted
-Drill,
(ptype-of High-Helix-Drill Drill-Bit)  ;;Oil-Hole-Drill, Gun-Drill, Core-Drill,
(ptype-of Tap Drill-Bit)
(ptype-of Countersink Drill-Bit)
(ptype-of Counterbore Drill-Bit)
(ptype-of Reamer Drill-Bit)
(ptype-of Twist-Drill Drill-Bit)

;;;Lathe Toolbits: Rough-Toolbit, Finish-Toolbit, V-Thread, Knurl

;;;Cutting Tools
(ptype-of Roughing-Cutting-Tool Cutting-Tool)
(ptype-of Finishing-Cutting-Tool Cutting-Tool)

;;;Circular Saw Attachments: Cold-Saw, Friction-Saw
;;;Band-Saw-Attachment: Saw-Band, Band-File

;;;Milling Cutters
(ptype-of Plain-Mill Milling-Cutter)
(ptype-of End-Mill Milling-Cutter)

;;;Operator Tools
(ptype-of Brush Operator-Tool) ;;other operator-tools: Lathe-File, Abrasive-Cloth, Torch
;;;Wires: Spraying-Metal-Wire

;;;Fluids
(ptype-of Fluid :Top-Type)             (ptype-of Cutting-Fluid Fluid)
(ptype-of Soluble-Oil Cutting-Fluid)   (ptype-of Mineral-Oil Cutting-Fluid)

;;;Materials
(ptype-of Material :Top-Type)
(pinstance-of COPPER ALUMINUM STEEL STAINLESS-STEEL BRASS EMERY BRONZE
              IRON TUNGSTEN MOLYBDENUM ZIRCONIUM-OXIDE ALUMINUM-OXIDE Material)

;;;Alloys
(ptype-of Alloy :Top-Type)             (pinstance-of FERROUS NON-FERROUS Alloy)

;;;Surface coatings
(ptype-of Surface-Coating :Top-Type)
(pinstance-of CORROSION-RESISTANT HEAT-RESISTANT WEAR-RESISTANT FUSED-METAL
              Surface-Coating)

;;;Surface finishes
(ptype-of Surface-finish :Top-Type)
(pinstance-of COLD-ROLLED POLISHED SAWCUT TAPPED KNURLED
              FINISH-MILL FINISH-TURN FINISH-SHAPED FINISH-GRIND FINISH-PLANED
              ROUGH-MILL ROUGH-TURN ROUGH-SHAPED ROUGH-GRIND ROUGH-PLANED Surface-finish)

;;;Surface finish qualities
(ptype-of Surface-finish-quality :Top-Type)
(pinstance-of MACHINED GROUND surface-finish-quality)

;;;Hardnesses of materials
(ptype-of Hardness :Top-Type)     (pinstance-of HARD SOFT Hardness)

;;;Grits of wheel
(ptype-of Grit :Top-Type)         (pinstance-of FINE-GRIT COARSE-GRIT Grit)

;;; Parts
(ptype-of Part :Top-Type)
(ptype-of Side :Top-Type)         (pinstance-of SIDE0 SIDE1 SIDE2 SIDE3 SIDE4 SIDE5 SIDE6
Side)
(ptype-of Side-Pair :Top-Type)    (pinstance-of SIDE1-SIDE4 SIDE2-SIDE5 SIDE3-SIDE6
                                                Side-Pair)

;;;Holes
(ptype-of Hole :Top-Type)
(ptype-of Center-Hole :Top-Type)  (pinstance-of CENTER-HOLE-SIDE3 CENTER-HOLE-SIDE6
                                                Center-Hole)

;;;Part Dimensions
(ptype-of Dimension :Top-Type)    (pinstance-of WIDTH LENGTH HEIGHT DIAMETER Dimension)

;;;Part shapes
(ptype-of Shape :Top-Type)        (pinstance-of RECTANGULAR CYLINDRICAL Shape)

(infinite-type Hole-Depth #'numberp)     (infinite-type Hole-Diameter #'numberp)
(infinite-type Hole-Location #'numberp)  (infinite-type Size #'numberp)
(infinite-type Angle #'numberp)

;;;*************************************************************
;;; Operators
;;;*************************************************************
;;; MACHINE: DRILL
;;; only these devices can be used on a drill (see operator PUT-HOLDING-DEVICE-IN-DRILL)
;;; -  4-JAW-CHUCK V-BLOCK VISE TOE-CLAMP
;;; Tool has to be of type Drill-bit

; operators for making holes

(Operator DRILL-WITH-SPOT-DRILL
 (params <machine> <drill-bit> <holding-device> <part> <hole> <side>
         <side-pair> <loc-x> <loc-y>)
 (preconds ((<machine> Drill)
            (<drill-bit> Spot-Drill)
```

```
                (<holding-device> (or 4-JAW-CHUCK VISE TOE-CLAMP)
                 (<part> Part)
                 (<side> Side) ;side up
                 (<side-pair> Side-Pair) ;sides to holding-device
                 (<hole> Hole)
                 (<loc-x> (and Hole-Location (x-location-of <part> <loc-x>)))
                 (<loc-y> (and Hole-Location (y-location-of <part> <loc-y>))))
    (and (sides-for-holding-device <side> <side-pair>)
         (holding-tool <machine> <drill-bit>)
         (holding <machine> <holding-device> <part> <side> <side-pair>))
    (effects () ((del (is-clean <part>))
                 (add (has-burrs <part>))
                 (add (has-spot <part> <hole> <side> <loc-x> <loc-y>))))
)

(Operator DRILL-WITH-TWIST-DRILL
 (params <machine> <drill-bit> <holding-device> <part> <hole>
         <side> <side-pair> <hole-depth> <hole-diameter>
         <drill-bit-diameter> <loc-x> <loc-y>)
 (preconds ((<machine> Drill)
            (<drill-bit> Twist-Drill)
            (<holding-device> (or 4-JAW-CHUCK VISE TOE-CLAMP)
            (<hole> Hole)
            (<drill-bit-diameter>
              (and Hole-Diameter
                   (gen-from-pred (diameter-of-drill-bit <drill-bit> <drill-bit-diameter>
))))
           (<hole-diameter> (and Hole-Diameter (same <hole-diameter> <drill-bit-diamete
r>)))
           (<side> Side)
           (<side-pair> Side-Pair)       ;sides to holding-device.
           (<part> Part)
           (<loc-x> (and Hole-Location (x-location-of <part> <loc-x>)))
           (<loc-y> (and Hole-Location (y-location-of <part> <loc-y>)))
           (<hole-depth> Hole-Depth))
 (and (sides-for-holding-device <side> <side-pair>)
      (has-spot <part> <hole> <side> <loc-x> <loc-y>)
      (holding-tool <machine> <drill-bit>)
      (holding <machine> <holding-device> <part> <side> <side-pair>)))
 (effects () ((del (is-clean <part>))
              (add (has-burrs <part>))
              (add (has-hole <part> <hole> <side> <hole-depth> <hole-diameter>
                             <loc-x> <loc-y>))))
)

(Operator DRILL-WITH-HIGH-HELIX-DRILL
 (params <machine> <drill-bit> <holding-device> <part> <hole>
         <side> <side-pair> <fluid> <hole-depth> <hole-diameter>
         <loc-x> <loc-y>)
 (preconds ((<machine> Drill)
            (<drill-bit> High-Helix-Drill)
            (<holding-device> (or 4-JAW-CHUCK VISE TOE-CLAMP))
            (<fluid> Fluid)
            (<part> Part)
            (<side> Side)
            (<side-pair> Side-Pair)      ;sides to holding-device
            (<hole> Hole)
            (<loc-x> (and Hole-Location (x-location-of <part> <loc-x>)))
            (<loc-y> (and Hole-Location (y-location-of <part> <loc-y>)))
            (<hole-depth> Hole-Depth)
            (<hole-diameter>
              (and Hole-Diameter
                   (gen-from-pred (diameter-of-drill-bit <drill-bit> <hole-diameter>)))))
 (and (sides-for-holding-device <side> <side-pair>)
      (has-hole <part> <hole> <side> <loc-x> <loc-y>)
      (has-fluid <machine> <fluid> <part>)
      (has-spot <part> <hole> <side> <loc-x> <loc-y>)
                (holding-tool <machine> <drill-bit>)
                (holding <machine> <holding-device> <part> <side> <side-pair>))
 (effects () ((del (is-clean <part>))
              (add (has-burrs <part>))
              (add (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <l
oc-y>)))))
 (add (has-burrs <part>))))
 (effects () ((del (is-clean <part>))
              (add (has-burrs <part>))
              (add (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <l
oc-y>)))))

;; operators for finishing holes

(Operator TAP
 (params <machine> <drill-bit> <holding-device> <part> <hole> <side>
         <side-pair> <hole-depth> <hole-diameter> <drill-bit-diameter>
         <loc-x> <loc-y>)
 (preconds ((<machine> DRILL)
            (<drill-bit> TAP)
            (<holding-device> (or 4-JAW-CHUCK VISE TOE-CLAMP))
            (<part> PART)
            (<hole> HOLE)
            (<side> Side)
            (<side-pair> Side-Pair)          ;sides to holding-device
            (<hole-depth> Hole-Depth)
            (<drill-bit-diameter>
              (and Hole-Diameter
                   (gen-from-pred (diameter-of-drill-bit <drill-bit> <drill-bit-diameter>)
)))
           (<hole-diameter> (and Hole-Diameter (same <drill-bit-diameter> <hole-diameter
>)))
           (<loc-x> (and Hole-Location (x-location-of <part> <loc-x>)))
           (<loc-y> (and Hole-Location (y-location-of <part> <loc-y>))))
 (and (sides-for-holding-device <side> <side-pair>)
      (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>)
      (holding-tool <machine> <drill-bit>)
      (~ (has-burrs <part>))
      (is-clean <part>)
      (holding <machine> <holding-device> <part> <side> <side-pair>))
 (effects ()
          ((del (is-clean <part>))
           (add (has-burrs <part>))
           (if (is-reamed <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>
)))
           ((del (is-reamed <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>
))))
 (add (is-tapped <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>))))

(Operator COUNTERSINK
 (params <machine> <drill-bit> <holding-device> <part> <hole> <side>
         <side-pair> <hole-depth> <hole-diameter> <angle> <loc-x>
         <loc-y>)
 (preconds ((<machine> DRILL)
            (<drill-bit> COUNTERSINK)
            (<holding-device> (or 4-JAW-CHUCK VISE TOE-CLAMP))
            (<part> PART)
            (<hole> Hole)
            (<side> Side)
            (<side-pair> Side-Pair)          ;sides to holding-device
            (<hole-depth> Hole-Depth)
            (<hole-diameter> Hole-Diameter)
            (<loc-x> (and Hole-Location (x-location-of <part> <loc-x>)))
            (<loc-y> (and Hole-Location (y-location-of <part> <loc-y>)))
            (<angle>
              (and Angle (gen-from-pred (angle-of-drill-bit <drill-bit> <angle>)))))
 (and (sides-for-holding-device <side> <side-pair>)
      (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>)
      (holding-tool <machine> <drill-bit>)
      (~ (has-burrs <part>))
```

```
      (is-clean <part>)
      (holding <machine> <holding-device> <part> <side> <side-pair>)))
 (effects ()
   ((del (is-clean <part>))
   (add (has-burrs <part>))
   (add (is-countersinked <part> <hole> <side> <hole-depth> <hole-diameter>
                          <loc-x> <loc-y> <angle>)))))

(Operator COUNTERBORE
 (params <machine> <drill-bit> <holding-device> <part> <hole> <side>
         <side-pair> <hole-depth> <hole-diameter> <counterbore-size>
         <loc-x> <loc-y>)
 (preconds ((<machine> DRILL)
            (<drill-bit> COUNTERBORE)
            (<holding-device> (or 4-JAW-CHUCK  VISE TOE-CLAMP))
            (<part> Part)
            (<hole> Hole)
            (<side> Side)
            (<side-pair> Side-Pair)        ;sides to holding-device
            (<counterbore-size>
             (and Hole-Diameter (gen-from-pred
                                 (size-of-drill-bit <drill-bit> <counterbore-size>))))
            (<hole-depth> Hole-Depth)
            (<hole-diameter> Hole-Diameter)
            (<loc-x> Hole-Location)
            (<loc-y> Hole-Location))
  (and (sides-for-holding-device <side> <side-pair>)
       (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>)
       (holding-tool <machine> <drill-bit>)
       (- (has-burrs <part>))
       (is-clean <part>)
       (holding <machine> <holding-device> <part> <side> <side-pair>)))
 (effects () ((del (is-clean <part>))
              (add (has-burrs <part>))
              (add (is-counterbored <part> <hole> <side> <hole-depth> <hole-diameter>
                                    <loc-x> <loc-y> <counterbore-size>)))))

(Operator REAM
 (params <machine> <drill-bit> <holding-device> <part> <hole>
         <side> <side-pair> <fluid> <hole-depth> <hole-diameter>
         <drill-bit-diameter> <loc-x> <loc-y>)
 (preconds ((<machine> DRILL)
            (<drill-bit> REAMER)
            (<holding-device> (or 4-JAW-CHUCK  VISE TOE-CLAMP))
            (<part> PART)
            (<hole> HOLE)
            (<side> Side)
            (<side-pair> Side-Pair)        ;sides to holding-device
            (<fluid> FLUID)
            (<hole-depth> (and Hole-Depth (smaller <hole-depth> 2)))
            (<loc-x> Hole-Location)
            (<loc-y> Hole-Location)
            (<drill-bit-diameter>
             (and Hole-Diameter
                  (gen-from-pred (diameter-of-drill-bit <drill-bit> <drill-bit-diameter>
))))
            (<hole-diameter>
             (and Hole-Diameter
                  (same <hole-diameter> <drill-bit-diameter>))))
  (and (sides-for-holding-device <side> <side-pair>)
       (has-fluid <machine> <fluid> <part>)
       (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>)
       (holding-tool <machine> <drill-bit>)
       (- (has-burrs <part>))
       (is-clean <part>)
       (holding <machine> <holding-device> <part> <side> <side-pair>)))
 (effects ()
   ((del (is-clean <part>))
   (add (has-burrs <part>))
   (del (is-tapped <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>))
   (add (is-reamed <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>)))))

;;;*********************************************************
;;; MACHINE: MILLING MACHINE
;;; only these devices can be used on a milling machine  (see
;;; PUT-HOLDING-DEVICE-IN-MILLING-MACHINE)
;;; -  4-JAW-CHUCK V-BLOCK VISE COLLET-CHUCK TOE-CLAMP
;;; Tools can be:
;;; - Milling-Cutter Drill-bit

;;; side-mill is also known as end-mill.

(Operator FACE-MILL
 (params <machine> <part> <milling-cutter> <holding-device>
         <side> <side-pair> <dim> <value-old> <value>)
 (preconds ((<machine> MILLING-MACHINE)
            (<milling-cutter> MILLING-CUTTER)
            (<holding-device> (or 4-JAW-CHUCK VISE COLLET-CHUCK TOE-CLAMP))
            (<part> Part)
            (<dim> Dimension)
            (<side> Side)
            (<side-pair> Side-Pair)        ;sides to holding-device
            (<value-old>
             (and Size (gen-from-pred (size-of <part> <dim> <value-old>))))
            (<value> (and Size (smaller <value> <value-old>))))
  (and (shape-of <part> RECTANGULAR)
       (side-up-for-machining <dim> <side>)
       (sides-for-holding-device <side> <side-pair>)
       (holding-tool <machine> <milling-cutter>)
       (holding <machine> <holding-device> <part> <side> <side-pair>)))
 (effects ((<surface-coating> Surface-coating)
           (<surface-finish> Surface-finish))
          ((del (is-clean <part>))
           (add (has-burrs <part>))
           (del (surface-coating-side <part> <side> <surface-coating>))
           (del (surface-finish-side <part> <side> <surface-finish>))
           (add (surface-finish-side <part> <side> ROUGH-MILL))
           (add (size-of <part> <dim> <value>))
           (del (size-of <part> <dim> <value-old>)))))

(Operator SIDE-MILL
 (params <machine> <part> <milling-cutter> <holding-device>
         <side> <side-pair> <mach-side> <dim> <value-old> <value>)
 (preconds ((<machine> MILLING-MACHINE)
            (<part> Part)
            (<milling-cutter> MILLING-CUTTER)
            (<holding-device> (or 4-JAW-CHUCK  VISE COLLET-CHUCK TOE-CLAMP))
            (<dim> Dimension)
            (<side> Side)                  ;side facing up ;(one-of <side> '(SIDE1 SIDE4))
            (<side-pair> Side-Pair)        ;sides to holding-device
            (<mach-side>                   ;side touched by the operation
             (and Side (not-in-side-pair <mach-side> <side-pair>)))
            (<value-old>
             (and Size (gen-from-pred (size-of <part> <dim> <value-old>))))
            (<value> (and Size (smaller <value> <value-old>)
                               (smaller-than-2in <value-old> <value>))))
  (and (side-for-side-mill <dim> <side> <mach-side>)
       (sides-for-holding-device <side> <side-pair>)
```

```
                (holding-tool <machine> <milling-cutter>)
                (holding <machine> <holding-device> <part> <side> <side-pair>))
   (effects ((<surface-coating> Surface-coating)
             (<surface-finish> Surface-finish))
            ((del (is-clean <part>))
             (add (has-burrs <part>))
             (del (surface-coating-side <part> <mach-side> <surface-coating>))
             (del (surface-finish-side <part> <mach-side> <surface-finish>))
             (add (surface-finish-side <part> <mach-side> ROUGH-MILL))
             (add (size-of <part> <dim> <value>))
             (del (size-of <part> <dim> <value-old>)))))

(Operator DRILL-WITH-SPOT-DRILL-IN-MILLING-MACHINE
 (params <machine> <drill-bit> <holding-device> <part> <hole> <side>
         <side-pair> <loc-x> <loc-y>)
 (preconds ((<machine> MILLING-MACHINE)
            (<drill-bit> SPOT-DRILL)
            (<holding-device> (or 4-JAW-CHUCK VISE COLLET-CHUCK TOE-CLAMP))
            (<part> Part)
            (<side> Side)
            (<side-pair> Side-Pair)     ;sides to holding-device
            (<loc-x> (and Hole-Location (x-location-of <part> <loc-x>)))
            (<loc-y> (and Hole-Location (y-location-of <part> <loc-y>)))
            (<hole> Hole)
            (and (sides-for-holding-device <side> <side-pair>)
                 (holding-tool <machine> <drill-bit>)
                 (holding <machine> <holding-device> <part> <side> <side-pair>)))
  (effects () ((del (is-clean <part>))
               (add (has-burrs <part>))
               (add (has-spot <part> <hole> <side> <loc-x> <loc-y>)))))

(Operator DRILL-WITH-TWIST-DRILL-IN-MILLING-MACHINE
 (params <machine> <drill-bit> <holding-device> <part> <hole>
         <side> <side-pair> <hole-depth> <hole-diameter>
         <drill-bit-diameter> <loc-x> <loc-y>)
 (preconds ((<machine> MILLING-MACHINE)
            (<drill-bit> TWIST-DRILL)
            (<holding-device> (or 4-JAW-CHUCK VISE COLLET-CHUCK TOE-CLAMP))
            (<part> Part)
            (<side> Side)
            (<side-pair> Side-Pair)     ;sides to holding-device
            (<hole> Hole)
            (<drill-bit-diameter>
             (and Hole-Diameter (gen-from-pred
                                 (diameter-of-drill-bit <drill-bit> <drill-bit-diameter>
))))
            (<hole-diameter>
             (and Hole-Diameter (same <drill-bit-diameter> <hole-diameter>)))
            (<loc-x> Hole-Location)
            (<loc-y> Hole-Location)
            (<hole-depth> Hole-Depth)
            (and
             (sides-for-holding-device <side> <side-pair>)
             (has-spot <part> <hole> <side> <loc-x> <loc-y>)
             (holding-tool <machine> <drill-bit>)
             (holding <machine> <holding-device> <part> <side> <side-pair>)))
  (effects () ((del (is-clean <part>))
               (add (has-burrs <part>))
               (add (has-hole <part> <hole> <side> <hole-depth> <hole-diameter>
                     <loc-x> <loc-y>)))))

;;;*****************************************************************
;;; OTHER OPERATIONS

(Operator CLEAN
 (params <part>)
 (preconds ((<part> PART)
            (is-available-part <part>))
  (effects () ((add (is-clean <part>)))))

(Operator REMOVE-BURRS
 (params <part> <brush>)
 (preconds ((<brush> BRUSH)
            (<part> PART)
            (is-available-part <part>)
            (has-burrs <part>)))
  (effects () ((del (is-clean <part>))
               (del (has-burrs <part>))))))

;;;*****************************************************************
; operators for preparing the machines
;;;*****************************************************************
; tools in machines

;;this allows any kind of drill-bit to be held by the milling machine,
;;but the ops only can use spot-drills and twist-drills.

(Operator PUT-TOOL-ON-MILLING-MACHINE
 (params <machine> <attachment>)
 (preconds ((<machine> MILLING-MACHINE) (<attachment> (or MILLING-CUTTER DRILL-BIT)))
  (and (is-available-tool-holder <machine>)
       (is-available-tool <attachment>)))
  (effects () ((add (holding-tool <machine> <attachment>)))))

(Operator PUT-IN-DRILL-SPINDLE
 (params <machine> <drill-bit>)
 (preconds ((<machine> DRILL) (<drill-bit> DRILL-BIT)
  (and (is-available-tool-holder <machine>)
       (is-available-tool <drill-bit>)))
  (effects () ((add (holding-tool <machine> <drill-bit>))))))

(Operator REMOVE-TOOL-FROM-MACHINE
 (params <machine> <tool>)
 (preconds ((<machine> MACHINE) (<tool> TOOL)
  (holding-tool <machine> <tool>))
  (effects () ((del (holding-tool <machine> <tool>))))))

;;;*****************************************************************
; holding devices in machines

(Operator PUT-HOLDING-DEVICE-IN-MILLING-MACHINE
 (params <machine> <holding-device>)
 (preconds ((<machine> MILLING-MACHINE)
            (<holding-device> (or 4-JAW-CHUCK V-BLOCK VISE COLLET-CHUCK TOE-CLAMP))
  (and (is-available-table <machine> <holding-device>)
       (is-available-holding-device <holding-device>)))
  (effects () ((add (has-device <machine> <holding-device>))))))

(Operator PUT-HOLDING-DEVICE-IN-DRILL
 (params <machine> <holding-device>)
 (preconds ((<machine> DRILL) (<holding-device> (or 4-JAW-CHUCK V-BLOCK VISE TOE-CLAMP))
  (and (is-available-table <machine> <holding-device>)
       (is-available-holding-device <holding-device>)))
  (effects () ((add (has-device <machine> <holding-device>))))))

(Operator REMOVE-HOLDING-DEVICE-FROM-MACHINE
 (params <machine> <holding-device>)
 (preconds ((<machine> Machine) (<holding-device> Holding-device))
  (and (has-device <machine> <holding-device>)
```

```
       (is-empty-holding-device <holding-device> <machine>)))
 (effects () ((del (has-device <machine> <holding-device>)))))

;;;*************************************************************
;;; cutting fluid in machines
;;; The fluid type depends on the material:
;;; - iron:              mineral-oil
;;; - steel, aluminum:   soluble-oil
;;; - brass, copper, bronze: any cutting fluid

(Operator ADD-SOLUBLE-OIL
 (params <machine> <fluid>)
 (preconds ((<machine> Machine) (<part> Part) (<fluid> SOLUBLE-OIL))
  (or (material-of <part> STEEL)
      (material-of <part> ALUMINUM)))
 (effects () ((add (has-fluid <machine> <fluid> <part>)))))

(Operator ADD-MINERAL-OIL
 (params <machine> <fluid>)
 (preconds ((<machine> Machine) (<part> Part) (<fluid> MINERAL-OIL))
  (material-of <part> IRON))
 (effects () ((add (has-fluid <machine> <fluid> <part>)))))

(Operator ADD-ANY-CUTTING-FLUID
 (params <machine> <fluid>)
 (preconds ((<machine> Machine) (<part> Part) (<fluid> CUTTING-FLUID))
  (or (material-of <part> BRASS)
      (material-of <part> BRONZE)
      (material-of <part> COPPER)))
 (effects () ((add (has-fluid <machine> <fluid> <part>)))))

;;;*************************************************************
;;; operators for holding parts with a device in a machine

(Operator PUT-ON-MACHINE-TABLE
 (params <machine> <part>)
 (preconds ((<machine> (COMP Machine Shaper)) ;complement
            (<part> Part))
  (and (is-available-part <part>)
       (is-available-machine <machine>)))
 (effects ((<another-machine> Machine))
          ((del (on-table <another-machine> <part>))
           (add (on-table <machine> <part>)))))

(Operator HOLD-WITH-VISE
 (params <machine> <holding-device> <part> <side> <side-pair>)
 (preconds ((<holding-device> VISE)
            (<machine> Machine)
            (<part> Part)
            (<side> Side) ;(<side-hd1> Side)(<side-hd2> Side)
            (<side-pair> Side-Pair))
  (and (has-device <machine> <holding-device>)
       (~ (has-burrs <part>))
       (is-clean <part>)
       (on-table <machine> <part>)
       (is-empty-holding-device <holding-device> <machine>)
       (is-available-part <part>)))
 (effects ()
   (del (on-table <machine> <part>))
   (if (shape-of <part> CYLINDRICAL)
       ((add (holding-weakly <machine> <holding-device> <part> <side> <side-pair>))))
   (if (shape-of <part> RECTANGULAR)
       ((add (holding <machine> <holding-device> <part> <side> <side-pair>))))))

(Operator REMOVE-FROM-MACHINE-TABLE
  (params <machine> <part>)
  (preconds ((<machine> MACHINE) (<part> Part))
   (and (on-table <machine> <part>)
        (is-available-part <part>)))
  (effects () ((del (on-table <machine> <part>)))))

(Operator RELEASE-FROM-HOLDING-DEVICE
 (params <machine> <holding-device> <part> <side> <side-pair>)
 (preconds ((<machine> MACHINE)
            (<holding-device> Holding-device)
            (<part> PART)
            (<side> SIDE) (<side-pair> SIDE-PAIR))
  (holding <machine> <holding-device> <part> <side> <side-pair>))
 (effects () ((del (holding <machine> <holding-device> <part> <side> <side-pair>))
              (add (on-table <machine> <part>)))))

;;other holding operators (we have omitted their preconditions and effects
;;because they are not used in the examples throughout the thesis).
(Operator HOLD-WITH-V-BLOCK ...
(Operator HOLD-WITH-TOE-CLAMP-CYLINDRICAL-PARTS
(Operator HOLD-WITH-TOE-CLAMP-RECTANGULAR-PARTS
(Operator SECURE-WITH-TOE-CLAMP
(Operator HOLD-WITH-CENTERS
(Operator HOLD-WITH-4-JAW-CHUCK
(Operator HOLD-WITH-COLLET-CHUCK
(Operator HOLD-WITH-MAGNETIC-CHUCK
(Operator RELEASE-FROM-HOLDING-DEVICE-WEAK

;;;*************************************************************
;;; Inference Rules
;;;*************************************************************

;;; inference rules for availability

(Inference-Rule MACHINE-AVAILABLE
 (params <machine>)
 (preconds  ((<machine> Machine))
  (forall ((<part> Part)(<holding-device> Holding-Device)
           (<side> Side)
           (<side-pair>
            (and Side-Pair (gen-from-pred (sides-for-holding-device <side> <side-pair>)))
))
          (and (~ (on-table <machine> <part>))
               (~ (holding <machine> <holding-device> <part> <side>
                           <side-pair>)))))
 (effects () ((add (is-available-machine <machine>)))))

(Inference-Rule MACHINE-NOT-AVAILABLE
 (mode eager)
 (params <machine>)
 (preconds ((<machine> Machine) (<other-part> Part))
  (on-table <machine> <other-part>))
 (effects () ((del (is-available-machine <machine>)))))

(Inference-Rule TOOL-HOLDER-AVAILABLE
 (params <machine>)
 (preconds ((<machine> Machine))
  (forall ((<tool> Tool))
          (~ (holding-tool <machine> <tool>))))
 (effects () ((add (is-available-tool-holder <machine>)))))

(Inference-Rule TOOL-AVAILABLE
 (params <tool>)
```

```
(preconds ((<tool> Tool))
 (forall ((<machine> Machine)
          (~(holding-tool <machine> <tool>)))))
 (effects () ((add (is-available-tool <tool>)))))

(Inference-Rule TOOL-AND-TOOL-HOLDER-NOT-AVAILABLE
 (mode eager)
 (params <machine>)
 (preconds ((<machine> Machine) (<tool> Tool))
  (holding-tool <machine> <tool>))
 (effects () ((del (is-available-tool-holder <machine>)
                   (del (is-available-tool <tool>))))))

(Inference-Rule TABLE-AVAILABLE1
 (params <machine>)
 (preconds ((<machine> Machine)
  (<holding-device> TOE-CLAMP))
  t)
 (effects () ((add (is-available-table <machine> <holding-device>)))))

(Inference-Rule TABLE-AVAILABLE2
 (params <machine>)
 (preconds ((<machine> Machine) (<holding-device> Holding-Device))
  (forall ((<another-holding-device> Holding-device))
          (~ (has-device <machine> <another-holding-device>))))
 (effects () ((add (is-available-table <machine> <holding-device>)))))

(Inference-Rule HOLDING-DEVICE-AVAILABLE
 (params <holding-device>)
 (preconds ((<holding-device> Holding-device))
  (forall ((<machine> Machine)
           (~(has-device <machine> <holding-device>))))
 (effects () ((add (is-available-holding-device <holding-device>)))))

(Inference-Rule TABLE-AND-HOLDING-DEVICE-NOT-AVAILABLE
 (mode eager)
 (params <machine>)
 (preconds ((<machine> Machine) (<another-holding-device> Holding-device))
  (has-device <machine> <another-holding-device>))
 (effects
 ;;if the holding device is a Toe-Clamp, the table is available
  ((<holding-device>
    (and (comp Holding-Device TOE-CLAMP)
         (diff <holding-device> <another-holding-device>))))
   ((del (is-available-table <machine> <holding-device>))
    (del (is-available-holding-device <another-holding-device>)))))

(Inference-Rule PART-AVAILABLE
 (params <part>)
 (preconds ((<part> Part))
  (forall
   ((<machine> Machine)
    (<holding-device> Holding-device)
    (<side> Side)
    (<side-pair> ;;Side-Pair
     (and Side-Pair (compute-side-pair <side> <side-pair>)) ;;slower
          (and Side-Pair (gen-from-pred (sides-for-holding-device <side> <side-pair>)))))
   (and
    (~(holding-weakly <machine> <holding-device> <part> <side> <side-pair>))
    (~(holding <machine> <holding-device> <part> <side> <side-pair>))))
 (effects () ((add (is-available-part <part>))))))

(Inference-Rule PART-NOT-AVAILABLE-AND-HOLDING-DEVICE-NOT-EMPTY-AND-MACHINE-NOT-AVAILABLE
 (mode eager)
 (params <part>)
 (preconds (<part> Part)
           (<machine> Machine)
           (<holding-device> Holding-device)
           (<side> Side) (<side-pair> Side-Pair))
 (or (holding-weakly <machine> <holding-device> <part> <side> <side-pair>)
     (holding <machine> <holding-device> <part> <side> <side-pair>))
 (effects () ((del (is-available-part <part>))
              (del (is-empty-holding-device <holding-device> <machine>))
              (del (is-available-machine <machine>)))))

(Inference-Rule HOLDING-DEVICE-EMPTY
 (params <machine> <holding-device>)
 (preconds (<holding-device> Holding-device) (<machine> Machine)
  (forall
   ((<part> Part)
    (<side> Side)
    (<side-pair>
     (and Side-Pair (gen-from-pred (sides-for-holding-device <side> <side-pair>)))))
   (and (~ (holding-weakly <machine> <holding-device> <part> <side> <side-pair>))
        (~ (holding <machine> <holding-device> <part> <side> <side-pair>)))))
 (effects () ((add (is-empty-holding-device <holding-device> <machine>)))))

;;;*****************************************************
;; Inference rules to generate legal orientations (only fire at the beginning of
;; planning, because they are eager and they involve static predicates)

(Inference-Rule SIDES-FOR-HD
 (mode eager)
 (params)
 (preconds () t) ;;args are <side-up> <side-pair>
 (effects () ((add (sides-for-holding-device SIDE1 SIDE2-SIDE5))
              (add (sides-for-holding-device SIDE1 SIDE3-SIDE6))
              (add (sides-for-holding-device SIDE2 SIDE1-SIDE4))
              (add (sides-for-holding-device SIDE2 SIDE3-SIDE6))
              (add (sides-for-holding-device SIDE3 SIDE2-SIDE5))
              (add (sides-for-holding-device SIDE3 SIDE1-SIDE4))
              (add (sides-for-holding-device SIDE4 SIDE2-SIDE5))
              (add (sides-for-holding-device SIDE4 SIDE3-SIDE6))
              (add (sides-for-holding-device SIDE5 SIDE1-SIDE4))
              (add (sides-for-holding-device SIDE5 SIDE3-SIDE6))
              (add (sides-for-holding-device SIDE6 SIDE1-SIDE4))
              (add (sides-for-holding-device SIDE6 SIDE2-SIDE5)))))

(Inference-Rule SIDE-UP
 (mode eager)
 (params)
 (preconds () t)
 (effects () ((add (side-up-for-machining LENGTH SIDE3))
              (add (side-up-for-machining LENGTH SIDE6))
              (add (side-up-for-machining WIDTH SIDE2))
              (add (side-up-for-machining WIDTH SIDE5))
              (add (side-up-for-machining HEIGHT SIDE1))
              (add (side-up-for-machining HEIGHT SIDE4))
              (add (side-up-for-machining DIAMETER SIDE1))
              (add (side-up-for-machining DIAMETER SIDE0)))))

;;; Side-mill is the only operator that does not machine the part that
;;; is up, but one of the sides. Once the part is held with some
;;; orientation, it is impossible to side-mill both sides as one is
;;; being covered by the holding device. This is avoided in the
;;; SIDE-MILL operator. Therefore to machine both sides the part has
;;; to be released and held again.
```

```
;;; Here we model the part sides and orientations.

(Inference-Rule SIDE-FOR-END-MILL
 (mode eager)
 (params)
 (preconds () t)   ;;args are <dim> <side-up> <machined-side>
 (effects () ((add (side-for-side-mill WIDTH SIDE1 SIDE2))
              (add (side-for-side-mill WIDTH SIDE1 SIDE5))
              (add (side-for-side-mill WIDTH SIDE3 SIDE2))
              (add (side-for-side-mill WIDTH SIDE3 SIDE5))
              (add (side-for-side-mill WIDTH SIDE4 SIDE2))
              (add (side-for-side-mill WIDTH SIDE4 SIDE5))
              (add (side-for-side-mill WIDTH SIDE6 SIDE2))
              (add (side-for-side-mill WIDTH SIDE6 SIDE5))

              (add (side-for-side-mill LENGTH SIDE1 SIDE3))
              (add (side-for-side-mill LENGTH SIDE1 SIDE6))
              (add (side-for-side-mill LENGTH SIDE2 SIDE3))
              (add (side-for-side-mill LENGTH SIDE2 SIDE6))
              (add (side-for-side-mill LENGTH SIDE4 SIDE3))
              (add (side-for-side-mill LENGTH SIDE4 SIDE6))
              (add (side-for-side-mill LENGTH SIDE5 SIDE3))
              (add (side-for-side-mill LENGTH SIDE5 SIDE6))

              (add (side-for-side-mill HEIGHT SIDE2 SIDE1))
              (add (side-for-side-mill HEIGHT SIDE2 SIDE4))
              (add (side-for-side-mill HEIGHT SIDE3 SIDE1))
              (add (side-for-side-mill HEIGHT SIDE3 SIDE4))
              (add (side-for-side-mill HEIGHT SIDE5 SIDE1))
              (add (side-for-side-mill HEIGHT SIDE5 SIDE4))
              (add (side-for-side-mill HEIGHT SIDE6 SIDE1))
              (add (side-for-side-mill HEIGHT SIDE6 SIDE4)))))

;;;*************************************************************
; inference rules for shape

(Inference-Rule IS-RECTANGULAR
 (params <part>)
 (preconds ((<part> Part)
            (<l> (and Size (gen-from-pred (size-of <part> LENGTH <l>))))
            (<w> (and Size (gen-from-pred (size-of <part> WIDTH <w>))))
            (<h> (and Size (gen-from-pred (size-of <part> HEIGHT <h>)))))
   t)
 (effects () ((add (shape-of <part> RECTANGULAR)))))

(Inference-Rule IS-CYLINDRICAL
 (params <part>)
 (preconds ((<part> Part)
            (<l> (and Size (gen-from-pred (size-of <part> LENGTH <l>))))
            (<d> (and Hole-Diameter
                      (gen-from-pred (size-of <part> DIAMETER <d>)))))
  t)
 (effects () ((add (shape-of <part> CYLINDRICAL)))))

(Inference-Rule ARE-SIDES-OF-RECTANGULAR-PART
 (params <part>)
 (preconds ((<part> Part))
  (shape-of <part> RECTANGULAR))
 (effects () ((add (side-of <part> SIDE1))
              (add (side-of <part> SIDE2))
              (add (side-of <part> SIDE3))
              (add (side-of <part> SIDE4))
              (add (side-of <part> SIDE5))
              (add (side-of <part> SIDE6)))))

(Inference-Rule ARE-SIDES-OF-CYLINDRICAL-PART
 (params <part>)
 (preconds ((<part> Part))
  (shape-of <part> CYLINDRICAL))
 (effects () ((add (side-of <part> SIDE0))
              (add (side-of <part> SIDE3))
              (add (side-of <part> SIDE6)))))

;;;*************************************************************
;;; Properties of part materials:
;;; ALLOYS:
;;; - non-ferrous
;;;    - brass
;;;    - copper
;;;    - bronze
;;; - ferrous
;;;    - steel
;;;    - iron
;;; HARDNESS:
;;; - soft
;;;    - aluminum
;;;    - non-ferrous
;;; - hard
;;;    - ferrous
;;; HIGH-MELTING-POINT:
;;; - tungsten
;;; - molybdenum

(Inference-Rule MATERIAL-FERROUS
 (params <part>)
 (preconds ((<part> Part))
  (or (material-of <part> STEEL)
      (material-of <part> IRON)))
 (effects () ((add (alloy-of <part> FERROUS)))))

(Inference-Rule MATERIAL-NON-FERROUS
 (params <part>)
 (preconds ((<part> Part))
  (or (material-of <part> BRASS)
      (material-of <part> COPPER)
      (material-of <part> BRONZE)))
 (effects () ((add (alloy-of <part> NON-FERROUS)))))

(Inference-Rule HARDNESS-OF-MATERIAL-SOFT
 (params <part>)
 (preconds ((<part> Part))
  (or (alloy-of <part> NON-FERROUS)
      (material-of <part> ALUMINUM)))
 (effects () ((add (hardness-of <part> SOFT)))))

(Inference-Rule HARDNESS-OF-MATERIAL-HARD
 (params <part>)
 (preconds ((<part> Part))
  (alloy-of <part> FERROUS))
 (effects () ((add (hardness-of <part> HARD)))))

(Inference-Rule HIGH-MELTING-POINT
 (params <wire>)
 (preconds ((<wire> WIRE))
  (or (material-of <wire> TUNGSTEN)
      (material-of <wire> MOLYBDENUM)))
 (effects () ((add (has-high-melting-point <wire>)))))
```

```
(Inference-Rule HAS-CENTER-HOLES1
 (params <part> <x2> <y2>)
 (preconds ((<part> PART)
            (<x> (and Size (gen-from-pred (size-of <part> WIDTH <x>))))
            (<y> (and Size (gen-from-pred (size-of <part> HEIGHT <y>))))
            (<x2> (and Hole-Location (half-of <x> <x2>)))
            (<y2> (and Hole-Location (half-of <y> <y2>)))
            (and (shape-of <part> RECTANGULAR)
                 (has-center-hole <part> CENTER-HOLE-SIDE3 SIDE3 <x2> <y2>)
                 (is-countersinked <part> CENTER-HOLE-SIDE3 SIDE3 1/8 1/16 <x2> <y2> 60)
                 (has-center-hole <part> CENTER-HOLE-SIDE6 SIDE6 <x2> <y2>)
                 (is-countersinked <part> CENTER-HOLE-SIDE6 SIDE6 1/8 1/16 <x2> <y2> 60)))
 (effects () ((add (has-center-holes <part>)))))

(Inference-Rule HAS-CENTER-HOLES2
 (params <part> <x2> <y2>)
 (preconds ((<part> PART)
            (<x> (and Size (gen-from-pred (size-of <part> DIAMETER <x>))))
            (<y> (and Size (same <y> <x>)))
            (<x2> (and Hole-Location (half-of <x> <x2>)))
            (<y2> (and Hole-Location (same <y2> <x2>)))
            (and (shape-of <part> CYLINDRICAL)
                 (has-center-hole <part> CENTER-HOLE-SIDE3 SIDE3 <x2> <y2>)
                 (is-countersinked <part> CENTER-HOLE-SIDE3 SIDE3 1/8 1/16 <x2> <y2> 60)
                 (has-center-hole <part> CENTER-HOLE-SIDE6 SIDE6 <x2> <y2>)
                 (is-countersinked <part> CENTER-HOLE-SIDE6 SIDE6 1/8 1/16 <x2> <y2> 60)))
 (effects () ((add (has-center-holes <part>)))))

;;; ************************************************************
;;Vises hold cylindrical parts weakly only. Then we need another
;;device (toe clamp) to hold them so the machining op can be done.
;;Therefore if the goal is holding we can't use a vise when the part
;;is cylindrical.

(control-rule AVOID-VISE-FOR-CYLINDRICAL-PARTS
 (if (and (current-goal-first-arg <part>)
          ;;check if the part is cylindrical
          (known (size-of <part> DIAMETER <d>))
          ;;all ops that may use vises
          (current-ops (DRILL-WITH-SPOT-DRILL DRILL-WITH-TWIST-DRILL
                        DRILL-WITH-HIGH-HELIX-DRILL
                        TAP COUNTERSINK COUNTERBORE REAM
                        SIDE-MILL FACE-MILL
                        DRILL-WITH-SPOT-DRILL-IN-MILLING-MACHINE
                        DRILL-WITH-TWIST-DRILL-IN-MILLING-MACHINE))
          (type-of-object-gen <vise> VISE)))
 (then reject bindings ((<holding-device> . <vise>)))) 

;;;*****************************************************
(control-rule PUT-ON-MACHINE-TABLE-IF-NOT-HOLDING
 (if (and (current-goal (on-table <machine> <part>))
          (- (type-of-object <machine> SHAPER)
             (false-in-state-forall-values
              (holding <machine> <holding-device> <part> <s> <sp>)
              (<holding-device> HOLDING-DEVICE)
              (<s> SIDE) (<sp> SIDE-PAIR)
              (false-in-state-forall-values
               (holding-weakly <machine> <holding-device> <part> <s> <sp>)
               (<holding-device> HOLDING-DEVICE)
               (<s> SIDE) (<sp> SIDE-PAIR)))))
 (then select operator PUT-ON-MACHINE-TABLE)
 (control-rule PUT-ON-MACHINE-TABLE-IF-HOLDING
  (if (and (current-goal (on-table <machine> <part>))
                                                    (or-metapred
                                                     (known
                                                      (holding
                                                       <machine> <holding-device> <part> <s> <sl> <s2>))
                                                     (known
                                                      (holding-weakly
                                                       <machine> <holding-device> <part> <s> <sl> <s2>))))
                                                    (then reject operator PUT-ON-MACHINE-TABLE))

                                                   (control-rule REMOVE-FROM-TABLE
                                                    (if (and (current-goal (- (on-table <machine> <part>)))
                                                             (not-candidate-goal (on-table <other-machine> <part>)))
                                                    (then select operator REMOVE-FROM-MACHINE-TABLE))

                                                   (control-rule DONT-MAKE-RECTANGULAR-TO-HOLD-WITH-VISE
                                                    (if (and (current-goal
                                                              (holding <machine> <holding-device> <part> <s> <sp>))
                                                             ;;this is to ask if the part is cylindrical
                                                             (known (size-of <part> DIAMETER <diameter>)))
                                                    (then reject operator HOLD-WITH-VISE))

                                                   ;;;*****************************************************
                                                   ;;; rules for choosing the fluids.

                                                   (control-rule USE-MINERAL-OIL
                                                    (if (and (current-goal-first-arg <part>)
                                                             (current-ops (DRILL-WITH-OIL-HOLE-DRILL DRILL-WITH-HIGH-HELIX-DRILL
                                                                           DRILL-WITH-GUN-DRILL REAM
                                                                           ROUGH-GRIND-WITH-HARD-WHEEL ROUGH-GRIND-WITH-SOFT-WHEEL
                                                                           FINISH-GRIND-WITH-HARD-WHEEL
                                                                           FINISH-GRIND-WITH-SOFT-WHEEL
                                                                           CUT-WITH-CIRCULAR-FRICTION-SAW)
                                                                       (or (known (material-of <part> STEEL))
                                                                           (known (material-of <part> ALUMINUM)))
                                                                       (type-of-object-gen <f> mineral-oil)))
                                                    (then reject bindings ((<fluid> . <f>)))) 

                                                   ;;; Rule for choosing operators to add the fluids

                                                   (control-rule ADD-OIL-ANY
                                                    (if (and (current-goal (has-fluid <mach> <fluid> <part>))
                                                             (known (material-of <part> <mat>))
                                                             (one-of-metapred <mat> (BRASS BRONZE COPPER)))
                                                    (then select operator ADD-ANY-CUTTING-FLUID))

                                                   (control-rule SIZE-BEFORE-SURFACE
                                                    (if (and (candidate-goal (size-of <part> <dim> <value>))
                                                             (known (side-up-for-machining <dim> <side>)))
                                                    (then prefer goal (size-of <part> <dim> <value>)
                                                          (surface-finish-side <part> <side> <sf>))
```

```
(control-rule reject-hole-goal
  (if (and (candidate-goal (has-hole <p> <h> <s> <d> <di> <x> <y>))
           (or-metapred
            (candidate-goal (is-tapped <p> <h> <s> <d> <di> <x> <y>))
            (candidate-goal (is-counterbored <p> <h> <s> <d> <di> <x> <y> <s>))
            (candidate-goal (is-countersinked <p> <h> <s> <d> <di> <x> <y> <a>))
            (candidate-goal (is-reamed <p> <h> <s> <d> <di> <x> <y>)))))
   (then reject goal (has-hole <p> <h> <s> <d> <di> <x> <y>)))
```

Appendix C

Learned Quality-Enhancing Control Rules

```
:problem 'tst-0
(control-rule prefer-drill-with-spot-drill-in-milling-machine7
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (known (holding-tool <machine> <drill-bit>))
          (type-of-object <drill-bit> spot-drill)
          (type-of-object <machine> milling-machine)))
 (then prefer operator drill-with-spot-drill-in-milling-machine drill-with-spot-drill)))

(control-rule prefer-bnds-drill-with-spot-drill-in-milling-machine8
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (current-operator drill-with-spot-drill-in-milling-machine)
          (known (holding-tool <machine-3> <drill-bit-4>))
          (or (diff <machine-3> <machine-1>) (diff <drill-bit-4> <drill-bit-2>))))
 (then prefer bindings ((<machine> . <machine-3>) (<drill-bit> . <drill-bit-4>))
                       ((<machine> . <machine-1>) (<drill-bit> . <drill-bit-2>))))

:problem 'tst-1
(control-rule prefer-drill-with-spot-drill-in-milling-machine9
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (known (on-table <machine> <part>))
          (type-of-object <machine> milling-machine)))
  (then prefer operator drill-with-spot-drill-in-milling-machine drill-with-spot-drill)))

(control-rule prefer-bnds-drill-with-spot-drill-in-milling-machine10
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (current-operator drill-with-spot-drill-in-milling-machine)
          (known (on-table <machine-2> <part>)) (diff <machine-2> <machine-1>)))
 (then prefer bindings ((<machine> . <machine-2>)) ((<machine> . <machine-1>))))

:problem 'tst-3
(control-rule prefer-drill-with-spot-drill-in-milling-machine11
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (known (has-device <machine> <holding-device>))
          (type-of-object <machine> milling-machine)))
 (then prefer operator drill-with-spot-drill-in-milling-machine drill-with-spot-drill)))

(control-rule prefer-bnds-drill-with-spot-drill-in-milling-machine12
```

```
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (current-operator drill-with-spot-drill-in-milling-machine)
          (known (has-device <machine-3> <holding-device-4>))
          (or (diff <machine-3> <machine-1>)
              (diff <holding-device-4> <holding-device-2>))))
 (then prefer bindings ((<machine> . <machine-3>) (<holding-device> . <holding-device-4>))
                       ((<machine> . <machine-1>) (<holding-device> . <holding-device-2>))))

:problem 'tst2-1
(control-rule prefer-drill-with-twist-drill-in-milling-machine13
 (if (and (current-goal
            (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>))
          (known (has-device <machine> <holding-device>))
          (type-of-object <machine> milling-machine)))
 (then prefer operator drill-with-twist-drill-in-milling-machine drill-with-twist-drill)))

(control-rule prefer-bnds-drill-with-twist-drill-in-milling-machine14
 (if (and (current-goal
            (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>))
          (current-operator drill-with-twist-drill-in-milling-machine)
          (known (has-device <machine-3> <holding-device-4>))
          (or (diff <machine-3> <machine-1>)
              (diff <holding-device-4> <holding-device-2>))))
 (then prefer bindings ((<machine> . <machine-3>) (<holding-device> . <holding-device-4>))
                       ((<machine> . <machine-1>) (<holding-device> . <holding-device-2>))))

:problem 'tst2-2
(control-rule prefer-bnds-drill-with-spot-drill-in-milling-machine15
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (current-operator drill-with-spot-drill-in-milling-machine)
          (pending-goal (holding <machine-4> <holding-device-5> <part> <side> <side-pair-6>))
          (or (diff <machine-4> <machine-1>)
              (diff <holding-device-5> <holding-device-2>)
              (diff <side-pair-6> <side-pair-3>))))
 (then prefer bindings ((<machine> . <machine-4>) (<holding-device> . <holding-device-5>)
                        (<side-pair> . <side-pair-6>))
                       ((<machine> . <machine-1>) (<holding-device> . <holding-device-2>)
                        (<side-pair> . <side-pair-3>)))))

:problem 'tst2-3
(control-rule prefer-bnds-drill-with-spot-drill16
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (current-operator drill-with-spot-drill)
          (pending-goal (holding <machine-4> <holding-device-5> <part> <side> <side-pair-6>))
          (or (diff <machine-4> <machine-1>)
              (diff <holding-device-5> <holding-device-2>)
              (diff <side-pair-6> <side-pair-3>))))
 (then prefer bindings ((<machine> . <machine-4>) (<holding-device> . <holding-device-5>)
                        (<side-pair> . <side-pair-6>))
                       ((<machine> . <machine-1>) (<holding-device> . <holding-device-2>)
                        (<side-pair> . <side-pair-3>)))))

:problem 'tst4-0
(control-rule prefer-side-mill17
 (if (and (current-goal (size-of <part> <dim> <value>))
          (or-metapred
            (known (holding <machine> <holding-device> <part> <side> <side-pair>))
            (pending-goal (holding <machine> <holding-device> <part> <side> <side-pair>)))
          (known (sides-for-holding-device <side> <side-pair>))
          (known (side-for-side-mill <dim> <side> <mach-side>))
```

```
          (not-in-side-pair <mach-side> <side-pair>)
          (type-of-object <machine> milling-machine)))
 (then prefer operator side-mill face-mill))

(control-rule prefer-bnds-side-mill18
 (if (and (current-goal (size-of <part> <dim> <value>))
          (current-operator side-mill)
          (or-metapred
            (known (holding <machine-5> <holding-device-6> <part> <side-7> <side-pair-8>))
            (pending-goal (holding <machine-5> <holding-device-6> <part> <side-7> <side-pair-8>)))
          (or (diff <machine-5> <machine-1>)
              (diff <holding-device-6> <holding-device-2>) (diff <side-7> <side-3>)
              (diff <side-pair-8> <side-pair-4>))))
 (then prefer bindings
    ((<machine> . <machine-5>) (<holding-device> . <holding-device-6>) (<side> . <side-7>)
     (<side-pair> . <side-pair-8>))
    ((<machine> . <machine-1>) (<holding-device> . <holding-device-2>) (<side> . <side-3>)
     (<side-pair> . <side-pair-4>))))

(control-rule prefer-bnds-face-mill19
 (if (and (current-goal (size-of <part> <dim> <value>))
          (current-operator face-mill)
          (pending-goal (size-of <part> <dim-1> <value-2>))
          (known (sides-for-holding-device <side-7> <side-pair-8>))
          (known (side-for-side-mill <dim-1> <side-7> <mach-side>))
          (not-in-side-pair <mach-side> <side-pair-8>)
          (forall-metapred (known (side-for-side-mill <dim-1> <side-3> <mach-side-9>))
                           (or (~ (known (sides-for-holding-device <side-3> <side-pair-4>)))
                               (~ (not-in-side-pair <mach-side-9> <side-pair-4>))))
          (or (diff <side-7> <side-3>) (diff <side-pair-8> <side-pair-4>))))
 (then prefer bindings
    ((<machine> . <machine-5>) (<holding-device> . <holding-device-6>) (<side> . <side-7>)
     (<side-pair> . <side-pair-8>))
    ((<machine> . <machine-1>) (<holding-device> . <holding-device-2>) (<side> . <side-3>)
     (<side-pair> . <side-pair-4>)))))

:problem 'tst5-2
(control-rule prefer-drill-with-spot-drill25
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (known (has-device <machine> <holding-device>))
          (type-of-object <machine> drill)))
 (then prefer operator drill-with-spot-drill drill-with-spot-drill-in-milling-machine)))

(control-rule prefer-bnds-drill-with-spot-drill26
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (current-operator drill-with-spot-drill)
          (known (has-device <machine-3> <holding-device-4>))
          (or (diff <machine-3> <machine-1>)
              (diff <holding-device-4> <holding-device-2>))))
 (then prefer bindings ((<machine> . <machine-3>) (<holding-device> . <holding-device-4>))
                       ((<machine> . <machine-1>) (<holding-device> . <holding-device-2>))))

(control-rule prefer-drill-with-spot-drill27
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (pending-goal (holding <machine> <holding-device> <part> <side> <side-pair>))
          (known (sides-for-holding-device <side> <side-pair>))
          (type-of-object <machine> drill)))
 (then prefer operator drill-with-spot-drill drill-with-spot-drill-in-milling-machine))

(control-rule prefer-bnds-drill-with-spot-drill28
```

```
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (current-operator drill-with-spot-drill)
          (pending-goal (holding <machine-4> <holding-device-5> <part> <side> <side-pair-6>))
          (or (diff <machine-4> <machine-1>)
              (diff <holding-device-5> <holding-device-2>)
              (diff <side-pair-6> <side-pair-3>))))
 (then prefer bindings ((<machine> . <machine-4>) (<holding-device> . <holding-device-5>)
                        (<side-pair> . <side-pair-6>))
                       ((<machine> . <machine-1>) (<holding-device> . <holding-device-2>)
                        (<side-pair> . <side-pair-3>))))

:problem 'tst5-3
(control-rule prefer-goal-30
 (if (and (candidate-goal (holding <drill05> <vise06> <part> <side17> <side2-side58>))
          (known (holding <machine-1> <holding-device-2> <part> <side-3> <side-pair-4>))
          (is-subgoal-of-ops
              (holding <machine-1> <holding-device-2> <part> <side-3> <side-pair-4>) <ops>)
          (first-pending-subgoal-in-subtree <pref-goal> <ops>)
          (diff <pref-goal> <other-goal>)
          (~ (is-pending-subgoal-in-subtree <other-goal> <ops>))))
 (then prefer goal <pref-goal> <other-goal>)))

:problem 'tst5-9
(control-rule prefer-drill-with-twist-drill-in-milling-machine31
 (if (and (current-goal
            (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>))
          (known (on-table <machine> <part>))
          (type-of-object <machine> milling-machine)))
 (then prefer operator drill-with-twist-drill-in-milling-machine drill-with-twist-drill)))

(control-rule prefer-bnds-drill-with-twist-drill-in-milling-machine32
 (if (and (current-goal
            (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>))
          (current-operator drill-with-twist-drill-in-milling-machine)
          (known (on-table <machine-2> <part>))
          (diff <machine-2> <machine-1>)))
 (then prefer bindings ((<machine> . <machine-2>)) ((<machine> . <machine-1>)))))

:problem 'tst6-1
(control-rule prefer-drill-with-twist-drill-in-milling-machine33
 (if (and (current-goal
            (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>))
          (pending-goal (size-of <part-1> <dim-2> <value-3>))))
 (then prefer operator drill-with-twist-drill-in-milling-machine drill-with-twist-drill)))

:problem 'tst6-2
(control-rule prefer-goal-34
 (if (and (candidate-goal (is-available-tool-holder <machine>))
          (known (holding-tool <machine> <milling-cutter>))
          (is-subgoal-of-ops (holding-tool <machine> <milling-cutter>) <ops>)
          (first-pending-subgoal-in-subtree <pref-goal> <ops>)
          (diff <pref-goal> <other-goal>)
          (~ (is-pending-subgoal-in-subtree <other-goal> <ops>))))
 (then prefer goal <pref-goal> <other-goal>)))

:problem 'tst6-3
(control-rule prefer-side-mill35
 (if (and (current-goal (size-of <part> <dim> <value>))
          (pending-goal
           (has-hole <part> <hole-1> <side> <hole-depth-2> <hole-diameter-3> <loc-x-4> <loc-y-5>))
```

```
          (known (sides-for-holding-device <side> <side-pair>))
          (known (side-for-side-mill <dim> <side> <mach-side>))
          (not-in-side-pair <mach-side> <side-pair>)))
 (then prefer operator side-mill face-mill)))

(control-rule prefer-bnds-side-mill36
 (if (and (current-goal (size-of <part> <dim> <value>))
          (current-operator side-mill)
          (pending-goal
            (has-hole <part> <hole-1> <side-2> <hole-depth-2> <hole-diameter-3> <loc-x-4> <loc-y-5>))
          (diff <side-2> <side-1>)))
 (then prefer bindings ((<side> . <side-2>)) ((<side> . <side-1>)))))

:problem 'tst6-4
(control-rule prefer-bnds-drill-with-twist-drill-in-milling-machine37
 (if (and (current-goal
            (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>))
          (current-operator drill-with-twist-drill-in-milling-machine)
          (pending-goal
            (holding <machine-4> <holding-device-5> <part> <side> <side-pair-6>))
          (or (diff <machine-4> <machine-1>)
              (diff <holding-device-5> <holding-device-2>)
              (diff <side-pair-6> <side-pair-3>))))
 (then prefer bindings ((<machine> . <machine-4>) (<holding-device> . <holding-device-5>)
                        (<side-pair> . <side-pair-6>))
                       ((<machine> . <machine-1>) (<holding-device> . <holding-device-2>)
                        (<side-pair> . <side-pair-3>)))))

:problem 'tst6-5
(control-rule prefer-drill-with-spot-drill-in-milling-machine38
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (pending-goal (holding <machine> <holding-device> <part> <side> <side-pair>))
          (type-of-object <machine> milling-machine)))
 (then prefer operator drill-with-spot-drill-in-milling-machine drill-with-spot-drill)))

(control-rule prefer-bnds-drill-with-spot-drill-in-milling-machine39
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (current-operator drill-with-spot-drill-in-milling-machine)
          (pending-goal
            (holding <machine-4> <holding-device-5> <part> <side> <side-pair-6>))
          (or (diff <machine-4> <machine-1>)
              (diff <holding-device-5> <holding-device-2>)
              (diff <side-pair-6> <side-pair-3>))))
 (then prefer bindings ((<machine> . <machine-4>) (<holding-device> . <holding-device-5>)
                        (<side-pair> . <side-pair-6>))
                       ((<machine> . <machine-1>) (<holding-device> . <holding-device-2>)
                        (<side-pair> . <side-pair-3>)))))

*rule-relative-prefs* =
((prefer-drill-with-spot-drill-in-milling-machine38 prefer-drill-with-spot-drill25))

:problem 'tst6-6
(control-rule prefer-bnds-face-mill40
 (if (and (current-goal (size-of <part> <dim> <value>))
          (current-operator face-mill)
          (pending-goal (holding <machine-5> <holding-device-6> <part> <side-7> <side-pair-8>))
          (or (diff <machine-5> <machine-1>)
              (diff <holding-device-6> <holding-device-2>) (diff <side-7> <side-3>)
              (diff <side-pair-8> <side-pair-4>))))
 (then prefer bindings
```

```
    ((<machine> . <machine-5>) (<holding-device> . <holding-device-6>) (<side> . <side-7>)
     (<side-pair> . <side-pair-8>))
    ((<machine> . <machine-1>) (<holding-device> . <holding-device-2>) (<side> . <side-3>)
     (<side-pair> . <side-pair-4>)))))

:problem 'tst7-2
(control-rule prefer-goal-42
 (if (and (candidate-goal (holding <machine-2> <holding-device> <part06> <side27> <side1-side48>))
          (known (has-device <machine-1> <holding-device>))
          (pending-goal (holding <machine-1> <holding-device> <part03> <side24> <side3-side65>))
          (is-subgoal-of-ops
             (holding <machine-1> <holding-device> <part03> <side24> <side3-side65>) <ops>)
          (diff <machine-2> <machine-1>)
          (first-pending-subgoal-in-subtree <pref-goal> <ops>)
          (diff <pref-goal> <other-goal>)
          (~ (is-pending-subgoal-in-subtree <other-goal> <ops>))))
 (then prefer goal <pref-goal> <other-goal>)))

(control-rule prefer-goal-43
 (if (and (candidate-goal (holding <machine-2> <vise06> <part> <side27> <side1-side48>))
          (known (on-table <machine-1> <part>))
          (pending-goal (holding <machine-1> <vise03> <part> <side14> <side3-side65>))
          (is-subgoal-of-ops (holding <machine-1> <vise03> <part> <side14> <side3-side65>)
                             <ops>)
          (diff <machine-2> <machine-1>)
          (first-pending-subgoal-in-subtree <pref-goal> <ops>)
          (diff <pref-goal> <other-goal>)
          (~ (is-pending-subgoal-in-subtree <other-goal> <ops>))))
 (then prefer goal <pref-goal> <other-goal>))

:problem 'tst7-4
(control-rule prefer-drill-with-twist-drill47
 (if (and (current-goal
            (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>))
          (known (holding <machine> <holding-device> <part> <side> <side-pair>))
          (type-of-object <machine> drill)))
 (then prefer operator drill-with-twist-drill drill-with-twist-drill-in-milling-machine)))

(control-rule prefer-bnds-drill-with-twist-drill48
 (if (and (current-goal
            (has-hole <part> <hole> <side> <hole-depth> <hole-diameter> <loc-x> <loc-y>))
          (current-operator drill-with-twist-drill)
          (known (holding <machine-4> <holding-device-5> <part> <side> <side-pair-6>))
          (or (diff <machine-4> <machine-1>)
              (diff <holding-device-5> <holding-device-2>)
              (diff <side-pair-6> <side-pair-3>))))
 (then prefer bindings ((<machine> . <machine-4>) (<holding-device> . <holding-device-5>)
                        (<side-pair> . <side-pair-6>))
                       ((<machine> . <machine-1>) (<holding-device> . <holding-device-2>)
                        (<side-pair> . <side-pair-3>)))))

(control-rule prefer-drill-with-spot-drill49
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (known (holding <machine> <holding-device> <part> <side> <side-pair>))
          (type-of-object <machine> drill)))
 (then prefer operator drill-with-spot-drill drill-with-spot-drill-in-milling-machine))

(control-rule prefer-bnds-drill-with-spot-drill50
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (current-operator drill-with-spot-drill)
```

```
        (known (holding <machine-4> <holding-device-5> <part> <side> <side-pair-6>))
        (or (diff <machine-4> <machine-1>)
            (diff <holding-device-5> <holding-device-2>)
            (diff <side-pair-6> <side-pair-3>))))
(then prefer bindings ((<machine> . <machine-4>) (<holding-device> . <holding-device-5>)
                       (<side-pair> . <side-pair-6>))
                      ((<machine> . <machine-1>) (<holding-device> . <holding-device-2>)
                       (<side-pair> . <side-pair-3>))))
```

Appendix D

Detailed Experimental Results

The experiments described in Chapters 3 and 4 were run in Allegro Common Lisp 4.1 on a Sun Sparcstation ELC under the Mach/Unix operating system. The tables below show the data for the complete set of problems in the test phase using the quality metric of Table 3.1. For each problem results are shown of solving it (a) without any quality-enhancing control knowledge, (b) with the control rules learned from the training set (see Section 3.13.2), and (c) with the control knowledge trees learned from the same training set (see Section 4.8.3). The meaning of the columns is the following:

- **Prob num**: the problem number; the prefix indicates the problem set to which it belongs.
- **Quality**: the quality value of the solution found.
- **Time**: the CPU time, in seconds, that the PRODIGY4.0 took to solve the problem.
- **Nodes**: the number of nodes searched.
- **Plan length**: the length of the solution found, i.e. the number of steps of the plan, *including the inference rules* fired.

Prob	No quality control knowledge				Learned control rules				Learned cktrees			
Num	Qual	Time	Nodes	Length	Qual	Time	Nodes	Length	Qual	Time	Nodes	Length
1-0	26	3	47	38	26	3	47	38	26	4	47	38
1-1	41	4	63	48	24	3	42	46	24	4	42	46
1-2	24	3	42	53	24	3	42	53	24	4	42	53
1-3	40	4	59	62	25	3	46	36	25	4	46	36
1-4	24	3	39	44	24	2	39	44	24	3	39	44
1-5	24	2	39	35	24	2	39	35	24	3	39	35

Prob	No quality control knowledge				Learned control rules				Learned cktrees			
Num	Qual	Time	Nodes	Length	Qual	Time	Nodes	Length	Qual	Time	Nodes	Length
1-6	24	2	39	35	24	3	39	35	24	3	39	35
1-7	25	3	46	36	25	3	46	36	25	4	46	36
1-8	24	2	39	44	24	2	39	44	24	3	39	44
1-9	41	4	63	60	15	2	25	24	15	2	25	24
1-10	26	3	47	38	26	3	47	38	26	4	47	38
1-11	24	3	42	46	24	3	42	46	24	4	42	46
1-12	49	4	67	61	23	2	29	25	23	3	29	25
1-13	23	2	32	36	23	2	32	36	23	3	32	36
1-14	23	2	35	42	23	2	35	42	23	3	35	42
1-15	24	3	45	40	24	3	45	40	24	3	45	40
1-16	23	2	29	31	23	2	29	31	23	3	29	31
1-17	1	0	8	5	1	0	8	5	1	1	8	5
1-18	33	3	56	53	33	3	56	53	33	4	56	53
1-19	50	4	71	62	23	2	29	31	23	3	29	31
1-20	23	2	32	32	23	2	32	32	23	3	32	32
1-21	32	3	52	52	32	3	52	52	32	4	52	52
1-22	16	2	35	34	16	2	35	34	16	3	35	34
1-23	33	3	56	51	24	3	42	53	24	4	42	53
1-24	18	2	43	37	18	2	43	37	18	3	43	37
1-25	24	2	39	44	24	3	39	44	24	3	39	44
1-26	23	2	35	42	23	2	35	42	23	3	35	42
1-27	48	4	63	74	25	2	43	36	25	3	43	36
1-28	50	4	71	62	23	2	29	31	23	3	29	31
1-29	50	4	71	62	23	2	29	31	23	3	29	31
2-0	47	3	53	61	26	3	47	38	26	4	47	38
2-1	56	6	86	93	28	4	67	72	28	6	67	72
2-2	56	5	83	68	28	4	64	51	28	6	64	51
2-3	55	5	73	70	27	3	54	53	27	5	54	53
2-4	72	7	100	103	29	4	68	64	29	6	68	64
2-5	62	5	79	73	28	4	67	57	28	6	67	57
2-6	56	6	86	93	28	4	67	72	28	6	67	72
2-7	56	5	80	85	28	4	61	64	28	6	61	64
2-8	31	3	42	34	31	3	42	34	31	5	42	34
2-9	55	4	70	65	27	3	51	48	27	5	51	48
2-10	49	4	67	75	25	3	43	45	25	4	43	45
2-11	42	4	67	61	15	2	25	30	15	3	25	30
2-12	56	5	80	85	28	4	61	64	28	6	61	64
2-13	62	5	79	73	28	4	67	57	28	6	67	57
2-14	56	5	83	87	28	4	64	66	28	6	64	66
2-15	80	7	104	92	28	4	61	61	52	7	85	75
2-16	54	5	69	60	26	3	50	43	26	5	50	43
2-17	64	6	93	81	28	4	67	69	28	6	67	69
2-18	26	3	47	38	26	3	47	38	26	5	47	38
2-19	25	3	49	41	25	3	49	41	25	5	49	41
2-20	72	7	100	114	21	4	61	61	21	6	61	61

Prob	No quality control knowledge				Learned control rules				Learned cktrees			
Num	Qual	Time	Nodes	Length	Qual	Time	Nodes	Length	Qual	Time	Nodes	Length
2-21	33	3	56	49	24	3	45	52	24	4	45	52
2-22	1	0	8	4	1	0	8	4	1	2	8	4
2-23	81	7	108	94	26	3	47	50	26	5	47	50
2-24	4	1	26	24	4	1	26	24	4	3	26	24
2-25	56	6	83	94	36	5	74	83	28	6	64	73
2-26	41	4	63	63	25	3	46	47	25	5	46	47
2-27	73	7	104	82	27	3	54	53	27	5	54	53
2-28	81	7	108	109	28	4	61	64	28	6	61	64
2-29	26	3	50	38	26	3	50	38	26	5	50	38
3-0	60	5	65	71	32	4	46	50	32	6	46	50
3-1	24	3	45	51	24	3	45	51	24	4	45	52
3-2	25	3	49	53	25	3	49	53	25	4	49	53
3-3	106	9	109	125	78	15	90	104	78	18	90	104
3-4	90	8	98	113	62	14	79	92	62	28	79	92
3-5	53	5	65	71	53	6	66	71	25	9	46	50
3-6	76	6	76	87	76	7	77	87	76	10	76	87
3-7	16	2	35	42	16	3	35	42	16	4	35	43
3-8	76	6	76	87	76	7	77	87	76	10	76	87
3-9	32	3	52	61	32	4	52	61	32	5	52	62
3-10	78	7	90	101	50	6	71	80	50	10	71	80
3-11	91	8	102	115	63	16	83	94	63	30	83	94
3-12	41	4	63	63	41	4	63	63	41	5	63	63
3-13	82	7	88	97	54	14	69	76	54	25	69	76
3-14	91	8	102	105	63	15	83	84	63	29	83	84
3-15	89	7	88	97	89	20	89	97	61	22	69	76
3-16	48	4	63	73	48	5	63	73	48	5	63	73
3-17	32	4	52	51	32	4	52	51	32	5	52	52
3-18	52	4	55	66	24	4	36	45	24	6	36	45
3-19	32	3	51	55	4	2	29	29	4	5	29	29
3-20	62	5	79	79	34	5	60	58	34	9	60	58
3-21	41	4	63	74	41	4	63	74	41	6	63	74
3-22	60	5	65	71	60	6	66	71	32	9	46	50
3-23	33	3	56	53	33	4	56	53	33	4	56	53
3-24	24	3	42	45	24	3	42	45	24	4	42	46
3-25	70	6	86	89	70	8	87	89	42	11	67	68
3-26	62	5	79	79	34	5	60	58	34	9	60	58
3-27	54	5	72	79	26	4	53	58	26	6	53	58
3-28	54	5	66	71	32	4	51	55	32	6	51	55
3-29	62	5	79	90	34	5	60	69	34	9	60	69
4-0	79	8	113	124	51	7	94	103	51	9	94	103
4-1	142	14	178	172	114	14	159	155	114	20	159	155
4-2	93	9	116	125	65	8	97	104	65	17	97	104
4-3	151	15	181	184	97	13	157	145	97	19	157	145
4-4	91	9	128	117	63	8	109	100	63	10	109	100
4-5	121	12	158	136	93	11	139	119	138	17	166	189

Prob	No quality control knowledge				Learned control rules				Learned cktrees			
Num	Qual	Time	Nodes	Length	Qual	Time	Nodes	Length	Qual	Time	Nodes	Length
4-6	87	8	117	100	59	9	98	83	59	10	98	83
4-7	62	5	79	90	80	8	107	124	62	8	79	90
4-8	138	15	182	185	109	14	159	178	110	18	163	168
4-9	173	19	222	191	117	18	184	157	117	23	190	163
4-10	127	13	171	151	99	12	152	134	99	16	152	134
4-11	104	10	141	125	73	8	104	119	73	10	104	119
4-12	107	11	145	129	76	8	108	124	76	10	108	124
4-13	91	10	142	124	92	11	136	146	61	18	109	104
4-14	25	3	43	45	3	2	22	19	3	6	22	19
4-15	54	5	69	73	26	4	50	52	26	55	50	52
4-16	131	10	134	152	87	9	112	116	87	18	112	125
4-17	95	8	117	123	71	8	96	113	71	12	96	112
4-18	31	3	42	52	31	3	42	52	31	8	42	52
4-19	87	8	117	100	59	7	98	83	59	9	98	83
4-20	87	8	117	125	59	7	98	104	59	12	98	104
4-21	34	5	46	42	34	3	46	42	34	3	46	42
4-22	117	11	140	159	83	10	128	142	83	13	128	142
4-23	65	6	83	88	65	8	83	88	65	6	83	88
4-24	41	4	63	74	41	4	63	74	41	9	63	74
4-25	81	7	108	94	78	7	90	101	81	13	108	94
4-26	96	10	134	117	81	9	111	126	68	11	115	100
4-27	73	7	104	82	26	3	50	55	26	5	50	55
4-28	150	16	191	203	92	12	139	150	92	19	139	150
4-29	32	3	52	62	32	4	52	62	32	6	52	62
5-0	80	10	110	123	36	6	74	82	64	17	93	103
5-1	115	10	126	141	35	5	64	69	35	94	64	69
5-2	92	7	93	103	27	4	54	53	55	16	73	70
5-3	115	10	126	148	58	8	94	79	58	181	94	80
5-4	107	9	116	132	79	9	97	111	46	71	89	70
5-5	132	11	144	150	56	7	80	89	56	203	80	88
5-6	132	11	144	150	56	8	81	88	84	80	99	109
5-7	107	9	113	127	79	16	94	106	79	585	94	106
5-8	88	9	116	127	86	10	102	116	60	380	97	106
5-9	115	10	126	141	35	6	64	69	35	35	64	69
5-10	123	10	130	143	30	6	72	58	30	183	72	58
5-11	124	11	140	152	100	11	120	127	100	167	119	127
5-12	131	11	134	152	55	7	71	76	55	109	70	76
5-13	164	16	185	210	87	11	118	119	31	167	76	81
5-14	104	9	125	127	4	2	29	29	4	8	29	29
5-15	156	13	159	179	80	10	105	118	80	143	104	110
5-16	86	8	108	113	5	3	33	27	5	80	33	27
5-17	88	8	117	123	36	5	74	72	36	9	74	72
5-18	80	7	104	108	50	5	71	80	80	18	104	108
5-19	138	15	153	165	49	8	107	85	49	223	107	85
5-20	110	10	137	153	59	8	99	100	59	13	98	100

Prob	No quality control knowledge				Learned control rules				Learned cktrees			
Num	Qual	Time	Nodes	Length	Qual	Time	Nodes	Length	Qual	Time	Nodes	Length
5-21	154	14	167	191	50	8	85	94	50	76	84	94
5-22	85	8	104	111	34	6	65	58	34	72	65	57
5-23	162	15	171	185	11	5	69	56	11	42	69	56
5-24	139	14	163	163	47	8	99	104	75	221	118	125
5-25	79	7	100	105	51	6	81	93	79	18	100	105
5-26	79	7	103	112	35	5	70	82	63	16	89	102
5-27	140	14	164	174	56	9	107	118	84	381	126	139
5-28	111	11	141	155	30	5	69	67	30	9	69	67
5-29	85	9	96	104	56	7	84	76	56	13	83	76
6-0	88	9	124	102	88	10	124	102	88	12	124	102
6-1	64	6	93	91	64	7	93	91	64	7	93	91
6-2	56	5	80	85	56	6	80	85	56	6	80	85
6-3	138	15	182	150	115	14	152	170	115	15	152	170
6-4	149	16	188	201	121	18	169	181	121	23	172	186
6-5	83	8	108	120	83	9	108	120	83	9	108	120
6-6	174	42	342	250	174	44	346	250	174	56	364	262
6-7	125	15	180	166	125	17	180	170	172	24	221	244
6-8	129	14	171	149	114	14	151	172	114	15	151	172
6-9	142	15	178	195	114	14	159	174	114	16	159	174
6-10	152	18	198	212	124	16	179	191	124	20	179	191
6-11	114	20	211	142	109	11	133	152	109	13	133	152
6-12	147	26	259	172	99	13	152	121	131	20	203	178
6-13	87	9	123	133	87	12	123	133	117	16	140	162
6-14	146	16	186	177	117	15	163	179	117	16	163	179
6-15	87	9	123	133	87	10	123	133	87	11	123	133
6-16	166	29	279	200	115	13	149	162	148	19	200	184
6-17	153	28	258	212	113	13	158	160	137	20	198	210
6-18	129	14	171	178	101	13	152	157	101	15	152	157
6-19	68	7	106	91	68	8	106	91	68	12	106	91
6-20	115	13	163	141	84	10	126	132	84	115	126	132
6-21	116	11	140	151	88	10	121	130	88	15	121	130
6-22	114	12	145	160	92	11	130	144	92	12	130	144
6-23	223	37	333	240	148	21	214	234	149	28	218	190
6-24	119	12	158	134	91	11	139	117	91	14	139	117
6-25	112	11	145	126	109	11	127	141	109	12	127	141
6-26	219	27	275	238	176	28	236	262	163	32	240	207
6-27	98	10	134	138	98	11	134	138	98	13	134	138
6-28	194	39	322	251	145	20	196	203	178	37	251	250
6-29	127	16	171	149	140	16	167	183	140	30	167	183