RL-TR-96-225
Final Technical Report
February 1997

ALPS: AN ADAPTIVE LEARNING AND PLANNING SYSTEM

Odyssey Research Associates, Inc.

Randall J. Calistri-Yeh and Alberto M. Segre

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-96-225 has been reviewed and is approved for publication.

APPROVED: Antonette S. Melvin

ANTONETTE S. MELVIN
Project Engineer

FOR THE COMMANDER:

JOHN A. GRANIERO
Chief Scientist
Command, Control, & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/C3CA, 525 Brooks Rd, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1997	FINAL Feb 93 - Feb 96

4. TITLE AND SUBTITLE
ALPS: AN ADAPTIVE LEARNING AND PLANNING SYSTEM

5. FUNDING NUMBERS
C - F30602-93-C-0018
PE - 63728F
PR - 2532
TA - 01
WU - 44

6. AUTHOR(S)
Randall J. Calistri-Yeh, Alberto M. Segre

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Prime
ODYSSEY RESEARCH ASSOCIATES, Inc.
301 Dates Drive
Ithaca, NY 14850-1326

Subcontractor
DEPT OF MGT SCIENCE
The Univ. of Iowa
Iowa City, IA 52242-1000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory/C3CA
525 Brooks Rd.
Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-96-225

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Antonette S. Melvin/C3CA/(315) 330-4031

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
The goal of the ALPS project is to design and prototype a next generation, distributed, real-time, Al planning architecture. The central theme of our research is the real-world planning sytems must necessarily be adaptive; that is, they must reconcile themselves to limited environments, and real-time response. We address these concerns with a combination of methods from deliberative planning, machine learning, distributed computing, probabilistic reasoning, plan repair, and reactive planning.

14. SUBJECT TERMS
learning; planning and scheduling; explanation base learning (EBL); transportation scheduling; inserence engine; TPSDD

15. NUMBER OF PAGES
152

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Contents

1 Introduction **1**
- 1.1 ALPS and Transportation Scheduling 2
- 1.2 An Overview of the ALPS Architecture 3

2 The ALPS Lisp Inference Engine **11**
- 2.1 Introduction 11
- 2.2 Proofs and Inference 12
- 2.3 The ALPS Lisp Adaptive Inference Engine 14

3 Success and Failure Caching **17**
- 3.1 Introduction 17
- 3.2 Bounded-Overhead Caching 18
- 3.3 Evaluating Bounded-Overhead Caching 19
- 3.4 Summary of Bounded-Overhead Caching 35

4 Explanation-Based Learning **37**
- 4.1 Introduction 37
- 4.2 A Reconstruction of Traditional EBL 38
- 4.3 The Utility Problem 41
- 4.4 Learning from Determinations 43
- 4.5 Two Additional Explanation Transformation Operators 45
- 4.6 A Domain-Independent EBL* Algorithm 45
- 4.7 Evaluating EBL*DI 51
- 4.8 Summary of EBL*DI 58

5 Nagging and the DALI Inference Engine **60**
- 5.1 Introduction 60
- 5.2 Nagging 62
- 5.3 Nagging in First-Order Inference 65
- 5.4 Refinements to Nagging 70
- 5.5 Refinements to the Search Procedure 74
- 5.6 Empirical Evaluation 76
- 5.7 Discussion 79
- 5.8 Summary 81

6 Iterative Strengthening and Anytime Optimization **82**
6.1 Introduction . . . 82
6.2 The Concept of Iterative Strengthening . . . 83
6.3 Flexibility of Optimality Criteria . . . 84
6.4 Node Expansion Requirements . . . 85
6.5 Admissibility Requirements . . . 86
6.6 Decidability Requirements . . . 87
6.7 Discussion . . . 88
6.8 Summary . . . 89

7 Combining Multiple Speedup Techniques **90**
7.1 Introduction . . . 90
7.2 Combining Caching and EBL . . . 90
7.3 Combining Caching and Nagging . . . 95
7.4 Combining Caching and Iterative Strengthening . . . 96

8 The ALPS Fast Scheduler and the Transportation Domain **99**
8.1 Evaluating the ALPS Fast Scheduler . . . 100
8.2 Graphical User Interface . . . 101

9 Iterative Plan Repair **104**
9.1 Introduction . . . 104
9.2 Related Work . . . 105
9.3 General Plan Repair . . . 106
9.4 An Example . . . 113
9.5 Plan Repair in the Transportation Domain . . . 116
9.6 Discussion and Conclusions . . . 120

10 The ALPS TPFDD Simulator **122**
10.1 Simulator Design . . . 123
10.2 Process Queue Details . . . 125
10.3 Simulation of Bottlenecks . . . 125
10.4 Simulation of External Events . . . 126

11 Conclusions **127**
11.1 Speedup Techniques . . . 127
11.2 Transportation Planning and the Fast Scheduler . . . 130

A Acronyms **131**

B Bibliography **132**

List of Figures

1.1 The ALPS architecture. 4

2.1 Sample proof. 15

3.1 Performance of a non-caching iterative-deepening theorem prover 21
3.2 Performance of an unlimited-size caching iterative-deepening theorem prover 21
3.3 Search performance of an unlimited-size caching iterative-deepening theorem prover 23
3.4 Performance of four bounded-overhead caching schemes as a function of cache size. . 24
3.5 Search performance of a bounded-overhead caching iterative-deepening theorem prover using four cache management strategies. 25
3.6 Performance of CLRU and DLRU compared with LRU as a function of cache size. . 27
3.7 Search performance of CLRU and DLRU compared with LRU as a function of cache size. 28
3.8 Performance of success-only and failure-only caching systems using an LRU replacement policy as a function of cache size. 29
3.9 Search performance of success-only and failure-only caching systems using an LRU replacement policy as a function of cache size. 30
3.10 Performance of success-only and failure-only caching systems using a DLRU replacement policy as a function of cache size. 31
3.11 Performance of an assortment of dual-cache implementation trials compared to success-only, failure-only, and mixed-mode caching systems using a DLRU replacement policy as a function of cache size. 32
3.12 Search performance of a selection of dual-cache systems compared to success-only, failure-only, and mixed-mode caching systems using a DLRU replacement policy as a function of cache size. 32
3.13 Performance of DLRU caching both with and without redundant entries allowed as a function of cache size. 34
3.14 Search performance of DLRU caching both with and without redundant entries allowed as a function of cache size. 35

4.1 Sample proof of Figure 2.1 after applying procedure *ebl*. 42
4.2 Determining the sales tax rate at Cartier in New York. 44
4.3 Average CPU time ratios for selected training sets with $k \leq 8$. 52
4.4 Average node exploration ratios for selected training sets with $k \leq 8$. 52
4.5 Proof of $thm(or(P, not(P)))$. 54
4.6 Average percentage of unsolved test problems for nested training sets with $k \leq 85$. . 57
4.7 CPU time ratios for nested training sets with $k \leq 85$. 58
4.8 Node exploration ratios for nested training sets with $k \leq 85$. 59

5.1 Performance comparison of two closely related search procedures. 61
5.2 Search pruning via nagging. 63
5.3 Example of transformation under $\mathcal{P}$. 68
5.4 Example of transformation under $\mathcal{A}$. 69
5.5 Recursive nagging model. 71
5.6 The nagger's discovery of a subproof for $r(c)$ may permit the master to avoid proving it a second time. 72
5.7 Potential for cooperative search pruning. 74
5.8 Interference of nagging with intelligent backtracking. 76
5.9 Comparison of a 16-nagger parallel system with an equivalent serial system. 77
5.10 Nagging configuration used in second experiment. 78
5.11 Comparison of a more sophisticated 16-nagger system and an equivalent serial system. 79

6.1 The iterative strengthening algorithm. 84
6.2 Admissible vs inadmissible heuristics for iterative strengthening. 87

7.1 Search performance of a non-caching iterative-deepening theorem prover 91
7.2 Search performance of an iterative-deepening theorem prover with a 45-element LRU cache. 92
7.3 Search performance of an iterative-deepening theorem prover using EBL*DI. 93
7.4 Search performance of an iterative-deepening theorem prover using a 45-element LRU cache and EBL*DI. 94
7.5 Potential for redundancy control through success cache. 95
7.6 Benefits of caching for iterative strengthening. 97
7.7 Overhead of caching for iterative strengthening. 98

8.1 Comparison of three ALPS inference engines 101
8.2 The ALPS TPFDD Simulator. 102

9.1 The state-based plan repair algorithm. 107
9.2 The goal-based plan repair algorithm. 109
9.3 The algorithm for generating state properties. 110
9.4 An example goal structure. 111
9.5 The algorithm for constructing the goal structure. 112
9.6 The initial state. 114
9.7 The original plan. 114
9.8 The goal structure. 115
9.9 The first solution of subproblem P_i. 116
9.10 The final solution of subproblem P_i. 116
9.11 The final plan. 117

10.1 The ALPS simulator process queue. 124

List of Tables

4.1 Summary Results for LT Domain. 55

9.1 Terminology for generic and transportation planning. 117
9.2 A sample transportation problem. 119

Chapter 1

Introduction

The **ALPS** *(Adaptive Learning and Planning System) project is a three-year effort to design and prototype a next-generation adaptive planning architecture as part of the ARPA / Rome Laboratory Planning Initiative (ARPI). ALPS is being used within the Planning Initiative to perform large-scale military transportation scheduling, taking a Time-Phased Force Deployment Data (TPFDD) file with thousands of cargo requests and assigning those cargos to particular transportation resources with specific embarkation and debarkation times. This chapter presents the architectural design of ALPS and gives a brief overview of the innovative techniques incorporated in the system.*[1]

The **ALPS** (Adaptive Learning and Planning System) project is a three-year effort to design and prototype a next-generation adaptive planning architecture as part of the ARPA / Rome Laboratory Planning Initiative (ARPI). ALPS is a joint project between Odyssey Research Associates (ORA), Cornell University, and the University of Iowa.[2]

Two motivating themes drive the ALPS project. The first theme is that real-world planning systems must necessarily be *adaptive*, that is, they must reconcile themselves to their environment by improving, refining, and perfecting their own behavior with practice. The second theme is that we want to see how far we can push the paradigm of planning as resource-bounded logical deduction, particularly within the somewhat atypical domain of large-scale military transportation scheduling.

Within the first theme of adaptive systems, we have conducted basic research, experimentation, and evaluation in several areas:

- We have developed *machine learning speedup techniques*, including a new domain-independent explanation-based learning algorithm and bounded-overhead success and failure caching, to improve performance with experience [88, 89, 90].
- We have produced a new method for distributing search transparently across a network of processors, called *nagging* [97, 104].

[1] This chapter is adapted from [21].

[2] Support for this research has been provided by Rome Laboratory through Contract Number F30602-93-C-0018. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

The following people have contributed to the ALPS project: Kurt Bischoff, Randy Calistri-Yeh, James Cash, Sarah Choi, Geoffrey Hird, Yungui Huang, Harshvardan Kaul, Jinghou Li, Howard Lu, Marcel Rosu, Alberto Segre, David Sturgill, Alex Vinograd, and Yunshan Zhu.

- We have developed a *probabilistic theory revision* technique for correcting flaws in domain theories [56].
- We have formulated an algorithm called *iterative strengthening* that performs anytime optimal planning [12, 13, 14].

Within the second theme of applying our techniques to transportation scheduling, we have made several advances in domain-specific methods:

- We have built a *transportation problem generator* that creates random scalable transportation scheduling problems.
- We have designed a logical domain theory and a customized *transportation scheduler* that can rapidly solve large-scale military transportation scheduling problems, scheduling 10,000 cargos on 50 squadrons of aircraft in about 3.5 minutes.
- We have implemented a *transportation simulator* that can test transportation plans for robustness in the presence of resource bottlenecks and external events.
- We have designed an *iterative plan repair* module that can work with the scheduler and simulator to fix flaws in transportation plans.

Overviews of the ALPS project are presented in [15, 16, 17, 18, 19, 20, 21]. Further information on the ALPS project can be found on the ALPS web page at:

<http://www.oracorp.com/ai/Planning/alps.html>.

The remainder of this chapter gives a brief description of the transportation scheduling domain, presents the architecture of the ALPS system, and introduces the innovative technology employed in each of the major components. Chapter 2 introduces the first of the three ALPS inference engines. Chapters 3 and 4 present details of our work on adaptive inference using caching and explanation-based learning. Chapter 5 introduces the second ALPS inference engine and describes our new method of distributed theorem proving. Chapter 7 discusses how multiple speedup techniques can be combined for synergistic benefits. Chapter 6 discusses our approach to anytime optimal planning. Chapter 8 introduces the final ALPS inference engine and discusses its use in the domain of transportation planning. Chapter 9 presents our methods of iterative plan repair. Chapters 9 and 10 describe the ALPS transportation simulator and other domain-related components. Finally, Chapter 11 summarizes the results we have obtained during this project.

1.1 ALPS and Transportation Scheduling

Within the Planning Initiative, ALPS is being used to perform large-scale transportation scheduling. A formal description of the transportation scheduling problem can be found in [30], but one possible interpretation can be informally stated as follows:

> *Given a list of schedule requirements consisting of cargo to be transported, availability and delivery deadlines, and ports of embarkation and debarkation, along with other domain constraints (such as vehicle availability), construct a plan that satisfies these requirements by specifying, for each cargo item, the vehicle and departure time (along with any other necessary information). Then carry out this plan, dynamically modifying it if changing situations require, in order to satisfy the original requirements.*

A typical transportation planning task can range from 1,500 to 200,000 movement requirements [27, App. 1], so scaleup is definitely an issue. Since the transportation domain itself is open-ended, a planner's domain theory will obviously have to begin as a simplified approximation that is progressively refined. Rapidly developing crisis situations are not conducive to complete, accurate knowledge; much information will be inaccurate, incomplete, or totally missing. These rapidly developing situations will mean that an initially viable transportation schedule may no longer work, and the schedule will have to be modified to fit the new situation.

In addition, the transportation domain is especially dependent on large amounts of temporal, geometric, and geographic knowledge. Each individual movement in a transportation plan involves complex reasoning about time intervals such as earliest arrival date to latest arrival date (EAD-LAD) and time points such as required delivery date (RDD) [3, pp. 6.34–6.36], and cargo must be divided into different categories based on what size and shape pallet is required for storage [27, App. 1].

The information that defines a particular military transportation problem is typically presented in a Time-Phased Force Deployment Data (TPFDD) database file [50]. Since real TPFDD files are difficult to acquire and are often restricted or classified, we have created a random problem generator called TGEN that produces scalable, customizable transportation scheduling problems. Each problem consists of a set of airports/seaports, a set of airplanes/ships, and a set of cargos to be transported. Each problem is further constrained by a number of factors such as required delivery times, travel times, minimum runway lengths, and weight limits. TGEN can optionally generate full TPFDD files. TGEN is available from the ALPS web page at <http://www.oracorp.com/ai/Planning/tgen.html>.

1.2 An Overview of the ALPS Architecture

The architecture design of the ALPS system is shown in Figure 1.1.

The input to the system is a query and a set of data files provided by the user. These inputs pass through a (possibly empty) series of domain-specific *pre-filters* that construct a domain theory and problem statement appropriate for ALPS. That information is then fed to an *inference engine*, which generates a solution to the user's query. The user can select any of three ALPS inference engines (two generic and one domain-specific); they differ in properties such as speed, customizability, diagnostic output, and extensibility. The plan produced by the inference engine can be optimized by the *iterative strengthening* module, a flexible anytime optimization algorithm that is layered on top of the inference engines. Once the inference engine has produced a solution to the user's query, the resulting plan is run through a *simulator* and is possibly modified by a *plan repair module*; the final answer is then passed through another set of domain-specific *post-filters* before being presented to the user.

In addition to several domain theories for classic AI domains, ALPS contains a domain theory and set of filters for scheduling TPFDD problems as described above. Figure 8.2 on page 102 shows a snapshot of the ALPS graphical user interface as ALPS is solving a transportation problem.

1.2.1 The Adaptive Inference Engines

Our inference engines are *adaptive* in the sense that their performance characteristics change with experience. Adaptive inference is an effort to bias the order of search exploration so that more problems of interest are solvable within a given resource limit.[3] ALPS achieves this bias by using

[3]Typical resource limits are CPU cycles, execution time, or memory usage. But to factor out machine dependencies, experiments usually measure the number of nodes expanded or visited.

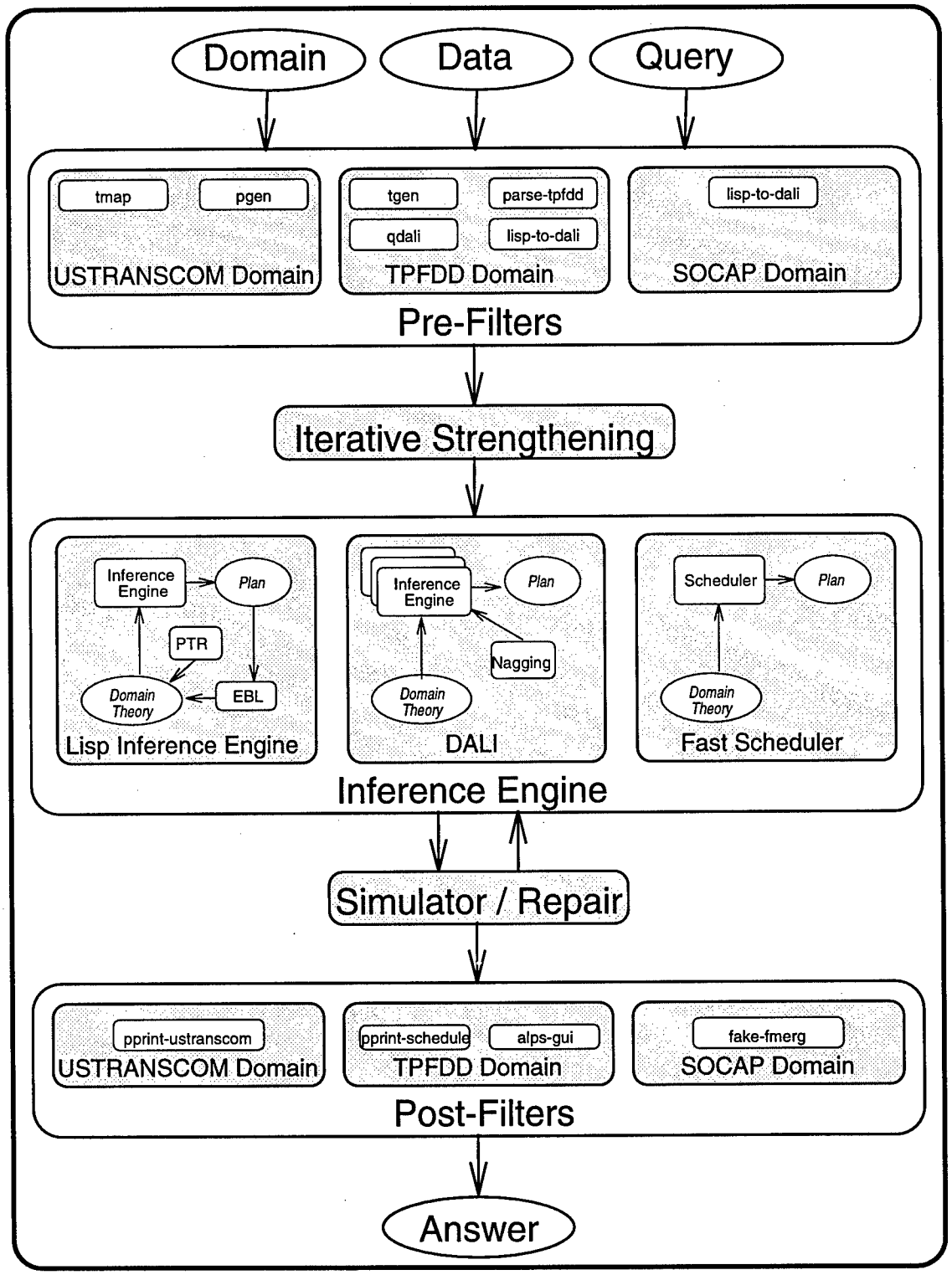

Figure 1.1: The ALPS architecture. Boxes are processes, *ellipses* are data.

multiple speedup techniques including *bounded-overhead success and failure caching*, *explanation-based learning* (EBL), and a new distributed computing technique called *nagging*. An important result of our research is that multiple speedup techniques can be applied in combination to significantly improve the performance of an automated deduction system.

The user can select the ALPS inference engine most appropriate for the current task. The Lisp Inference Engine is well suited for early development work on new domain theories since it includes better tracing and debugging features, simpler manipulation of specialized caching strategies, and the availability of an explanation-based learning (EBL) module to help extract more efficient rules. The DALI (Distributed Adaptive Logical Inference) engine is appropriate for larger problems since it is up to 30 times faster than the Lisp Inference Engine and since in addition to caching and EBL it also provides distributed computing through nagging. The "Fast Scheduler" (discussed in Chapter 8) is a special-purpose engine tailored specifically for large-scale transportation scheduling problems.

1.2.2 Caching

A *cache* is a device that stores the result of a previous computation so that it can be reused. It trades increased storage cost for reduced dependency on a slow resource. In the case of planners and deduction systems, the extra storage required to store successfully proven subgoals is traded against the increased cost of repeatedly proving these subgoals. The utility of such a cache depends on how often subgoals are likely to be repeated. Since the ALPS adaptive inference engine uses *iterative deepening* [57] to force completeness in recursive domains, we know a priori that subgoals will be repeated frequently.

It is possible to cache both successfully proven subgoals and failed subgoals. Failure cache entries may record either an outright failure (i.e., the entire search tree rooted at the subgoal was exhausted without success) or a resource-limited failure (i.e., the search tree rooted at the subgoal was examined unsuccessfully as far as resources allowed, but greater resources may later yield a solution). Future attempts to prove a cached subgoal are not undertaken unless the resources available are greater than they were when the failed attempt occurred. Success and failure caches serve to prune the search space rooted at the current subgoal. Success caches act as extra database facts, grounding the search process, while failure caches censor a search that is already known to be fruitless. Either way, they serve as effective speedup techniques by dynamically injecting bias into the search, altering the set of problems that are solvable within a given resource bound.

Allowing the cache to grow without limit will generally increase the number of cache hits, but it will also cause the cache overhead to grow monotonically, eventually outweighing any possible advantage of caching. To address this tradeoff, we make use of *bounded-overhead caches* [89]. A bounded-overhead cache is one that requires at most a fixed amount of space and entails a fixed amount of overhead per lookup. Once the cache is full, adding a new entry entails deleting an existing one. The system uses a cache management policy such as first-in-first-out (FIFO) or least-recently-used (LRU) to decide which existing entry should be replaced. Using a fixed-size cache allows us to apply information acquired in the course of solving one problem to subsequent problems, while limiting the overhead associated with a caching scheme.

1.2.3 Explanation-Based Learning

Generalizing on the caching method described above, a simple way to increase the performance of a resource-limited problem solver is to cache the proof result of each successful problem-solving episode as a new fact in the domain theory. Unfortunately, this kind of rote learning is overly

constraining because there may exist another form of the query, provable with the same pattern of reasoning implicit in the proof of the current example, that will not match the cached entry.

Much more desirable is some mechanism by which the chain of logical reasoning used in the proof can be generalized so as to be more useful and then retained and reused. This is the essence of *explanation-based learning* (EBL): we operate on the proof of the query to generalize it in some validity-preserving manner, and then we extract a new, more general rule (explanation) to extend the domain theory.

Note that the addition of a new explanation will not change the deductive closure of a domain theory, although it may well have a significant effect on the future efficiency of the prover. Once a new rule has been added to the domain theory, the hope is that when a future query requires a similar proof structure, this structure will be found more quickly thanks to the presence of the acquired rule. If the distribution of future problems is favorable, then the prover should exhibit better overall (i.e., faster) performance. It may even solve additional problems that were previously unsolvable within a fixed resource bound. Unfortunately, the effect of EBL may actually be to slow down the prover: if the macro-operator does not lead to a solution for a particular problem, it just defines a redundant path in the search space, and using the macro-operator causes a region of the search space to be searched again in vain. This undesirable effect is called the *utility problem.*

We have defined five generic operators that transform proofs in various ways. These five operators constitute the *EBL** family of algorithms [88]; a specific EBL* algorithm is defined by applying these operators in some fixed combination. EBL* is *complete* in the sense that any macro-operator extracted from a proof by any explanation-based learning algorithm can also be learned by an EBL* algorithm. This implies that since any EBL algorithm can be rewritten as some combination of the five basic operators, the main difference in EBL algorithms is the control heuristics that they use to guide the transformation process.

We have incorporated one specific set of control heuristics into a domain-independent learning algorithm called *EBL*DI* that has shown itself to be useful over a broad range of domains. The EBL*DI algorithm is superior to traditional EBL algorithms in several ways. First, it is able to acquire useful macro-operators in situations where traditional algorithms cannot. Second, it produces macro-operators of significantly greater utility than those produced by traditional EBL algorithms. Finally, the EBL*DI algorithm is truly a domain-independent learning algorithm in the sense that it is useful over a broad range of domains.[4]

1.2.4 Nagging

The second inference engine used in ALPS is called *DALI* [97]; DALI is a distributed adaptive inference engine that can run transparently in a heterogeneous distributed environment (e.g., on a network of workstations and personal computers). Like the Lisp Inference Engine, DALI uses the adaptive techniques of caching and explanation-based learning. However, DALI has advantages over single-processor inference engines in that DALI can scale to larger more realistic target problems, it provides greater reliability and fault tolerance, and it exploits the natural synergy between parallelism and speedup learning.

DALI uses a novel asynchronous parallelism scheme called *nagging* [104]. Nagging is designed to work in highly constrained nondeterministic search problems. Under the typical left-to-right, depth-first evaluation order, subgoals to the left are completely satisfied before subgoals to the right are even examined. This policy can yield extremely bad search behavior: if variable bindings early in the search preclude the solution of a later goal, this inconsistency may not be resolved until the

[4] In practice, we expect that optimal EBL strategies may well be domain dependent: taking specific knowledge of a particular domain into account should lead to better, more useful, generalizations.

searcher has performed a great deal of intermediate backtracking. Nagging is designed to alleviate this problem by asynchronously verifying that pending conjunctive goals have not been rendered unsatisfiable.

Nagging employs two types of processes running in parallel: a *master* process attempts to solve a given problem while one or more *nagger* processes attempt to assist the master. While the master is searching for a solution, the nagging processes repeatedly extract sets of unsolved goals from the master's goal stack and attempt to solve them under some of the variable bindings that the master has effected. If a nagger cannot satisfy its subset of the master's goal stack, then it is guaranteed that the master will be unable to satisfy all of its outstanding goals. The nagger can then inform the master that its current search path cannot lead to a consistent solution. If the master has not yet backtracked out of that search path, it may do so immediately without risk of missing a solution.

This nagging policy essentially performs asynchronous pruning of the search space. Ordinarily, a search-based problem solver must balance the competing interest of being fast and being smart. It must choose some combination of a strategy of performing local search quickly and one of performing global consistency checks that may obviate some local search. Nagging can be characterized as a policy of verifying global consistency constraints asynchronously and in parallel. If a nagging process detects violation of a global constraint, it can force the search to backtrack. If a nagging process fails to detect an inconsistency, its master process has wasted no time in verifying the constraint.

Nagging offers the potential of greatly speeding the search. By forcing its parent prover to backtrack early, it may prune large subtrees from the prover's search space. This policy also enjoys many of the desirable properties associated with various conventional parallelization techniques:

- As with OR-parallel models, assignment of work is initiated by idle processors; busy processors don't have to constantly stop to see if they should delegate some of their workload.
- Like many varieties of OR-parallelism, communication is infrequent, occurring only when a process needs a new search problem. Accordingly, the run-time overhead of nagging is fairly low.
- Like the stream AND-parallel strategy, nagging can benefit from the communication of partial variable bindings. Nagging makes use of the variable bindings made in a subtree before that subtree is completed.
- As with some of the work in AND-parallelism and parallel Prolog, nagging does not alter the order in which the search space is explored; it simply prunes portions of the space that are guaranteed to be useless. The first solution discovered will be the same with or without nagging.
- Since nagging only serves to prune search branches that are known to be infeasible, the search behavior on the proving process will never be worse than that of the sequential algorithm.

Nagging is particularly appropriate for distributed planning and theorem proving since, in addition to promising low communication overhead, it is also more fault-tolerant than other types of distributed search. Since interaction with a nagger only results in a master prover potentially skipping ahead in its search, the prover has no real dependence on the nagger. For bounded search problems, any search space skipped as a result of nagging would eventually be exhausted by the prover. If messages between prover and nagger are lost or delayed, the prover may explore unnecessary search space, but it will eventually return an identical solution.

1.2.5 Iterative Strengthening

To perform adequately in real-world situations, a planning system must do more than simply generate a plan that satisfies the user's goals. In many domains, a given problem statement may have multiple solutions, and the user typically will want the *best* solution (although the criteria for "best" may change from one user to another or one problem to another). Additionally, many domains are time-critical and require support for "anytime" behavior. In this context, an anytime algorithm is one in which a solution is incrementally refined over time; if the algorithm is run to completion it will find an optimal solution, but the user can interrupt it at any point and demand a useful (but not necessarily optimal) solution.

We have developed an algorithm called *iterative strengthening* [18, 19], a flexible method of producing optimized plans where the user's criteria for optimization may change during the planning session. Iterative strengthening has the following properties: (1) the underlying knowledge base is independent of any specific optimizing parameters; (2) users can easily switch between different sets of optimizing criteria; (3) the method supports optimized planning within an "anytime" environment; (4) the method is consistent with Prolog-style inference engines such as the ALPS adaptive inference engine; and (5) the method can be used in situations where the optimality constraints are inadmissible or where the domain theory is undecidable. We have implemented this method in the ALPS planning system and have tested it in the domain of crisis-action transportation planning with optimality criteria such as total transport time, number of aircraft, and probability of success.

Iterative strengthening is related to the concept of iterative deepening (in which the system searches to a given depth in the search tree for a solution, and if none is found, the system restarts the search from the beginning with a larger depth cutoff). The iterative strengthening algorithm first performs an unconstrained search for any satisficing solution to the planning problem. When it finds that solution, it restarts the search, but now constrains the solution to be "better" than the first solution by some "increment" (where "better" is measured by an optimization function specified by the user and "increment" is a function applied to the optimization parameters of the current plan). For example, if the goal is to find the plan that takes the minimum time to execute, and if the system has already found a plan that takes n minutes, it will restart the search constraining the new plan to $n - \delta$, where δ is a user-defined constant. The system continues strengthening the optimization parameters until no more solutions can be found; the last solution is the optimal answer.

Although iterative strengthening may take longer to find the final optimized plan than other optimal search algorithms such as A^* [79], iterative strengthening has the advantage that it can be interrupted at any time after the initial plan is found and will always have a valid plan available for the user. Since this initial plan is found using satisficing criteria instead of optimizing criteria (i.e., since we first concentrate on finding a simply correct plan rather than a fully optimal one), it is likely that iterative strengthening will generate a valid plan significantly faster than algorithms such as A*. In other words, iterative strengthening supports incremental improvements to existing valid plans; it can deliver an initial plan promptly and then spend any remaining time improving it until an optimal plan is discovered or until the available planning time is exhausted. Additionally, iterative strengthening can be used in situations where other techniques will not work at all, such as inadmissible search heuristics and undecidable domains (see Chapter 6).

1.2.6 The ALPS Simulator

In the transportation planning domain, once the ALPS inference engine generates a schedule, the schedule is passed along to the simulator. The simulator performs two primary services. First, it

analyzes the schedule at a finer level of detail than the inference engine did. This analysis allows the simulator to identify resource contentions and bottlenecks that the inference engine would have missed. Second, the simulator can test the schedule for robustness in the presence of unanticipated difficulties by simulating nondeterministic external events (such as storms, mechanical failures, or terrorist activity) that may affect the outcome of the schedule .

The ALPS simulator is based on an object-oriented design. The simulator takes as input the initial world state (locations of cargos, allocation of transportation assets, etc.) that was given to the inference engine, along with the schedule that the inference engine generated. It constructs a stream of events and executes these events in a simulated world, reporting the results of this simulation. It simulates resource bottlenecks using *monitors* that manage and allocate resources.

The transportation domain theory currently used by ALPS deliberately ignores resource contention when constructing a schedule; it verifies that the necessary resources exist but does not verify that they are available for use. The rationale behind this design decision is that there is no point in scheduling a particular airplane to land on a particular runway within a 10-minute window one week in the future because in real life the schedule will have broken down long before it ever reaches that point. By using the simulator, ALPS can test whether bottleneck conditions are likely to occur without committing the schedule to an unreasonable level of detail. The results of the simulation are sent to the ALPS plan repair module (described below), which will make local modifications to correct any identified deficiencies.

1.2.7 Plan Repair in ALPS

The ALPS plan repair module [117] uses a general iterative repair technique that has been customized to work with the ALPS Fast Scheduler and transportation simulator. It takes as input an existing plan from the inference engine, a list of failures from the simulator, and an optional list of problem modifications from the user; the output of the repair process is an updated plan. ALPS uses a basic heuristic assumption that most failures can be fixed with local modifications (if a failure involves global changes to the original plan, it is unlikely that any repair method will be better than simply replanning from scratch).

The approach used by ALPS exploits this locality of plan repair and maintains completeness by doing iterative replanning. ALPS repairs a plan by retracting actions that are "local" to the failure, formulating a new planning problem based on the goals of those retracted actions, and solving that problem to generate a replacement sequence of actions. It continues retracting and replacing actions iteratively until the resulting plan is correct.

In the transportation domain, we have found two ways of defining "locality" that are particularly useful. One is to order trips based on airplanes (*single plane repair (SPR)*); the other is to order trips based on departure times (*multiple plane repair (MPR)*).

For SPR, we restrict the set of retracted actions to be within one single airplane schedule. Initially, the repair module retracts the single identified failed trip, updates its temporal intervals, and tries to fit this updated trip back in the original schedule for this airplane. If the updated trip does not fit, the module will iteratively retract trips before and/or after the initial faulty trip, reschedule all cargos on these trips in isolation (using only this airplane), and try to fit the new subschedule back into the original schedule. The iteration stops when the rescheduled trip sequence fits in the airplane schedule (possibly displacing some cargos because they are no longer possible to schedule).

MPR has the added ability of rearranging cargos among multiple airplanes. Initially, the repair module tries to insert an undelivered cargo directly into the existing global schedule. If this insertion is not successful, MPR will iteratively retract cargo trips within a certain time interval to create a

"window" across all airplane schedules and will try to fit all cargos back into the global schedule (not necessarily on their original airplanes). The iteration succeeds if the undelivered cargo *and all retracted cargos* fit in the schedule; otherwise the undelivered cargo is marked as "too hard".

Interestingly, SPR and MPR can be combined to handle different types of failures very efficiently. By ordering trip schedules differently, SPR and MPR can exploit two different views of locality to perform different types of "local repairs". SPR is more appropriate for handling delayed trips and deadline violations because these failures can most often be avoided by adjusting trips locally within the same airplanes. On the other hand, undelivered cargos or airplane failures often require the collaboration of multiple airplanes in MPR, and the most relevant trips are the ones clustered locally around similar departure times.

Using temporal locality can also help to minimize the changes to the overall plan structure. Optimally conservative plan modification is computationally intractable [76], but we do not necessarily need to absolutely minimize the number of changed actions. In the transportation domain, for example, it may be less intrusive to reorder 50 trips within one single airplane than to replace 20 trips spread across many airplanes. Using a combination of SPR and MPR in this domain clusters the changes naturally, producing good locality of modification without requiring optimal conservation.

Chapter 2

The ALPS Lisp Inference Engine

This chapter introduces the ALPS Lisp Inference Engine.[1]

2.1 Introduction

The ALPS Lisp Inference Engine is *adaptive* in the sense that its performance characteristics change with experience. While others have previously suggested augmenting Prolog interpreters with explanation-based learning components [82], our system is the first to integrate advanced speedup techniques such as explanation-based learning and bounded-overhead success and failure caching. Adaptive inference is an effort to bias the order of search exploration so that more problems of interest are solvable within a given resource limit. Adaptive methods include techniques normally considered speedup learning methods as well as other techniques not normally associated with machine learning. All the methods that we consider, however, rely on an underlying assumption about how the inference engine is to be used.

The goal of most work within the automated deduction community is to construct inference engines that are fast enough and powerful enough to solve very large problems once, then to move on to another unrelated problem. In contrast, we are interested in using our inference engine to solve a collection of related problems drawn from a fixed (but possibly unknown) problem distribution. These problems are all solved using the same domain theory. A complicating factor is that the inference engine is operating under rigid, externally imposed resource constraints.

For example, in the transportation scheduling domain, a stream of queries corresponding to transport requests are passed to the inference engine; the inference engine uses a logical formulation of domain knowledge to derive sequences of actions that are likely to achieve the goal. Since much of the world does not change from one query to the next, information obtained while answering one query can dramatically affect the size of the search space that must be explored for subsequent ones. The information retained may take many different forms: facts about the world state, generalized schemata of inferential reasoning, advice regarding fruitless search paths, etc. Regardless of form, however, the information is used to alter the search behavior of the inference engine. All of the adaptive inference techniques we employ share this same underlying theme.

[1] This chapter is adapted from [90].

2.2 Proofs and Inference

Before describing our inference engine, we must establish the underlying knowledge-representation formalism. For the purposes of this chapter, it is reasonable to adopt a very simple formalism: our choice is one involving only *facts* and *rules*. Facts are atomic formulae, or atoms, such as $fragile(chippendale)$ or $expensive(?x)$, where the leading question mark is used to indicate a logic variable.[2] Rules are implications, such as $light(?x) \leftarrow on(?x,?y) \wedge fragile(?y)$, where the *head* is $light(?x,?y)$ and the *antecedents* are $on(?x,?y)$ and $fragile(?y)$. Technically, facts and rules are both first-order definite clauses, with function symbols allowed, but with no special equality predicate. This same formalism underlies most of the work in the logic programming community, in particular the Prolog programming language.

In this formal framework, a *domain theory* consists of an initial set of facts and rules. The domain theory entails a certain *deductive closure*, which is the collection of all atomic formulae that follow logically from the given domain theory. Problem-solving consists of determining whether or not a given *query*, which may contain some number of existentially quantified variables, is a member of this deductive closure. The query is a member of the deductive closure if an explanation justifying the truth of some substitution instance of the query, *i.e.*, a *proof*, can be constructed.

Given our knowledge representation formalism of facts and rules, proofs are tree-structured and recursive. Formally:

Definition 1 : A *proof* is a tree composed of two types of nodes, *consequent nodes* and *subgoal nodes*, and two types of edges, *rule edges* and *match edges*. Each node n has two tags, a *formula*, denoted $f(n)$, and a *label*, denoted $l(n)$, which are atomic formulae.

A consequent node corresponds to the head of a domain theory rule, while a subgoal node corresponds to an antecedent of a domain theory rule. A match edge links a parent subgoal node to a (unique) child consequent node, while rule edges are used to link a parent consequent node to its child subgoal nodes.

Definition 2 : A *consequent node* n_c is a node with zero or more children, denoted $r(n_c)$, connected to n_c via outgoing *rule edges*, and a lone parent, denoted $p(n_c)$.

Definition 3 : A *subgoal node* n_s is a node with at most one child, denoted $m(n_s)$, connected to n_s via an outgoing *match edge*, and a lone parent, denoted $p(n_s)$.

The root $root(\rho)$ of a proof ρ is always a subgoal node representing the original query q.
For a given query, problem-solving activity may in general yield zero, one, or more proofs.

Definition 4 : A *problem-solving episode* $\Pi_R(q)$ for a given query q and resource bound R yields a series of results $\langle \pi_i \rangle_{i=1}^{i=n}$ such that $n \geq 1$ and

1. $\pi_i = \rho_i$ for $i < n$, where ρ_i is a proof with $l(root(\rho_i)) = q$ and corresponding *answer substitution* $\theta_i = l(root(\rho_i)) \circ f(root(\rho_i))$; and

[2] Atomic formula, predicate, function, variable, substitution, substitution instance, and other related terms are defined in [64]. In addition to the notation used by Lloyd, we will use $\circ$ to denote the "unify" relation (*i.e.*, $a \circ b$ *iff* $\exists\theta$ *such that* $a\theta = b\theta$), $\subseteq$ to denote the relation "is a substitution instance of" or "is at least as specific as" (*i.e.*, $a \subseteq b$ *iff* $\exists\theta$ *such that* $a\theta = b$), $\subset$ to denote the relation "is a non-trivial substitution instance of" or "is strictly more specific than" (*i.e.*, $a \subset b$ *iff* $a \subseteq b \wedge \neg b \subseteq a$), $=$ to denote the relation "is a variable-renaming substitution instance of" or "is exactly as specific/general as" (*i.e.*, $a = b$ *iff* $a \subseteq b \wedge b \subseteq a$), and $\equiv$ to denote the relation "is identical to."

2. π_n = *fail*, indicating that no further substitution instance of the query q lies within the deductive closure D; or π_n = *limit*, indicating that no further substitution instance of the query q lies within the resource-limited deductive closure D_R (although one may well lie within D).

It is sometimes more convenient to refer to the answer substitution series $\mathrm{A}_R(q) = \langle\theta_i\rangle_{i=1}^{i=n-1}$ instead of its corresponding problem-solving episode $\Pi_R(q)$.

Next we introduce a notion of soundness for proofs. Informally, a proof is *valid* if it is deductively correct. More formally:

Definition 5 : A proof ρ is *valid* if and only if

1. for each subgoal node with an outgoing match edge $n_s \in \rho$, node formulae are identical across the match edge:
$$f(n_s) = f(m(n_s));$$

2. for each consequent node $n_c \in \rho$, the logical implication
$$l(n_c) \leftarrow \bigwedge_{n_s \in r(n_c)} l(n_s)$$
follows deductively from the original domain theory; and

3. for each consequent node $n_c \in \rho$, the logical implication
$$f(n_c) \leftarrow \bigwedge_{n_s \in r(n_c)} f(n_s)$$
is a substitution instance of the logical implication
$$l(n_c) \leftarrow \bigwedge_{n_s \in r(n_c)} l(n_s).$$

The validity of a proof is independent of the original query and the problem-solving system used to construct it; rather, validity is an intrinsic property of the proof structure.

2.2.1 A Sample Proof

It is useful to look at a complete example of a valid proof. Consider the following simple domain theory consisting of just nine facts and three rules:

$$\begin{array}{rcl} k(D) \quad p(B) \quad q(?y) & & \\ k(h(?w)) \quad n(h(A)) \quad j(A) & & \\ p(A) \quad n(h(B)) \quad j(C) & & \\ s(?a) & \leftarrow & q(?b) \wedge r(?a, ?b) \\ r(?c, ?d) & \leftarrow & p(?e) \wedge m(?c, ?e) \wedge n(?c) \\ m(?f, ?g) & \leftarrow & j(?g) \wedge k(?f) \end{array}$$

The first result, ρ_1, of the problem solving episode $\Pi(s(?x))$ is shown in Figure 2.1.[3] The subgoal node $root(\rho_1)$ at the top represents the original query q, with $l(root(\rho_1)) = q = s(?x)$ and $f(root(\rho_1)) = s(h(A))$, a substitution instance of q with answer substitution $\theta = \{?x/h(A)\}$. The

[3] We omit the subscript R when no resource limit is imposed on the search.

consequent node directly below $root(\rho_1)$, $m(root(\rho_1))$, has label $l(m(root(\rho_1))) = s(?a)$, the head of the matching domain theory rule.[4] Each subgoal descendent has its label set to the corresponding rule antecedent, here $q(?b)$ and $r(?a, ?b)$, and its formula set to the appropriately instantiated version of the antecedent, here $q(A)$ and $r(A, A)$. Nodes connected by match edges have identical formulae, while the leaves of the explanation are childless consequent nodes whose labels correspond to domain theory facts and whose formulae correspond to appropriate instances of those facts. Thus it is clear that for any subgoal node n_s, the subtree rooted at n_s provides a valid proof for $f(n_s)$.

2.3 The ALPS Lisp Adaptive Inference Engine

We have implemented a backward-chaining definite-clause inference engine (referred to in this report as the "Lisp Inference Engine") that returns valid proof structures of the form just described. The inference engine's inference scheme is essentially equivalent to Prolog's SLD-resolution inference scheme. Axioms are stored in a discrimination net database along with rules indexed by the rule head. The database performs a pattern-matching retrieval guaranteed to return a superset of those database entries that unify with the retrieval pattern. The cost of a single database retrieval in this model grows linearly with the number of matches found and logarithmically with the number of entries in the database.

Like all definite-clause inference engines, ours searches an implicit AND/OR tree defined by the domain theory and the query, or goal, under consideration. Each OR node in this implicit AND/OR tree corresponds to a subgoal that must be unified with the head of some matching clause in the domain theory, while each AND node corresponds to the body of a clause in the domain theory. The children of an OR node represent alternative paths to search, while the children of an AND node represent sibling subgoals that require mutually consistent solutions.

The search strategy determines the order in which the nodes of the implicit AND/OR tree are explored. Different exploration orders correspond not only to different resource-limited deductive closures D_R, but also to different solutions of the queries in D_R as well as different node expansion costs. For example, breadth-first inference engines guarantee finding the shallowest solution, but require excessive space for problems of any significant size. Depth-first inference engines require less space, but risk not terminating when the domain theory is recursive. Choosing an appropriate search strategy is a critical design decision when constructing an inference engine.

Our system relies on a well-understood technique called *iterative deepening* [57] for forcing completeness in recursive domains while still taking advantage of depth-first search's favorable storage characteristics. As generally practiced, iterative deepening involves limiting depth-first search exploration to a fixed depth. If no solution is found by the time the depth-limited search space is exhausted, the depth limit is incremented and the search is restarted. In return for completeness in recursive domains, depth-first iterative deepening generally entails a constant factor overhead when compared to regular depth-first search: the size of this constant depends on the branching factor of the search space and the value of the depth increment. Changing the increment changes the order of exploration of the implicit search space and therefore the performance of the inference engine.

Our inference engine performs iterative deepening on a generalized, user-defined notion of depth while respecting the overall search resource limit specified at query time. Fixing a depth-update

[4] In practice, domain theory rules must be "standardized apart" (*i.e.*, a unique variable renaming substitution must be applied to rules at application time) to avoid variable name conflicts between multiple occurrences of the same rule. For clarity, we ensure that variable name conflicts do not occur in the examples of this chapter, so the variable names in figures match those in corresponding original domain theory rules.

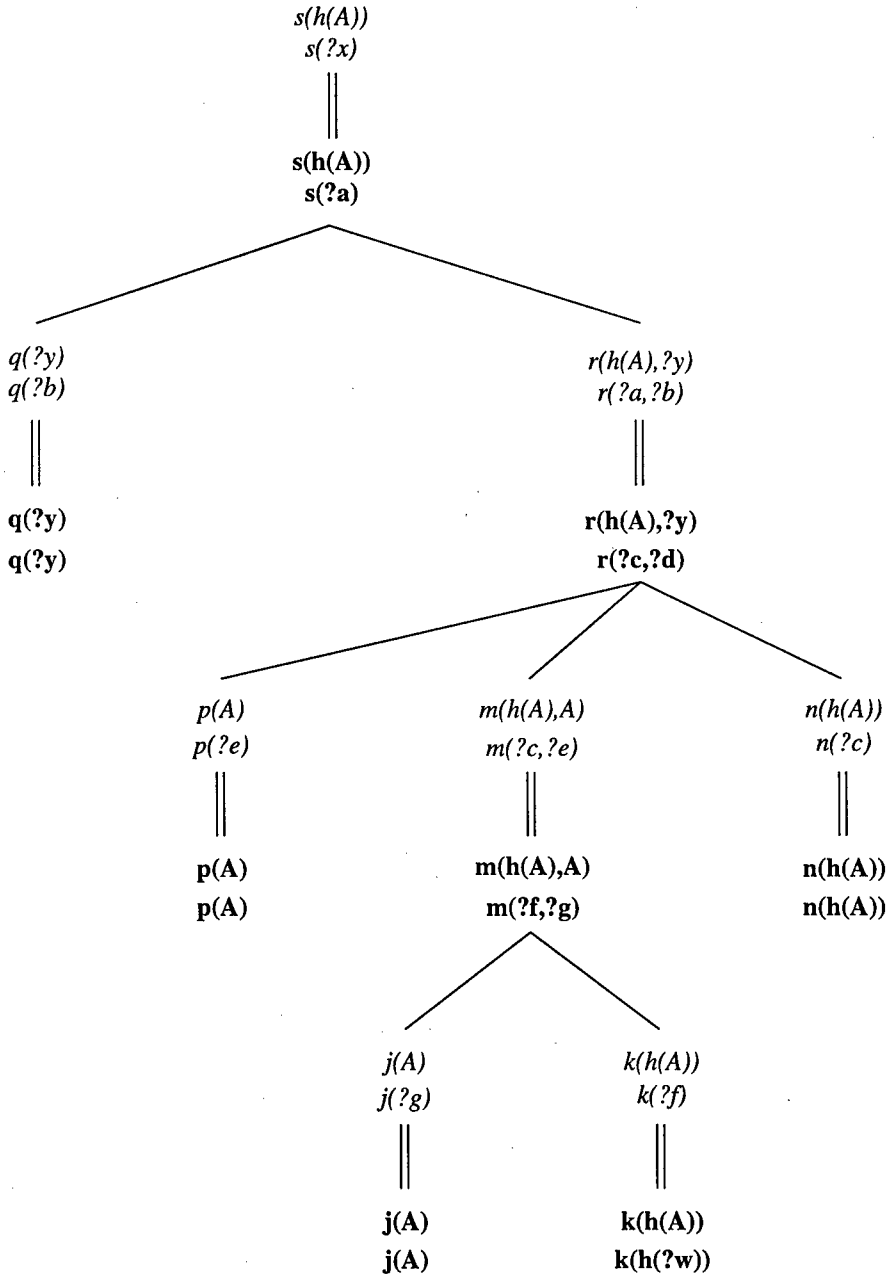

Figure 2.1: Sample proof. Bold face font is used to indicate consequent nodes, while italic font corresponds to subgoal nodes. The upper expression is the node formula, while the lower expression is the node label. Double lines represent match edges, while rule edges are represented with single lines.

function (and thus a precise definition of depth) and an iterative-deepening increment establishes the exploration order of the inference engine. For example, one might define the iterative-deepening update function to compute depth of the search; with this strategy, the system is performing traditional iterative deepening. Alternatively, one might specify update functions for *conspiratorial iterative deepening* [37], *iterative broadening* [41], or numerous other search strategies.

Chapter 3

Success and Failure Caching

This chapter[1] *surveys our work on adaptive inference and reports on the experiments we have performed. In particular, it reports on our work with bounded-overhead caching for definite-clause theorem provers and describes a particular adaptive inference engine that is used within the ALPS system.*

3.1 Introduction

A *cache* is a device that stores the result of a previous computation so that it can be reused. It trades increased storage cost for reduced dependency on a slow resource. The use of caches has been proposed for storing previously proven subgoals (*e.g.*, *success caching*) in automated deduction systems [80]. Here the extra storage required to store successfully-proven subgoals is traded against the increased cost of repeatedly proving these subgoals. The utility of such a cache depends on how often subgoals are likely to be repeated; in the case of iterative deepening, we know *a priori* that subgoals are repeated frequently.

In addition to caching successfully-proven subgoals, caching failed subgoals can also improve performance [37]. These *failure caches* record failed subgoals, along with the resource bounds in force at the time of the failures. Future attempts to prove a cached subgoal are not undertaken unless the resources available are greater than they were when the failed attempt occurred. Failure cache entries may record either an outright failure (*i.e.*, the entire search tree rooted at the subgoal was exhausted without success) or a resource-limited failure (*i.e.*, the search tree rooted at the subgoal was examined unsuccessfully as far as resources allowed, but greater resources may later yield a solution). Resource-limited failure cache entries must contain an additional annotation, describing the resources available at the time of the cached failure.

Success and failure caches affect the search at OR-node choice points. In their simplest forms, they serve to prune the search space rooted at the current subgoal. Success caches act as extra database facts, grounding the search process, while failure caches censor a search that is already known to be fruitless. Either way, they serve as effective speedup techniques by dynamically injecting bias into the search, altering the set of problems that are solvable within a given resource bound.

[1] This chapter is adapted from [90].

3.2 Bounded-Overhead Caching

A *bounded-overhead* cache is one that requires at most a fixed amount of space and entails a fixed amount of overhead per lookup. In our implementation, success and failure entries coexist in a single, fixed-size cache. At each OR-node choice point, the inference engine first checks the cache for a matching success entry (called a *cache hit*). If one is found, possibly by introducing new variable bindings, the subgoal is considered solved. If no matching entry is found, the inference engine checks for a failure entry. If it finds one with a sufficiently large resource limit, the subgoal is considered unsolvable, and the inference engine is forced to backtrack. If neither type of cache hit occurs, the inference engine proceeds to try proving the subgoal normally. When finished, it inserts a new entry into the cache: a success entry if the subgoal is solved, and a failure entry if no proof is found within the current resource bounds.

Once the cache is full, adding a new entry entails deleting an existing one. A *cache management policy* is used to decide which existing entry should be replaced. Cache management policies are nothing more than heuristics that assign relative importance to cache entries. Simple replacement policies such as *first-in-first-out* (FIFO), *least-recently used* (LRU), and *least-frequently used* (LFU) are suggested by analogy with paged memory systems. These cache management strategies exploit knowledge about memory access patterns. For paged memory systems, empirical studies of memory traces have shown that both programs and data exhibit *locality of reference*; that is, access patterns tend to cluster in locally-constrained areas of memory. In automated deduction, one might expect iterative deepening to exhibit some property that can serve in place of locality of reference; an analytic understanding of this property would certainly aid in designing high-performance management policies for automated deduction caches. For now, we continue to rely on simple policies such as LRU while actively studying the problem of designing high-performance cache management policies for iterative deepening.

Using a fixed-size cache permits us to apply information acquired in the course of solving one problem to subsequent problems, while limiting the overhead associated with a caching scheme. Unfortunately, even a bounded-overhead cache may adversely affect performance. To see why this is so, consider the interaction of a simple success cache with the inference engine's backtracking behavior. When forced to backtrack over a subgoal that has matched a success cache entry, the inference engine will necessarily consider all alternative paths at that choice point. Since cache entries represent deductively entailed — and therefore redundant — information, some of the alternate paths considered at this choice point are subsumed by the matching cache entry that has just failed. Thus the inference engine will waste time exploring some alternate paths that are known *a priori* to be fruitless. By increasing the branching factor with redundant choice points, unsuccessful cache entries may actually cause an inflated number of nodes to be searched.[2]

We can avoid this problem in a general sense by restricting the applicability of cache entries and changing the backtracking behavior of the inference engine at cache hits [37]. By permitting success cache hits only where the candidate cache entry is at least as general as the current subgoal, we can ignore alternative choice points when backtracking over a cache hit. Failure cache hits are also restricted to situations where the cache entry is at least as general as the subgoal, but in addition the current resource limit must be dominated by the resource limit associated with the cache entry. These *cache hit generality constraints* prevent a cache hit from binding variables in the current search context, eliminating the need to consider any alternate search paths that may exist at this subgoal. Once a cache hit occurs, the entire search space rooted at that subgoal is effectively pruned and need not be explored upon backtracking. Thus, although imposing cache hit

[2]This problem is related to the *utility problem* found in speedup learning systems; see Section 4.3 [69].

generality constraints produces less frequent cache hits, it avoids adverse search effects altogether.[3]

3.3 Evaluating Bounded-Overhead Caching

This section empirically measures the performance of our caching system, contrasting various caching strategies and configurations. We have studied many aspects of cache design, including the relative performance of different cache management policies, the coexistence of success and failure entries in a unique cache, and the impact of redundant cache entries on system performance.

3.3.1 Methodology

It is difficult to extrapolate reliably from empirical data. In [94] we outline some common methodological problems encountered in experimental evaluations of speedup learning systems. In [95], we present an experimental methodology for comparing speedup learning systems that avoids many of these pitfalls. Since caching can also be viewed as a form of speedup learning, we can adopt some of these techniques to our evaluation of caching. These techniques allow us to obtain a more precise, quantitative picture of the effect caching has on performance.

The experimental methodology used here follows that introduced in [95] and later refined in [96] and [43]. It is based on a mathematical model of theorem proving as search; our basic assumption is that, independent of a particular theorem-proving system's implementation details, the size of the space explored — and therefore the time required to search — grows exponentially with the difficulty of the problem being solved. More formally, we can relate the time t to solve a problem of difficulty δ in a search space with average branching factor b and per-node exploration cost c as

$$t = cb^{\delta}. \tag{3.1}$$

By measuring t over a collection of problems of known difficulties, we can derive estimates of b and c using standard methods of parametric statistics. Direct performance comparisons between two different theorem provers — or the same theorem prover operating with different cache configurations — solving representative suites of test problems can be made by comparing their respective b and c parameters. If b for one is lower than b for the other, then, in the limit (*i.e.*, for difficult enough problems), we can conclude that the first theorem prover will perform systematically faster than the second.

For the experiments reported here, we use a breadth-first search control system to solve each problem in the test suite and use the number of nodes explored (e_{bfs}) as an approximation of problem difficulty δ. Thus, given a number of datapoints of the form $(\log(e_{bfs}), \log(t))$ obtained on a collection of test problems, we can obtain estimates of the regression parameters $\log(b)$ and $\log(c)$ using linear regression in accordance with the following regression model:

$$\log(t) \approx \log(b)\log(e_{bfs}) + \log(c). \tag{3.2}$$

A lower regression slope $\log(b)$ in general corresponds to a theorem prover whose performance scales better to larger problems. The main advantage of this methodology is that it allows us to predict performance on relatively large problems from data collected on relatively small problems.

[3] An unfortunate side effect of imposing generality constraints is the problem of introducing duplicate cache entries. Duplicate entries, if handled consistently, should eventually be deleted by any reasonable cache-management strategy.

3.3.2 Experiment 1

In this first experiment, we are interested in comparing the performance of a non-caching theorem prover with an identical theorem prover that uses an unlimited-size success and failure cache.

This study uses our depth-first iterative-deepening definite-clause theorem prover described previously in Section 2.3. As the theorem prover expands each subgoal, it checks the cache first, and only resorts to checking the database if necessary. The cache implementation is flexible, allowing the user to vary cache size and to specify arbitrary cache management strategies. Note that the theorem prover is not particularly fast, since it was designed primarily in order to support principled experimentation. For example, like the caching subsystem, the search strategy used by the theorem prover is flexible; it can be configured to perform iterative deepening, iterative broadening, conspiratorial best-first iterative deepening, or even simple breadth-first search. In fact, the same theorem prover (configured to perform breadth-first search) is used as the control system.

The domain theory and problem set used for this experiment consist of 26 problems drawn from a simple situation-calculus formulation of the classic AI block-stacking world [106]. Each problem is solved by the control theorem prover, a non-caching breadth-first search configuration of the theorem prover. The smallest problem requires searching 4 nodes and corresponds to a derivation tree consisting of 4 nodes 1 level deep. The largest problem requires searching approximately 16,000 nodes and corresponds to a derivation tree of 84 nodes 7 levels deep. The logarithm of the number of nodes explored $\log(e_{bfs})$ is recorded for each problem for use as the estimator of problem difficulty δ.

We performed two trials using the depth-first iterative-deepening theorem prover. The first trial involved no caching, while the second trial used an unlimited-size cache. Each trial consisted of solving all 26 problems presented in the same random order using a resource limit of 30,000 nodes explored. In the second trial, the cache was not cleared between problems. Both trials were performed using a unit-increment iterative deepening strategy.[4] We recorded elapsed time to solution (in milliseconds) for each of 26 problems solved, and we performed a two-parameter linear regression using Equation 3.2 as the regression model.

Figure 3.1 illustrates the performance of the non-caching system. As might be expected, this system achieves an excellent fit ($r^2 = 99.8\%$), since the unit-increment iterative deepening system and the control system explore the search space in identical order. Intuitively, this helps to lend credence to our methodology's underlying mathematical model by illustrating how the number of nodes exploited by a control system can in fact be excellent predictors of CPU time performance for a different system operating with the same domain theory on the same problem set.

Figure 3.2 shows the performance of the unlimited-size caching system. While the performance of the unlimited-size caching system is dependent on problem ordering, the performance of the non-caching system is not: nonetheless, the problems are presented in exactly the same random order as for the non-caching system of Figure 3.1. As noted previously, the cache is not flushed between problems. The 26 problems are solved in a total of 669.8 seconds, at which point the cache contains a total of 10,753 entries, 1,304 of which served to provide a cache hit at some time during the trial.

The plot in Figure 3.2 suggests several striking observations. First, we note that almost all datapoints in this second plot have greater y-values than their corresponding datapoint in Figure 3.1: thus, on this randomly-ordered set of test problems, the unlimited-size caching system is slower than the non-caching system on almost every problem. In fact, the caching system is more than

[4] A unit increment may well produce the worst-case performance for iterative deepening. Depending on the problem population, increasing the increment value may substantially improve the system's performance.

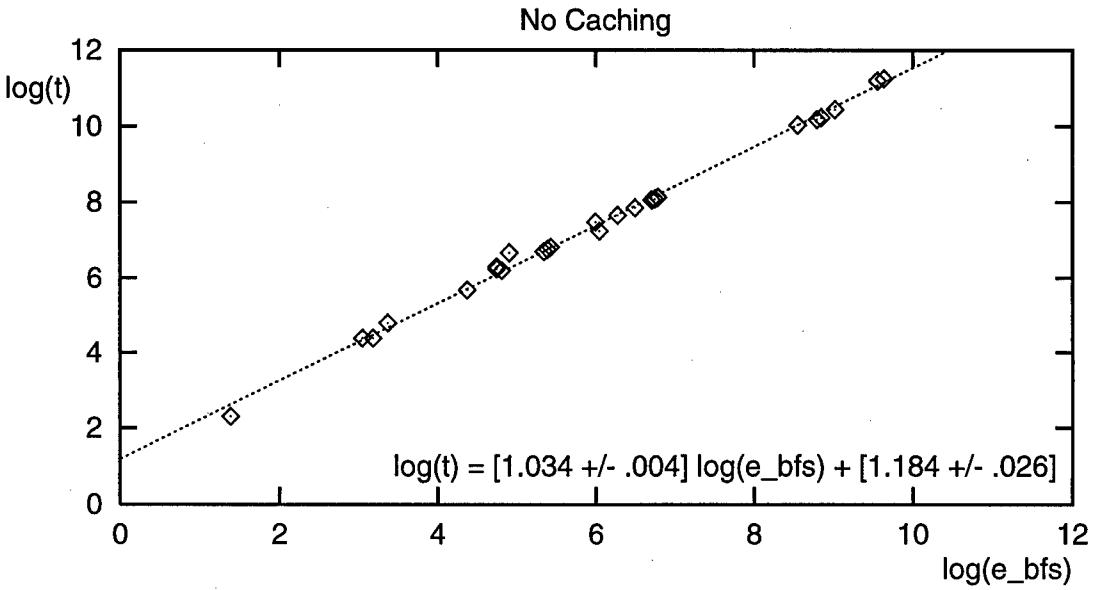

Figure 3.1: Performance of a non-caching iterative-deepening theorem prover on 26 problems from the situation-calculus domain theory of Experiment 1. Each datapoint shown corresponds to one or more problems, since some problems have exactly the same solution characteristics. Total time to solve 26 test problems was 273.9 seconds.

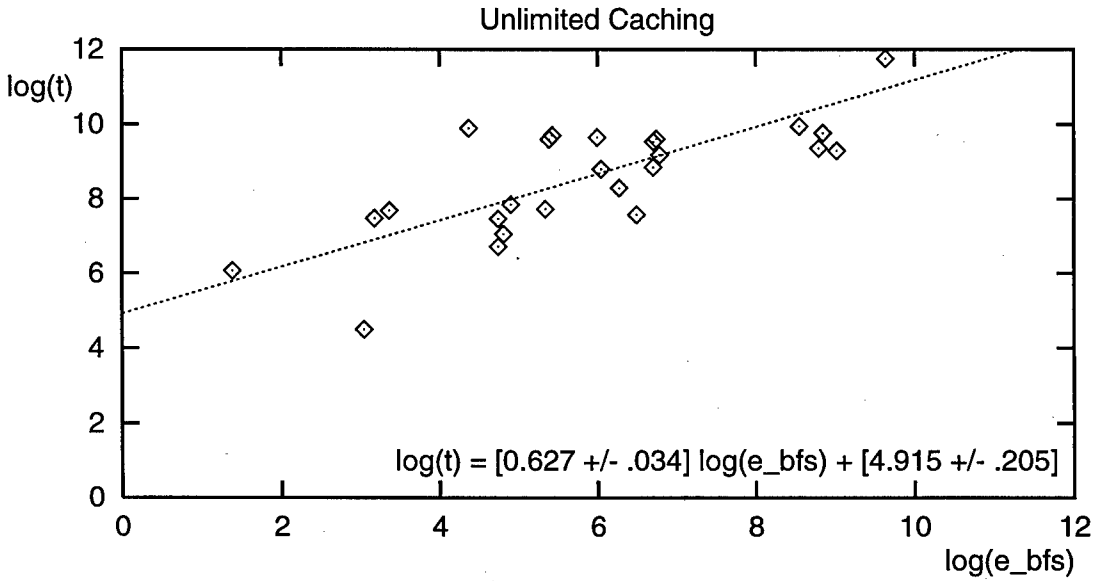

Figure 3.2: Performance of an unlimited-size caching iterative-deepening theorem prover on the same situation-calculus problem set of Figure 3.1. Total time to solve all 26 test problems was 669.8 seconds.

twice as slow as the non-caching system over the test problem set as a whole. Second, from the regression parameters obtained, it appears that the unlimited-size caching system has, as expected, a greater node exploration cost than the non-caching system. This greater node exploration cost c reflects the cache overhead costs and shows up in the plot as a larger y-intercept ($\log(c)$ in Equation 3.2) value. Finally, given the lower computed regression slope for the unlimited-size caching system, we might expect that, on large enough problems, the caching system will be faster.

Is this last conclusion warranted? Unfortunately, no: the problem lies in our methodological assumption that c is invariant for a given theorem prover, domain theory, and problem set. For the unlimited-size caching system, however, c clearly grows monotonically as more items are added to the cache. Thus while it might appear by extrapolation that, for large enough problems, the unlimited-size caching system will prove faster than the non-caching system, this may not be true given that the intercept value for the unlimited-size caching system will continue to increase.[5] Whether or not the unlimited-size caching system will ever prove to be quicker than the non-caching system depends on the implementation, domain theory, and problem distribution.

There is one other interesting piece of information that can be reliably extracted from our experiment with the unlimited-size caching system. In particular, we would very much like to know the magnitude of the beneficial search effect possible due to caching. As noted previously, the beneficial search effect due to unlimited-size caching represents a sort of empirically measured best-case reduction in search available for any caching scheme.

To isolate search effects due to caching from cache overhead, we make a small modification to our experimental methodology. Substituting t directly as the dependent variable in the experiment, we factor out the cache overhead leaving

$$\log(e) = \log(b)\log(e_{bfs}) \tag{3.3}$$

as the experimental regression model. This simplified model highlights implementation-independent search effects without conflating implementation-dependent cache overheads. The single-parameter regression equation also reflects the fact that proofs of problems that require exploring a single node (*e.g.*, retrieving an axiom from the database) without caching will still require an identical amount work even if a cache is in use; thus the plot goes through the origin as expected.

Figure 3.3 shows the search performance of the unbounded overhead success and failure caching system. Certain problems are helped (*i.e.*, fewer nodes are explored) by the presence of cache entries, and corresponding datapoints shift downwards since the cost of solving any given problem with the control system is invariant. Other problems are not affected by the presence of cache entries, so their respective datapoints remain unchanged.

Since the problems are presented in random order, linear regression — by minimizing the sum of the squares of the errors — provides a good estimate of the slope over the problem distribution as a whole. As the datapoints spread downwards, the regression slope decreases, reflecting the need to search fewer nodes on average over all problems in the population. The regression slope obtained here ($\log(b) = [0.796 \pm .015]$) implies that the system searches significantly fewer nodes than the breadth-first search control system, which would, by definition, yield a slope of exactly $\log(b) = 1$ when measured against itself. A similar analysis for the non-caching system (plot not shown) yields a one-parameter regression slope of $\log(b) = [1.033 \pm .004]$, indicating that the non-caching system explores a larger number of nodes than the control system. Again, this is to be expected, since unit-increment depth-first iterative deepening explores the space in precisely the same order as breadth-first search, but by performing iterative deepening will explore some nodes more than once.

[5] This last observation, of course, also implies that the computed regression parameters are not very meaningful here since they are computed using a regression model that assumes fixed c.

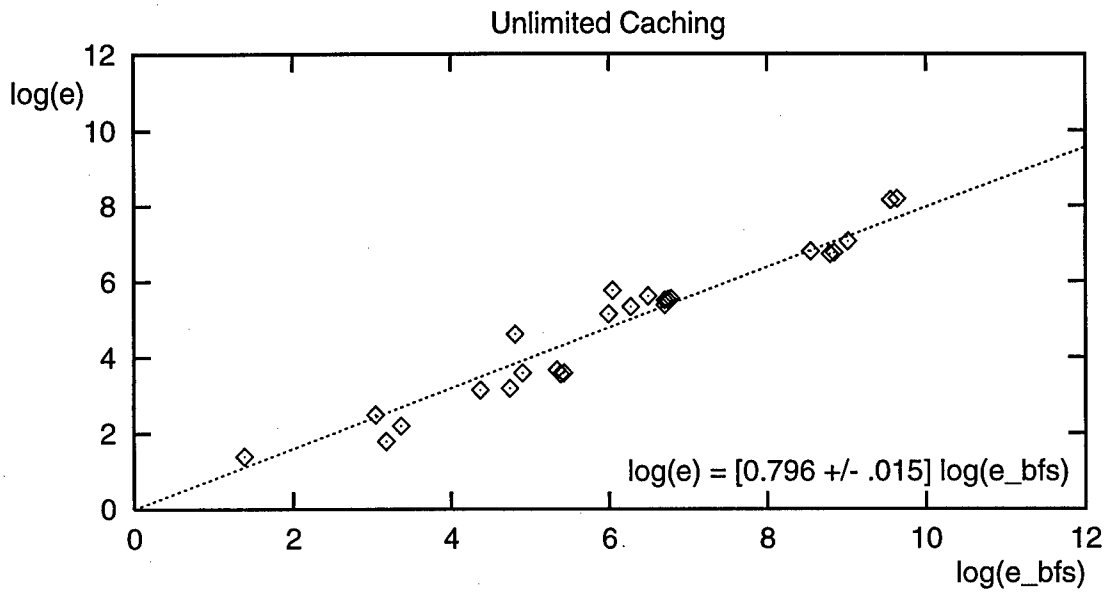

Figure 3.3: Search performance of an unlimited-size caching iterative-deepening theorem prover on the same situation-calculus problem set of Figure 3.1. Cache overhead effects are factored out.

Thus this simple situation-calculus domain serves as an example of an application where unlimited-size caching is inadequate. More precisely, while this kind of caching may reduce the number of nodes explored in search of a solution (as is evident from our analysis of the search effects), it causes an overall decrease in performance in this domain presumably due to increased overhead. The goal of bounded-overhead caching is to capture as much as possible of the beneficial search effect without incurring excessive node exploration costs.

3.3.3 Experiment 2

In this experiment, we evaluated the performance of bounded-overhead caches across a variety of cache sizes and management strategies. The configurations tested here were

- LRU replacement,
- LFU replacement,
- FIFO replacement, and
- RANDOM replacement.

These four systems were tested on all 26 problems using caches ranging from 10 to 1000 elements. As before, the problems were presented in the same random order for all trials; in addition, the caches are left undisturbed between problems.[6]

[6]The RANDOM cache management strategy involves selecting an arbitrary cache entry for replacement and therefore involves minimal overhead. FIFO maintains the cache entries as a queue, placing new entries at the end of the queue while deleting the first queue element, while LRU is implemented as a modification of FIFO where a cache hit causes the corresponding cache entry to move to the end of the queue. Both of these strategies also entail minimal overhead. More problematic is LFU, since a naive implementation would entail a substantially higher cache overhead than the other bounded-overhead strategies. Instead, a variant of LRU called a *creeping cache* is used to approximate

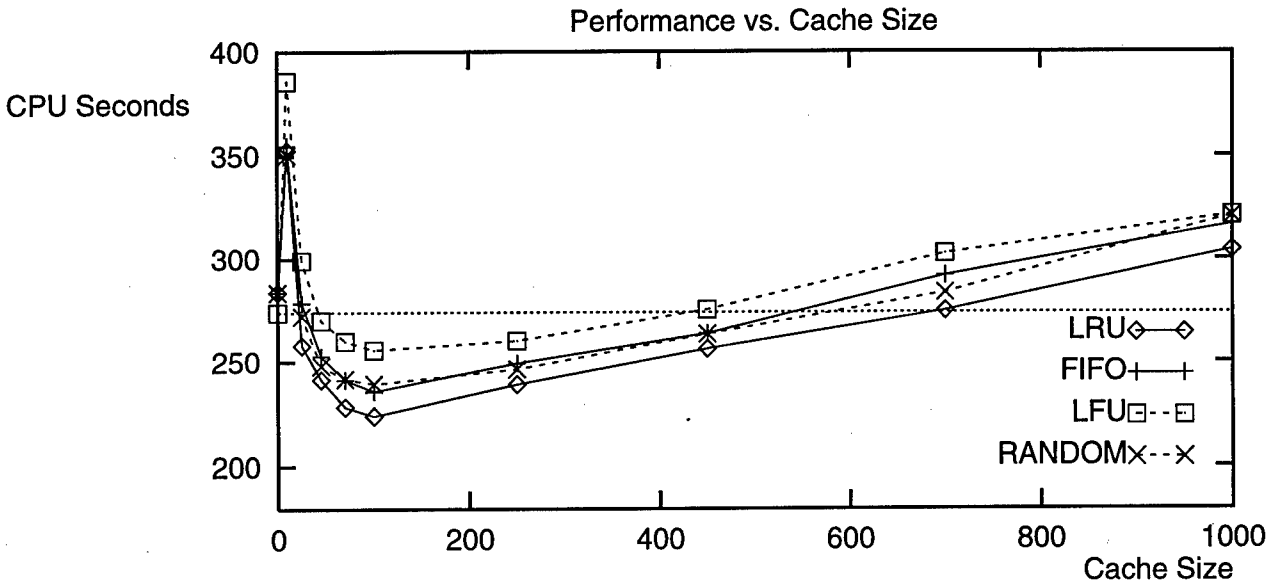

Figure 3.4: Performance of four bounded-overhead caching schemes as a function of cache size. Performance is measured in terms of cumulative CPU seconds to solve the same 26 situation-calculus problems used in Experiment 1. The horizontal line corresponds to the performance of the non-caching system (273.9 seconds); recall the unlimited-size caching system requires 669.8 seconds to solve all 26 problems.

The first question we would like to answer is whether or not a bounded-overhead caching system can outperform both the non-caching and the unlimited-size caching systems of the previous section. Since the resource limit given for each query was sufficient to solve every problem, as a first approximation we can simply plot cumulative solution times over the entire test suite.[7]

Figure 3.4 shows the results of this experiment. There are two observations that bear mentioning. First, for every cache management strategy tested, using a small cache initially causes an increase in time to solution. As the cache size is increased, performance improves and then degrades again. This behavior is consistent with our expectations; a very small cache bears much of the overhead costs yet yields little of the beneficial search effects. As the size grows larger, the beneficial effects of caching become evident but are eventually overwhelmed by increasing cache overhead. This analysis, of course, relies on a hidden, yet perhaps unwarranted, assumption that beneficial search effects increase monotonically with cache size. The second observation is that, while all four of the tested strategies behave in approximately the same fashion, LRU displays slightly better performance than the others. It is not possible to tell whether LRU's edge lies in lower cache overhead costs relative to the other strategies or in some increased beneficial search effect. The answer to this question hinges in part on implementation-specific aspects of the cache. In particular, while cache lookup costs are roughly equivalent for all implementations (modulo differences in actual cache contents, of course), the cost of maintaining the cache itself may differ

a real LFU policy while displaying exactly the same overhead characteristics as LRU. A creeping cache operates by demoting a corresponding cache entry pointer by one position in the queue for every cache hit. A frequently hit entry will thus trickle back to the end of the queue where it is unlikely to be replaced.

[7]Note that the use of cumulative CPU times does tend to skew the relative importance of individual problems by emphasizing the larger problems. While there might be other ways of presenting CPU performance data, the use of cumulative CPU times is simple, intuitive, and, most important, consistent with the literature.

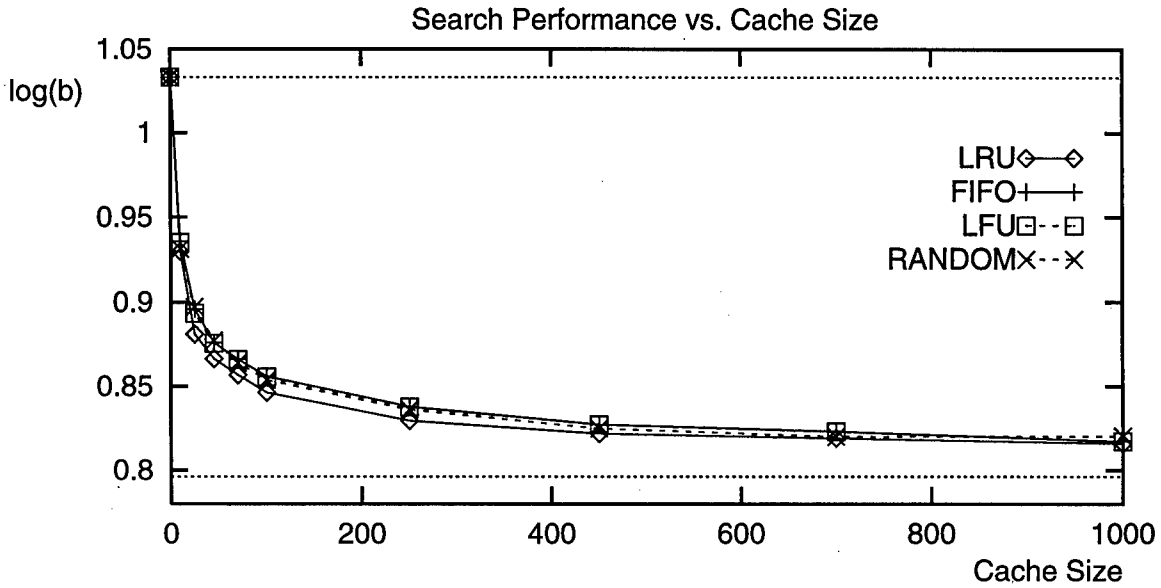

Figure 3.5: Search performance of a bounded-overhead caching iterative-deepening theorem prover using LRU, FIFO, LFU, and RANDOM cache management strategies. The graph plots the empirically obtained one-parameter regression slope $\log(b)$, an indicator of the search space size, against the size of success and failure caches. The horizontal lines correspond to the search performance of the non-caching system ($\log(b) = 1.033$) and the infinite-size caching system ($\log(b) = .796$).

from one strategy to the next.

We can check both of these informal analyses by once again factoring out the implementation-dependent cache overhead costs and focusing on the implementation-independent search effects. We would like to know, first, how the magnitude of the beneficial search effect changes with cache size, and, second, if the different caching strategies display substantially different beneficial search effects. We again use the one-parameter regression model of Equation 3.3 to factor cache overhead costs out of the analysis. In Figure 3.5, we plot the value of the regression parameter $\log(b)$ against the size of the cache used for all four bounded-overhead cache management strategies listed above. We expect that the smaller cache sizes will have empirically measured slopes very close to the value obtained for the non-caching system ($\log(b) = [1.033 \pm .004]$), while larger cache sizes should approach the slope obtained for the unlimited-size caching system ($\log(b) = [0.796 \pm .015]$).

We are now in a position to check the two observations cited earlier. First, we note that not only is the beneficial search effect due to caching increasing monotonically with cache size, but that most of this effect is evident even with relatively small caches. Second, we note that the search performance of different policies is relatively homogeneous, although LRU does show some slight edge for all cache sizes tested. This latter observation means that LRU's slight overall performance edge from Figure 3.4 is at least partially based on search effects rather than only differences in cache overhead.

Notwithstanding LRU's slight edge, the search performance over all four strategies is remarkably uniform. There are two alternative interpretations for this striking similarity in search performance. The first interpretation is that the subgoals explored by a theorem prover do not fit any regular pattern of exploration. If this is true, than a random replacement strategy will provide adequate cache entry replacement guidance. The second interpretation is that the exploration pattern of iterative deepening search strategies does exhibit some analogue to locality of reference, but that we are as

yet incapable of exploiting it because we simply do not understand it. This second interpretation leaves open the possibility that one might develop an analytic model of iterative-deepening exploration that will suggest an improved cache management strategy that would eventually outperform LRU. Selecting which of these two competing interpretations is correct is difficult. While we know that a hypothetical optimal caching system's performance should not exceed the performance of the unlimited caching system ($\log(b) = .796$) for this particular problem ordering, we do not really know how it compares with LRU for fixed-size caches.

When designing a hardware cache, one may resort to approximating the performance of an optimal or nearly optimal caching system by examining page-access traces collected during execution of some benchmark programs. Unfortunately, we cannot rely on this kind of static analysis to predict the performance of a fixed-size cache for theorem proving. The reason is that, unlike paged memory systems where the page access pattern is determined by the program being benchmarked, the pattern of cache accesses is not fixed, but rather changes depending on the cache contents. A cache hit (or lack thereof) changes the search behavior of the system; thus it is simply not possible to use static trace information from one cache configuration to predict system performance with a different cache configuration.

3.3.4 Experiment 3

In the previous experiment, we tested cache management policies suggested by analogy to hardware systems. In this section, we begin to explore alternative cache management strategies based on more theorem-proving specific models of cache entry utility. One would hope that these strategies might more adequately reflect the underlying iterative-deepening search process, resulting in better performance than simple LRU caching.

Traditional paged-memory hardware caching systems generally assume that the cost of a page replacement is independent of the page being replaced. Cache management policies such as LRU, LFU, and FIFO rely at least implicitly on this assumption; a decision to replace a cache element is made based only on its past usefulness rather than on any notion of its original cost. Our success and failure cache entries are not all of uniform cost.[8] In this experiment, we introduce and test two variants of the LRU cache management policy that do not assume all cache entries are of uniform cost.

The *cheapest least-recently used* policy (CLRU) selects for replacement the least-recently used cache entry whose solution cost (expressed in number of nodes explored) is exceeded by the new cache entry's solution cost. If no cache entry matching this criteria is found, the new entry is simply discarded. In a similar fashion, the *dearest least-recently used* policy (DLRU) looks for the least-recently used cache entry whose solution cost is larger than the new cache entry's solution cost. These two policies explore fundamentally different intuitions about which cache entries are more likely to be useful in solving future problems. CLRU looks for relatively infrequent cache hits that produce large savings, while DLRU strives for more frequent, less dramatic, cache hits.

While still qualifying as bounded-overhead caches, the CLRU and DLRU caches will carry higher cache overheads than the other caches described earlier; in a naive implementation, an unsuccessful cache insertion event may, in the worst case, take time proportional to the size of the cache. All other things being equal, even if more sophisticated implementations are available, the additional

[8] More recent work on caching systems for shared-memory non-uniform memory access (NUMA) machines must also take into account the differing latencies of local *vs.* remote data items. NUMA systems generally need only worry about two possible costs: cheaper local access as opposed to more expensive remote access. For inference engines, of course, success and failure cache entries are not of uniform cost and, unlike the binary NUMA model, may be arbitrarily expensive.

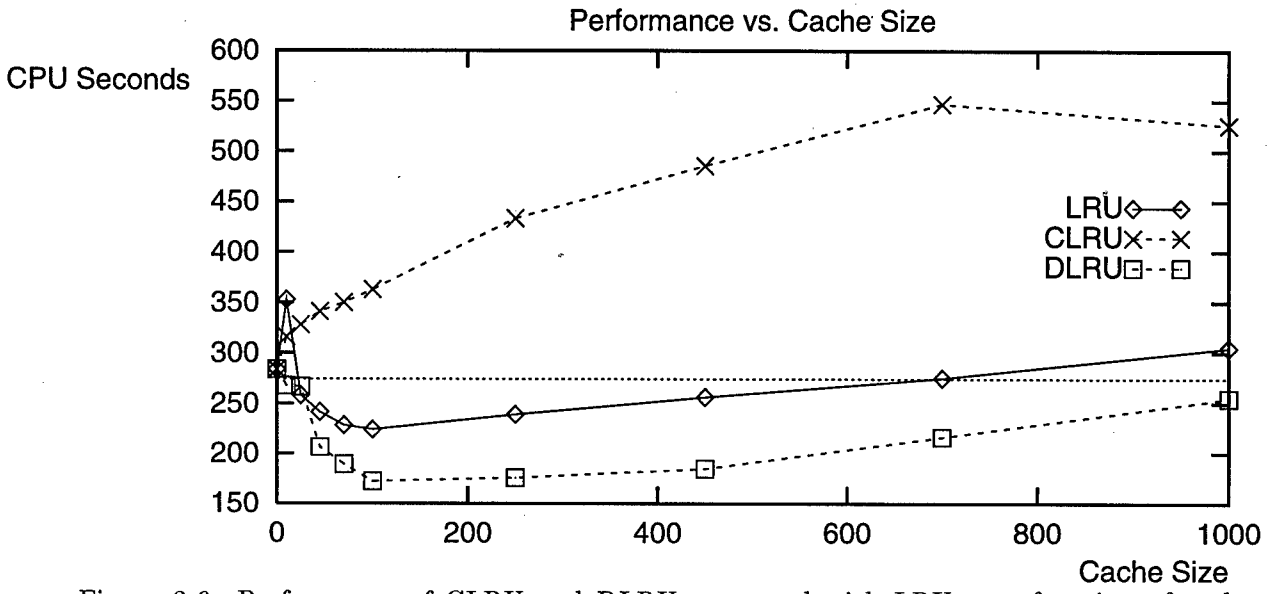

Figure 3.6: Performance of CLRU and DLRU compared with LRU as a function of cache size. Performance is measured in terms of cumulative CPU seconds to solve the same 26 situation-calculus problems used in Experiment 1. The horizontal line corresponds to the performance of the non-caching system (273.9 seconds); recall the unlimited-size caching system requires 669.8 seconds to solve all 26 problems. Note that the vertical scale is compressed with respect to Figure 3.4.

bookkeeping required will result in higher cache overheads than, for example, simple LRU.

Figure 3.6 plots the cumulative CPU time required to solve all 26 problems against cache size for CLRU, DLRU, and LRU. While DLRU significantly outperforms LRU, CLRU's performance is much worse than either of the other two strategies. In fact, even from a qualitative perspective, these three strategies display markedly different performance curves. Unlike LRU, DLRU provides an immediate gain in performance even for very small cache sizes; there is no initial degradation due to the extra cost of operating a cache followed by performance improvement as the beneficial search effects counteract the cache overhead. On the other hand, CLRU's performance degrades immediately and continues to get worse as the cache size increases.

Figure 3.7 plots the search performance of CLRU, DLRU, and LRU as a function of cache size. Given the respective performances of these policies shown in Figure 3.6, we would expect DLRU to explore the smallest space, followed by LRU and CLRU. While our expectations regarding the relative sizes of the spaces explored by CLRU and LRU are met, DLRU's performance advantage does not, surprisingly enough, seem based on a reduction in search space.

One explanation for CLRU's poor performance is that it might be possible to fully populate the cache with expensive — but useless — cache entries that are never replaced because they are more expensive than the entries being added. To test this hypothesis, we implemented a variant of CLRU called *probabilistic cheapest least-recently used* that allows a new entry to replace a more expensive existing cache entry with probability inversely proportional to cache size. In this fashion, a new cache entry always has at least a chance to replace an existing entry even if the cost-based replacement criteria are not strictly met. Testing of this variant policy supports our hypothesis, since the new search performance (not shown) was found to closely approximate the search performance of LRU.

The behavior of DLRU is somewhat more difficult to explain. On one hand, the overall perfor-

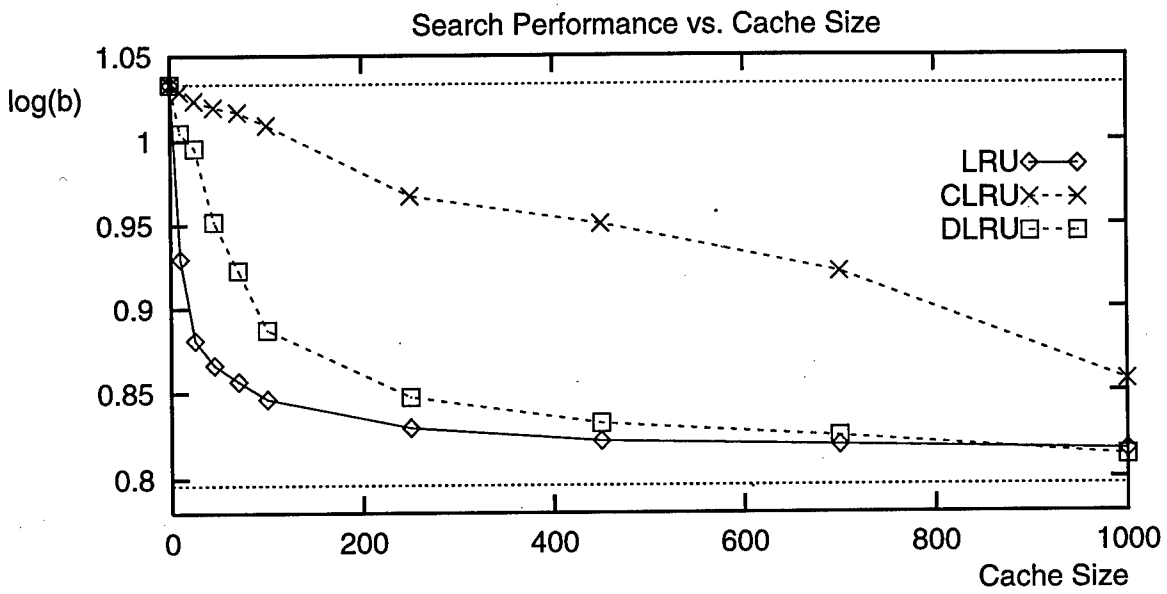

Figure 3.7: Search performance of CLRU and DLRU compared with LRU as a function of cache size. The graph plots the empirically obtained one-parameter regression slope $\log(b)$, an indicator of the search space size, against the size of success and failure caches. The horizontal lines correspond to the search performance of the non-caching system ($\log(b) = 1.033$) and the infinite-size caching system ($\log(b) = .796$).

mance (Figure 3.6) is better than LRU, but on the other hand, the search performance (Figure 3.7) is worse than LRU. These results imply that DLRU's performance advantage is based on lower cache overhead rather than reduced search. However, we also know that the implementations of DLRU and CLRU are identical, save for the sign of a single arithmetic comparison. Furthermore, we know that for identical cache contents both DLRU and CLRU carry, by design, cache overhead costs that dominate the overhead cost of LRU. Thus it is difficult to see how DLRU's average node expansion cost can be low enough to more than counteract the increase in nodes searched by DLRU with respect to LRU.

The clue to understanding this inconsistency lies in the relative proportion of success and failure entries within the caches. At the end of the problem suite, about 85% of the DLRU cache is devoted to failure entries, while LRU contains only about 40% failure entries and CLRU contains zero (or at most very few) failure entries. Given that we are performing iterative deepening, and that therefore many failures are due to encountering relatively small resource limits, we would expect that failures, on average, will be less costly than successes. Thus we would expect DLRU to populate its cache, on average, with a larger number of failure entries than CLRU.

This observation also helps to explain the measured difference in cache overhead between DLRU, CLRU, and LRU. Recall that we expected DLRU and CLRU to display identical overhead costs for identical cache contents. As the proportional differences in failure and success entries clearly shows, the cache contents are not identical. Thus if failure entries are inherently cheaper to maintain than success entries, we would expect that DLRU, on average, would display lower cache overhead costs than either LRU or CLRU based on the difference in relative proportion of failure to success entries. We explored this issue in the next experiment.

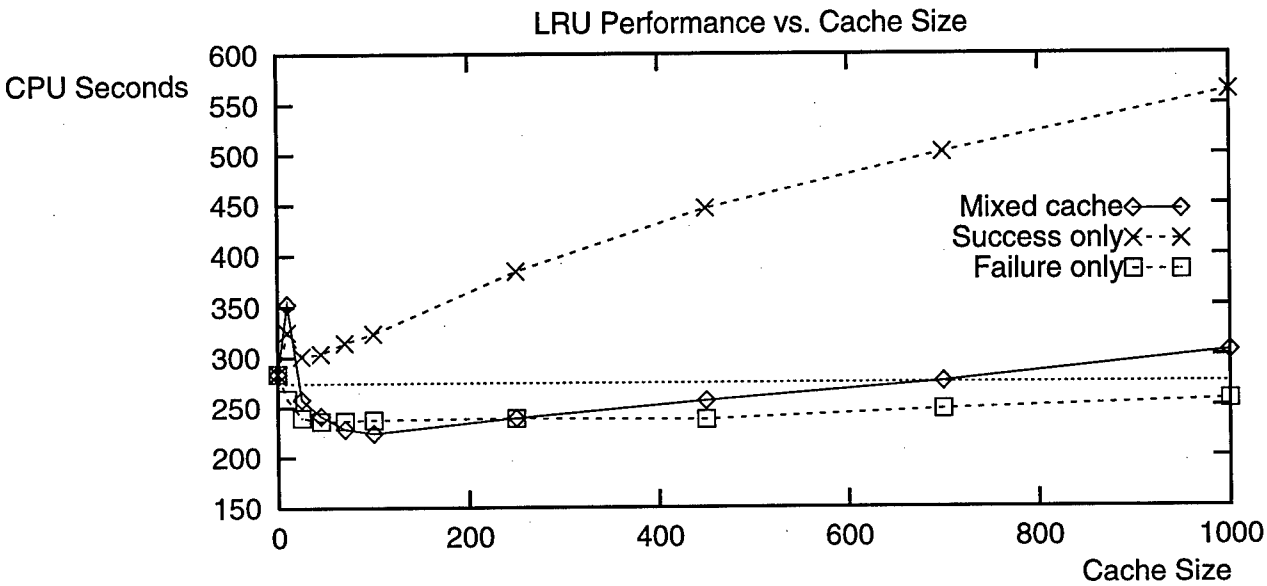

Figure 3.8: Performance of success-only and failure-only caching systems using an LRU replacement policy as a function of cache size. The mixed cache LRU system of Experiment 2 is included for comparison. Performance is measured in terms of cumulative CPU seconds; the horizontal line corresponds to the performance of the non-caching system (273.9 seconds). Recall that the unlimited-size caching system requires 669.8 seconds to solve all 26 problems.

3.3.5 Experiment 4

In this experiment we examined the respective contributions of success and failure cache entries. Recall that our caching system allows both types of cache entries to coexist in a single cache. Alternative implementations might maintain separate success and failure caches, or might perform only one kind of caching. Naturally, the relative worth of success and failure caching depends on the domain as well as the implementation, since different types of cache hits may entail a different magnitude of beneficial search effect, and failure and success cache overheads may also differ. In this experiment, we ran the same set of 26 blocks world problems in the same random order using both success-only caching and failure-only caching systems. Our intent was to measure the relative contributions of success and failure caching to reducing the search space, as well as to investigate possible implementation-dependent differences in cache overheads. As a basis for comparison, we also included the mixed-mode LRU caching scheme of Experiment 2.

Figure 3.8 compares the performance of success-only and failure-only caching with the mixed caching system used in the previous experiments. As we predicted, the failure-only system's performance curve matches qualitatively that of DLRU in the last experiment, while the success-only system's curve approximates that of the mixed LRU cache. To determine if the root cause is an actual difference in overhead for the two types of cache, Figure 3.9 plots the search performance of all three strategies. Given that the reductions in search space do not correspond with system performance plotted in Figure 3.8, we must conclude that per-node exploration costs are far from uniform for equivalent cache sizes. This conclusion is in fact easily confirmed via direct inspection of the data; for example, for 100 element caches, the mixed-mode cache explored a total of 18,815 nodes in 262.1 seconds (13.9 msec/node) over the entire test suite. The success-only system explored 41,120 nodes in 323.1 seconds (7.9 msec/node), while the failure-only system explored 41,549 nodes

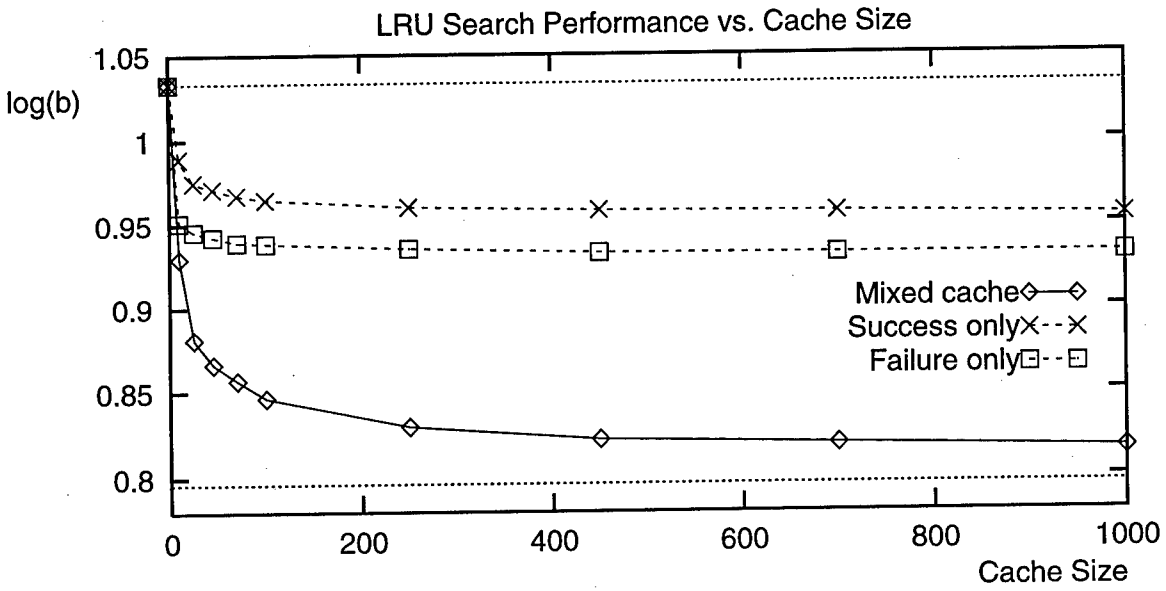

Figure 3.9: Search performance of success-only and failure-only caching systems using an LRU replacement policy as a function of cache size. The mixed cache LRU system of Experiment 2 is included for comparison. The graph plots the empirically obtained one-parameter regression slope $\log(b)$, an indicator of the search space size, against the size of success and failure caches. The horizontal lines correspond to the search performance of the non-caching system ($\log(b) = 1.033$) and the infinite-size caching system ($\log(b) = .796$).

in only 237.4 seconds (5.7 msec/node). How can we account for these fundamentally different node expansion costs? That such differences actually exist should not be surprising; success and failure entries are fundamentally different sorts of things. While the precise magnitude of the difference is undoubtedly specific to this implementation, any implementation would almost necessarily exhibit some difference in overhead cost for manipulating success and failure entries.

Given the relative node expansion costs, one might question the utility of caching success entries in this implementation. With the exception of 70 and 100 element caches, failure-only caches outperform mixed-mode caches for all other tested cache sizes (success-only caching performed poorly for all tested cache sizes). Even in the region where mixed-mode caching is faster than failure-only caching, the difference is not very large, and one might argue that the added performance does not warrant the additional implementation complexity. However, repeating this same experiment using a DLRU policy (the policy with the best measured performance in Experiment 2) supports quite a different conclusion.

Figure 3.10 plots the performance of the same three cache configurations as Figure 3.8, but with DLRU cache management as opposed to LRU cache management. In this plot the relative performance of the three configurations differs significantly from the relative performance of the LRU systems of Figure 3.8. Here, the mixed-mode cache system operating with the DLRU policy always outperforms the comparable failure-only system; the 100 element cache using a DLRU policy displays the best performance of any system tested in this chapter on this suite of problems. In addition, we note that the success-only system performs better than failure-only and the mixed-mode systems on very small cache sizes. We conclude that one should not discount the importance of success cache entries to the overall performance of the system.

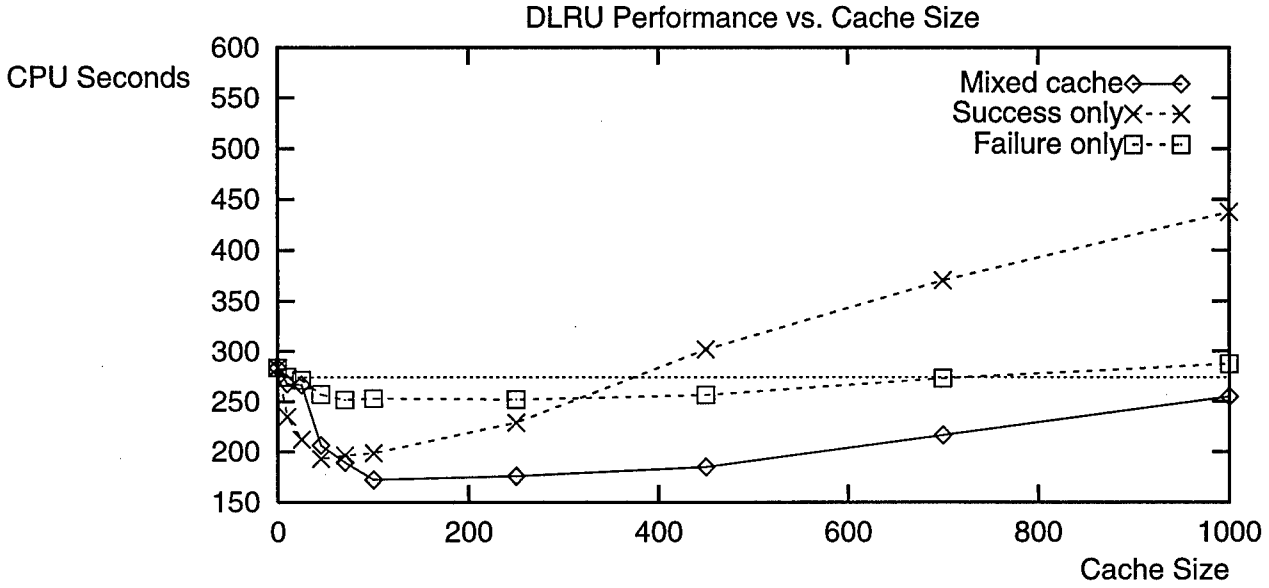

Figure 3.10: Performance of success-only and failure-only caching systems using a DLRU replacement policy as a function of cache size. The mixed cache DLRU system of Experiment 3 is included for comparison. Performance is measured in terms of cumulative CPU seconds; the horizontal line corresponds to the performance of the non-caching system (273.9 seconds). Recall the unlimited-size caching system requires 669.8 seconds to solve all 26 problems.

3.3.6 Experiment 5

Given that we have established the importance of both success and failure cache entries, we next turn our attention to the best relative proportion of these two types of cache entries. Should success and failure entries be managed separately in two smaller, separate, caches, or should they be allowed to intermingle in a single cache? If managed separately, what relative sizes should be chosen for the two caches?

In our previous tests using a mixed-mode DLRU cache (Experiment 3), we note that the success/failure ratio measured at the completion of the trial varied from 0/100 to 26/74 percentage of total cache size. For the smaller cache sizes (10 and 25 elements), no success entries were retained at all. The largest percentage of success entries (26%) occurred with a 250 element cache, and tapered off to 13% on the 1000 element cache trial. In the current experiment, we tested an alternative dual-cache implementation against the mixed-mode cache system. We ran four new DLRU trials for each cache size, fixing the ratios of success to failure cache sizes to 20/80, 40/60, 60/40, and 80/20 percentage of total cache size. We compared these results to the failure-only (*i.e.*, 0/100 success/failure ratio) and success-only (*i.e.*, 100/0 success/failure ratio) systems from Experiment 4 as well as the mixed-mode DLRU performance of Experiment 3.

Figure 3.11 plots the performance of all seven tested systems. This plot is consistent with several previously mentioned observations. First, it is clear that a mixture of success and failure entries generally outperforms a system that only performs one of success or failure caching (an exception is made for very small cache sizes, where success-only caching performs quite well). Second, we note that when the caches are managed separately, a larger proportion of failures to successes generally entails better performance. One might suspect that part of this effect may be due to the lower overhead costs associated with manipulating failure entries.

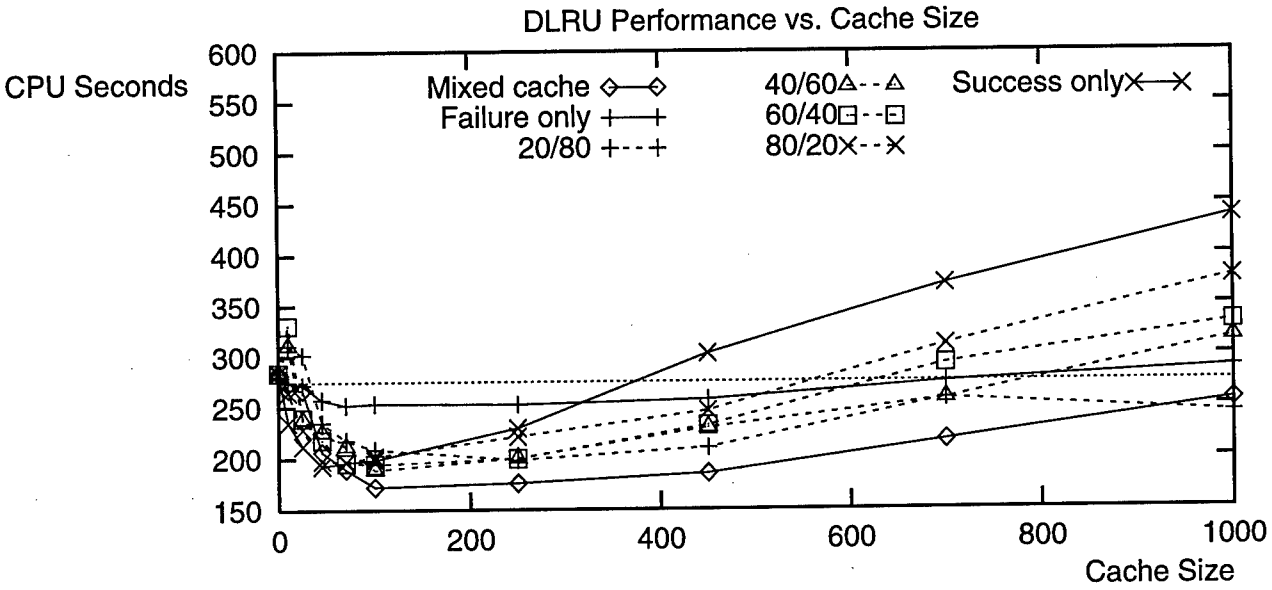

Figure 3.11: Performance of an assortment of dual-cache implementation trials compared to success-only, failure-only, and mixed-mode caching systems using a DLRU replacement policy as a function of cache size. Performance is measured in terms of cumulative CPU seconds; the horizontal line corresponds to the performance of the non-caching system (273.9 seconds); recall the unlimited-size caching system requires 669.8 seconds to solve all 26 problems.

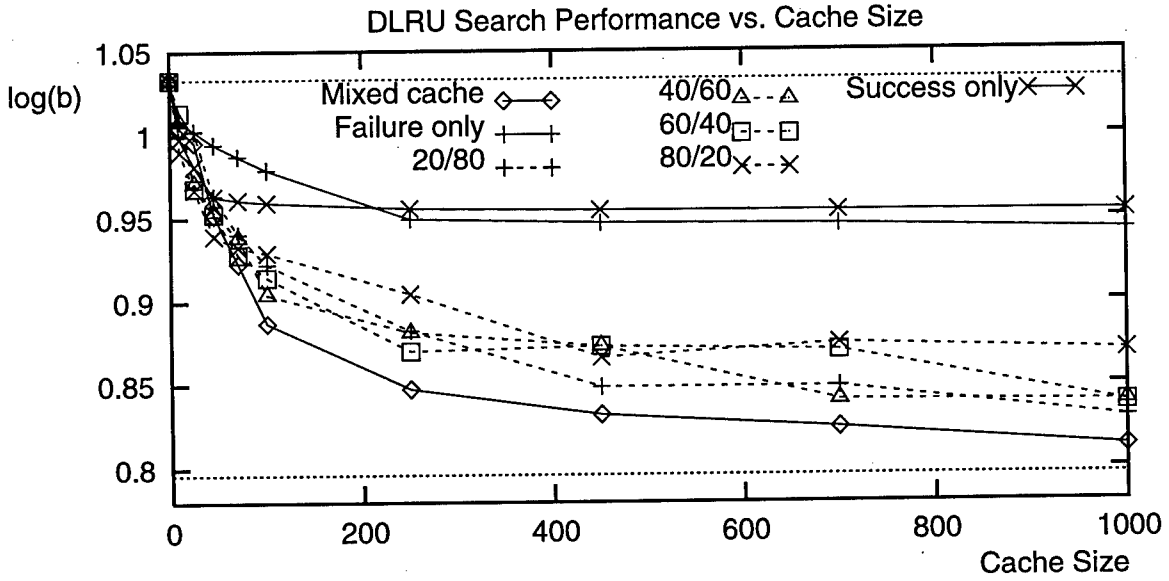

Figure 3.12: Search performance of a selection of dual-cache systems compared to success-only, failure-only, and mixed-mode caching systems using a DLRU replacement policy as a function of cache size. The graph plots the empirically obtained one-parameter regression slope $\log(b)$, an indicator of the search space size, against the size of success and failure caches. The horizontal lines correspond to the search performance of the non-caching system ($\log(b) = 1.033$) and the infinite-size caching system ($\log(b) = .796$).

Figure 3.12 shows the search performance of the same seven systems. We note that all of the fixed-proportion systems produce roughly the same amount of search reduction; thus we conclude that the increased performance observed with a larger proportion of failures is probably due to differing relative overhead costs between success and failure entries. On the other hand, it is equally clear that the dynamically-managed mixed-mode cache gets at least some of its performance advantage from actual reductions in search space rather than simply differences in relative cache overhead. It would certainly appear — at least for this cache management strategy and test domain — that forcing success and failure entries to coexist and fight for survival on a uniform basis is the best policy over a broad range of cache sizes, resulting in greater search reduction and better overall performance.

3.3.7 Experiment 6

In this experiment, we examined the effect of redundant cache entries on the performance of the system. A redundant entry is an entry that is either identical to or subsumed by a different cache entry. Redundant cache entries arise due to the imposition of cache hit generality constraints and also as a result of iterative deepening and failure caching. They reduce the performance improvements obtained with bounded-overhead caching by occupying a portion of the cache with redundant — and therefore useless — information. In addition, the presence of redundant cache entries may interfere with the cache replacement policy; if multiple entries exist for a given query, the usefulness of each of the entries may seem artificially low.

To see how multiple entries can arise, consider a subgoal $q(?x)$ where candidate success entries $q(a)$, $q(b)$, and $q(c)$ are already present in the cache. Since the cache entries are less general than the subgoal, these entries are not allowed to cause a cache hit. If the theorem prover eventually solves the $q(?x)$ subgoal while binding $?x$ to a, a new success entry $q(a)$ is added to the cache, which already contains a copy of this entry. Alternatively, if the theorem prover manages to solve the $q(?x)$ subgoal in its most general form, the new success cache entry $q(?x)$ renders the existing entries $q(a)$, $q(b)$, and $q(c)$ obsolete.

A second source of redundant cache entries is the natural interplay between iterative deepening and resource-limited failure caching. Consider a subgoal $q(a)$ that almost matches a candidate failure entry $q(a)$ in the cache, where the problem is that the resource annotation on the cache entry is smaller than the resources currently available for proving the subgoal. If the prover fails to prove $q(a)$ within the larger current resource limit, we can simply update the resource annotation on the original failure entry (if the original failure entry is still in the cache at failure time). Alternatively, we can simply add a new failure entry with the larger resource annotation and trust the cache replacement policy to discard the other, less general, entry at some time in the future.

There are two approaches for dealing with this problem. The first approach is to check for redundant entries whenever a new cache entry is made, at some additional overhead cost.[9] The extra overhead may be more than outweighed by increased cache efficiency. For example, a redundancy-free infinite size caching system requires 480.7 seconds to solve all 26 problems, leaving 4,216 entries in the cache, 697 of which provided cache hits at some point during the trial. When compared with the 10,753 entries and 669.8 seconds required by the standard infinite size caching system, it is clear that the extra redundancy check pays off. Note, however, that even with redundancy checking the infinite size cache does not achieve the level of performance obtained by some of the bounded-overhead cache implementations.

[9]Sophisticated indexing techniques may allow the redundancy check to occur as a side-effect of cache insertion. Nevertheless, while the magnitude of the additional overhead may be limited, some amount of additional overhead is inevitable.

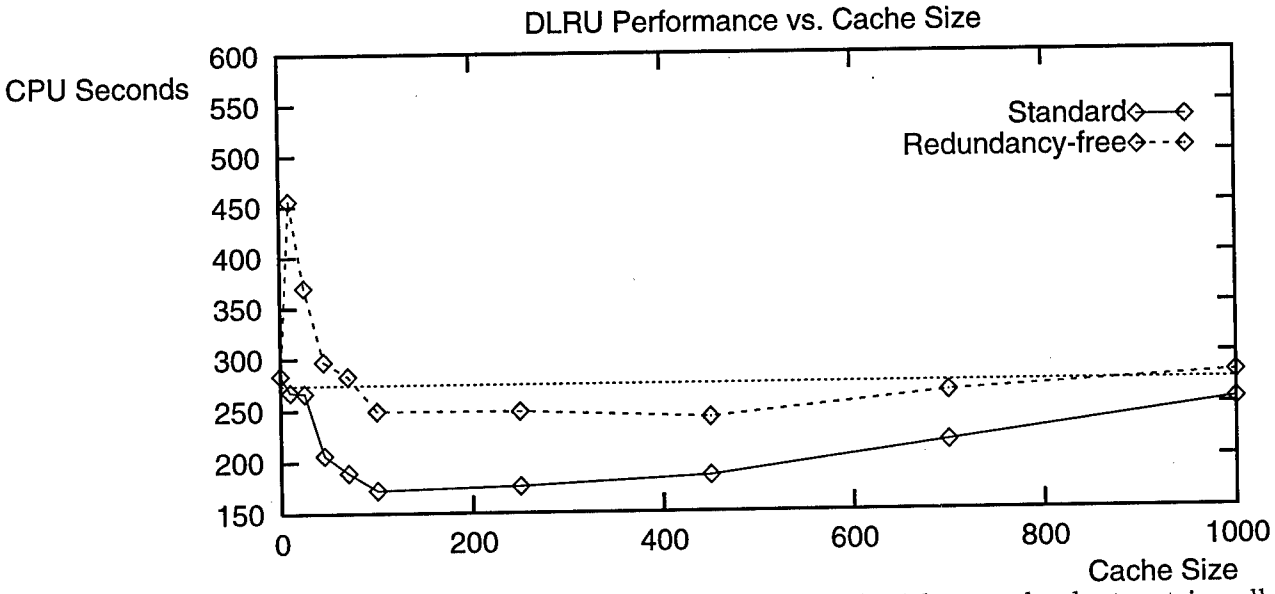

Figure 3.13: Performance of DLRU caching both with and without redundant entries allowed as a function of cache size. As in previous experiments, the horizontal line corresponds to the performance of the non-caching system (273.9 seconds); recall the unlimited-size caching system requires 669.8 seconds to solve all 26 problems.

The second approach is particular to fixed size caches. The idea is to ignore the problem and trust the cache management policy to eventually reclaim space allocated to redundant entries. This approach requires a cache retrieval algorithm that guarantees the same entry is retrieved on identical successive queries regardless of any redundant entries that may be lurking within the cache. Management policies such as LRU that are based on the notion of recency have this property; the RANDOM cache replacement policy does not. Note that this approach requires no additional overhead: we simply let the cache take care of itself.

In this experiment, we again used the same set of 26 situation-calculus problems used in the previous experiments. A version of the mixed-mode DLRU caching system was altered so that an extra cache lookup is performed at cache insertion time in order to check for redundant cache entries. We compared this system with the same mixed-mode DLRU caching system of Experiment 3. The standard DLRU caching system sorts candidate entries so that older entries are preferred over newer ones, ensuring that redundant entries are never responsible for cache hits.

Figure 3.13 plots the performance of the two tested systems. Clearly, the additional overhead required to censor redundant entries overwhelms any added search benefit. This is strictly an implementation-dependent result: different implementations will have different overhead characteristics and thus may produce different overall performance. We can, however, obtain some estimate of how much search reduction benefit can be expected when censoring redundant entries. Figure 3.14 plots the search performance of the two systems. As expected, the standard system requires a larger cache to attain the same search performance as the redundancy-free system, although the search performance advantage of the redundancy-free system does not appear to be terribly large. Of course, the decision to censor redundant entries can only be made in an implementation-specific manner by fully investigating the cache overhead/search performance tradeoff for a particular system. Implementing a more efficient scheme for censoring redundant entries that reduces the associated overhead will naturally tilt this decision in favor of censoring

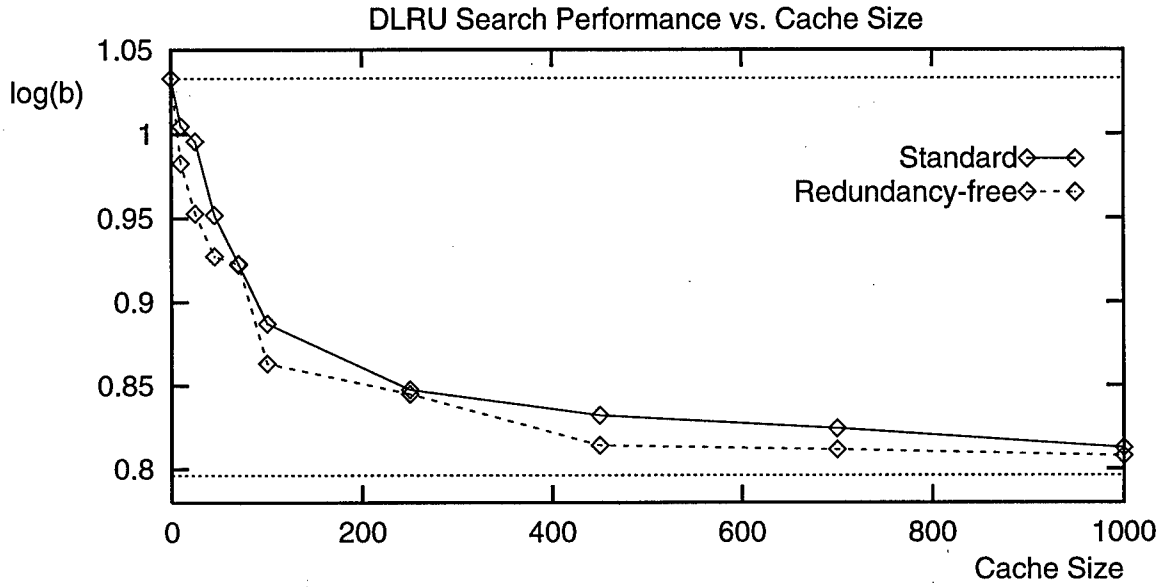

Figure 3.14: Search performance of DLRU caching both with and without redundant entries allowed as a function of cache size. The graph plots the empirically obtained one-parameter regression slope $\log(b)$, an indicator of the search space size, against the size of success and failure caches. The horizontal lines correspond to the search performance of the non-caching system ($\log(b) = 1.033$) and the infinite-size caching system ($\log(b) = .796$).

redundant entries.

3.4 Summary of Bounded-Overhead Caching

Our results have shown that bounded-overhead caching can be beneficial for definite-clause theorem proving systems. We have justified the use of such caches on the basis of a particular application context: the use of a theorem prover to solve many related problems drawn from a single problem distribution. We believe this application context is a realistic one for many real applications in artificial intelligence, deductive retrieval, and logic programming, and have shown how the traditional approach to caching for theorem proving, that is, the use of unlimited-size caches, is inappropriate in one instance of this general application context. Based on our experimental study, we have proposed a new bounded-overhead cache management policy we call *dearest least-recently used*, and have shown how this policy outperforms other, perhaps more obvious, cache management policies in at least one implementation and in one test domain. In summary, for this particular application context and this particular theorem prover, a 100-element DLRU mixed mode success/failure cache provides the best overall performance as measured by smallest total CPU time to solve all the problems in the test suite.

How well do these empirical results scale? There are three aspects to this important question. First, one might ask whether the results obtained with these particular theorem prover and cache implementations are indicative of results obtained with other implementations. We have been quite careful to distinguish between implementation-dependent results (such as the cumulative CPU curves of Figures 3.1, 3.2, 3.4, 3.6, 3.8, 3.10, 3.11, and 3.13) and implementation-independent results (such as the search performance curves of Figures 3.3, 3.5, 3.7, 3.9, 3.12, and 3.14). Thus

some results, such as the recommendation to forego redundancy-free caching in Experiment 6, depend on aspects of the implementation: in this case, the exact tradeoff between the added cache overhead and the extra search performance edge due to redundancy-free caching. In a similar fashion, the choice of cache size and management policy for best performance are implementation-dependent results. Other results, such as the relative search performance of the different caching strategies, are independent of the implementation altogether.

A second concern is whether the results scale from small problems to larger problems within the same domain. Our experiments give better reason to believe the results scale across problem size than do most other experiments. Because of the experimental methodology employed, we can extrapolate from small problems to large with the full faith we have in the underlying model (a model that just acknowledges that search cost grows exponentially with problem difficulty).

Finally, one might ask if the results obtained in this problem domain can be expected to generalize to other problem domains. Unfortunately, until an underlying model of domain theories is discovered that supports extrapolation across domains (like our model of theorem proving supports extrapolation over size), any results remain strictly domain-dependent. Thus, one should take these results as indicative of what can be achieved rather than a promise of what will be achieved.

Chapter 4

Explanation-Based Learning

This chapter surveys our work on adaptive inference and reports on the experiments we have performed. In particular, it reports on our work with bounded-overhead caching for definite-clause theorem provers and with the EBL family of explanation-based learning algorithms. We cover the integration of multiple speedup techniques and discuss appropriate configurations of cache and explanation-based learning. Finally, we describe a particular adaptive inference engine that forms the core of the ALPS system.*[1]

4.1 Introduction

In this section, we examine a second speedup technique, *explanation-based learning* (EBL). As we have seen in Section 3, perhaps the simplest way to increase the performance of a resource-limited problem solver is to cache $f(root(\rho)) = q\theta$ from each proof ρ of each successful problem-solving episode as a new fact in the domain theory.[2] Unfortunately, this kind of rote learning is overly constraining, in the sense that there may exist another form of the query, provable with the same pattern of reasoning implicit in the proof of the current example, which will not match the cached entry.

Much more desirable, therefore, is some mechanism by which the chain of logical reasoning used in the proof can be generalized — so as to be more useful — and then retained and reused. This is the essence of EBL: we operate on the structure supporting $root(\rho)$, that is, the subtree rooted at $m(root(\rho))$, in order to generalize it in some validity-preserving manner, and then extract (or *chunk*) a new, more general rule (called a *macro-operator*) to extend the domain theory.

Definition 6 : A generic EBL algorithm *generic-ebl* has the form

```
function generic-ebl(ρ:proof):rule;
  begin
    transform(ρ);
    return chunk(ρ);
  end
```

where ρ is the original proof, and $transform(\rho)$ leaves the proof ρ in a valid state.

What transformations are required to generalize a proof? Typical transformations performed by EBL algorithms involve pruning away portions of the proof tree. If some of the leaves of a proof tree

[1] This chapter is adapted from [90].

[2] See Chapter 2.2 for a description of the notation used.

are unmatched subgoal nodes (*i.e.*, $m(n_s) = \emptyset$) then we say the proof is a *partial proof*. Partial proofs can still be valid; a valid partial proof tree is a demonstration that $f(root(\rho)) = f(m(root(\rho)))$ is implied by the conjunction of its *premises*, or unmatched leaf subgoals. The *chunk* function creates a macro-operator that summarizes the logical argument supporting ρ, thus making this relationship between the premises and $m(\rho)$ explicit.

function *chunk*(ρ : *proof*) : *rule*;
  begin
    return $\left\{ f(m(\text{root}(p))) \leftarrow \bigwedge_{n \in premises(m(\text{root}(p)))} f(n) \right\}$;
  end

function *premises*(n : *node*) : *set of node*;
  begin
    if *consequent-node?*(n) **then return** $\bigcup_{s \in r(n)} premises(s)$;
    elseif *subgoal-node?*(n) $\wedge$ $m(n) = \emptyset$ **then return** $\{n\}$;
    else return *premises*($m(n)$);
  end

Of course, if there are no unmatched leaf subgoals in ρ, then $premises(m(root(\rho))) = \emptyset$, and the macro-operator obtained will have no antecedents. Hence rote learning is a special case of *generic-ebl*.

function *rote*(ρ : *proof*) : *rule*;
  begin
    return *chunk*(ρ);
  end

For the proof of Figure 2.1, this procedure would produce the new macro-operator

$$s(h(A)) \leftarrow .$$

Logically, this procedure is equivalent to simply adding the new fact $s(h(A)) = f(m(root(\rho))) = f(root(\rho)) = q\theta$ to the domain theory.

4.2 A Reconstruction of Traditional EBL

Given the relationship between EBL and rote learning just described, it is clear that the added power of EBL comes from the proof transformations applied before chunking. Traditional EBL algorithms, such as the EBG [71] and EGGS algorithms [73], generalize explanations by pruning portions of the proof, leaving some number of subgoals unmatched. This effectively relaxes the constraints once imposed by the pruned portions of the proof: once the proof's validity is restored, a macro-operator can be constructed that summarizes the general version of the logical argument used in the original proof. The macro-operator is added to the original domain theory with the provision that, where applicable, it takes precedence over other rules. The addition of the macro-operator will not change the deductive closure of a domain theory, although it may well have a significant effect on the future efficiency of the prover.

A traditional EBL algorithm can be reconstructed as a structured application of the following three basic proof transformation operators:

Definition 7 : Operator 1 (*Specialization*). Given a node n and a new expression α that is a substitution instance of the node formula, replace the node formula with the new expression:

$$Op1(n, \alpha) : \textbf{if } \alpha \subseteq f(n) \textbf{ then } f(n) \Leftarrow \alpha.$$

Definition 8 : Operator 2 (*Generalization*). Given a node n and a new expression α that is both a substitution instance of the node label and at least as general as the node formula, replace the node formula with the new expression:

$$Op2(n, \alpha) : \textbf{if } f(n) \subseteq \alpha \subseteq l(n) \textbf{ then } f(n) \Leftarrow \alpha.$$

Definition 9 : Operator 3 (*Match Edge Pruning*). Given a subgoal node n_s, delete the entire subtree below it:

$$Op3(n_s) : m(n_s) \Leftarrow \emptyset.$$

We can use these three operators to reconstruct a traditional EBL algorithm. As noted earlier, the general idea is to first prune away a portion of the proof, leaving some number of unmatched subgoal nodes, then maximally generalize the proof while still guaranteeing its validity, and finally extract a new macro-operator using the previously introduced function *chunk*:

```
function ebl(ρ : proof) : rule;
  begin
    trim(root(ρ));
    lift(root(ρ));
    return chunk(ρ);
  end
```

where *trim* and *lift* together constitute the proof transformation step. Traditional EBL algorithms differ in the exact criteria used to prune the proof (*i.e.*, *trim*) as well as the process used to restore the validity of the proof (*i.e.*, *lift*). The amount of pruning they perform crucially affects the future usefulness of the rule that can be learned from an explanation. This quality of usefulness is traditionally called *operationality* [6, 48, 55, 74, 75, 91]. It is impossible to determine in isolation whether a new rule will be useful: a formal measure of operationality, in the sense of guaranteeing improved problem-solving performance, has to take into account the distribution of future queries as well as what other rules are present in the domain theory.

For this reason, EBL systems have in the past relied on various *operationality heuristics* to guide the pruning process. Perhaps the simplest such heuristic is to flag some predicates as operational *a priori*, as in [71]. This approach is not always adequate [32], but it does have the advantage of being explicit. More sophisticated applications of EBL often have more sophisticated notions of operationality. For example, the ARMS system [92, 93] uses syntactic heuristics keyed on the structure of an explanation to determine where to prune. Other operationality heuristics that depend on the semantics of the explanation might also be used; unfortunately, such heuristics are often buried deep within a system, and thus they are often not rendered explicit.

Our reconstruction of traditional EBL relies on a very simple operationality heuristic. Suppose a subgoal is matched to a leaf consequent node, that is, a domain theory fact. It is reasonable to assume that a different version of the subgoal, perhaps with some alternative set of bindings, could also be proven by retrieving the same or a different domain theory fact.

```
procedure trim(n : node);
  begin
    if consequent-node?(n) then for s ∈ r(n) do trim(s);
    elseif subgoal-node?(n) ∧ r(m(n)) = ∅ then Op3(n);
    else trim(m(n));
  end
```

The condition $r(m(n)) = \emptyset$ in the fourth line of procedure *trim* is the same operationality criterion used implicitly in the EGGS algorithm as well as in [35]. This procedure simply strips all reference to specific domain theory facts used in the construction of the original proof. Once these axioms are removed, the remaining proof is still valid, since Operator 3 does not affect any of the validity conditions of Definition 5. However, the resulting proof is typically overly constrained, since the binding constraints that were imposed on the proof by the deleted leaf consequent nodes are still implicit in the proof node's formulae. The next step, then, is to "lift" the proof, producing a maximally general yet still valid partial proof structure.

```
procedure lift(n : node);
  begin
    relax-bindings(n);
    apply-bindings(n, collect-bindings(n, ∅));
  end
```

First, we relax all of the binding constraints using Operator 2 by replacing each element of the proof with its corresponding element from the original domain theory, which is available as the node label:

```
procedure relax-bindings(n : node);
  begin
    Op2(n, l(n));
    if consequent-node?(n) then for s ∈ r(n) do relax-bindings(s);
    elseif subgoal-node?(n) ∧ m(n) ≠ ∅ then relax-bindings(m(n));
  end
```

The resulting partial proof no longer contains reference to the constraints implicit in the pruned consequent nodes, but in general it violates the second validity condition of Definition 5. Next, we extract those binding constraints necessary to restore the validity of the proof (*i.e.*, the bindings required to unify the formula and label fields of each node along with the bindings required to enforce unification across proof match edges) and apply them uniformly throughout the entire proof.

```
function collect-bindings(n : node, θ : substitution) : substitution;
  begin
    θ ⇐ unify(l(n), f(n), θ);
    if consequent-node?(n) then for s ∈ r(n) do θ ⇐ collect-bindings(s, θ);
    elseif subgoal-node?(n) ∧ m(n) ≠ ∅
      then θ ⇐ collect-bindings(m(n), unify(f(n), f(m(n)), θ);
  return θ;
```

end

Here the function *unify*(x, y, θ) returns the substitution θ' that is the result of merging θ with the most general unifier of x and y.

Applying the binding constraints just collected requires a simple recursive descent algorithm that uses Operator 1 to specialize each node.

procedure *apply-bindings*$(n : node, \theta : substitution)$;
  begin
    $Op1(n, f(n)\theta)$;
    if *consequent-node?*(n) **then for** $s \in r(n)$ **do** *apply-bindings*(s, θ);
    elseif *subgoal-node?*$(n) \wedge m(n) \neq \emptyset$ **then** *apply-bindings*$(m(n), \theta)$;
end

Once the process terminates, we have restored the violated validity conditions, ensuring that the resulting proof is once again valid. Figure 4.1 shows what remains of the sample proof of Figure 2.1 once the process is complete.

We are now ready to extract a new macro-operator using function *chunk*. For the proof in Figure 4.1, this produces the new macro-operator:

$$s(?a) \leftarrow q(?d) \wedge p(?g) \wedge j(?g) \wedge k(?a) \wedge n(?a)$$

which is considerably more general than the macro-operator obtained by rote learning.

4.3 The Utility Problem

Once a new macro-operator has been added to the domain theory, the hope is that when a future query requires a similar proof structure, this will be found more quickly thanks to the presence of the acquired macro-operator. If the distribution of future problems is favorable, then the prover should exhibit better overall (*i.e.*, faster) performance. It may even solve additional problems that were previously unsolvable within a fixed resource bound. Unfortunately, the effect of EBL may actually be to slow down the prover. This undesirable effect has been dubbed the *utility problem* [36, 70].

To see how this can happen, consider the example from the previous section. The acquired macro-operator is intended to accelerate search by providing an alternative, shorter, path to the solution within the original search space defined by the domain theory. However, if this macro-operator does not lead to a solution for a particular problem, it just defines a redundant path in the search space, and using it causes a region of the search space to be searched again in vain.

While it is impossible to avoid the utility problem altogether, it is possible to minimize its impact. This goal is achieved by reducing both the frequency of inappropriate uses of acquired macro-operators (*i.e.*, uses that do not lead to a solution) as well as the cost incurred when an inappropriate use occurs. For example, all else being equal, the overhead of exploring an inappropriate macro-operator typically grows with the number of antecedents in that macro-operator. So for the example above, it is preferable to learn the equally valid (and equally general) macro-operator:

$$s(?a) \leftarrow k(?a) \wedge n(?a).$$

rather than the macro-operator learned by the traditional EBL algorithm, because this macro-operator will entail a smaller performance penalty when used inappropriately thanks to its smaller number of antecedents.

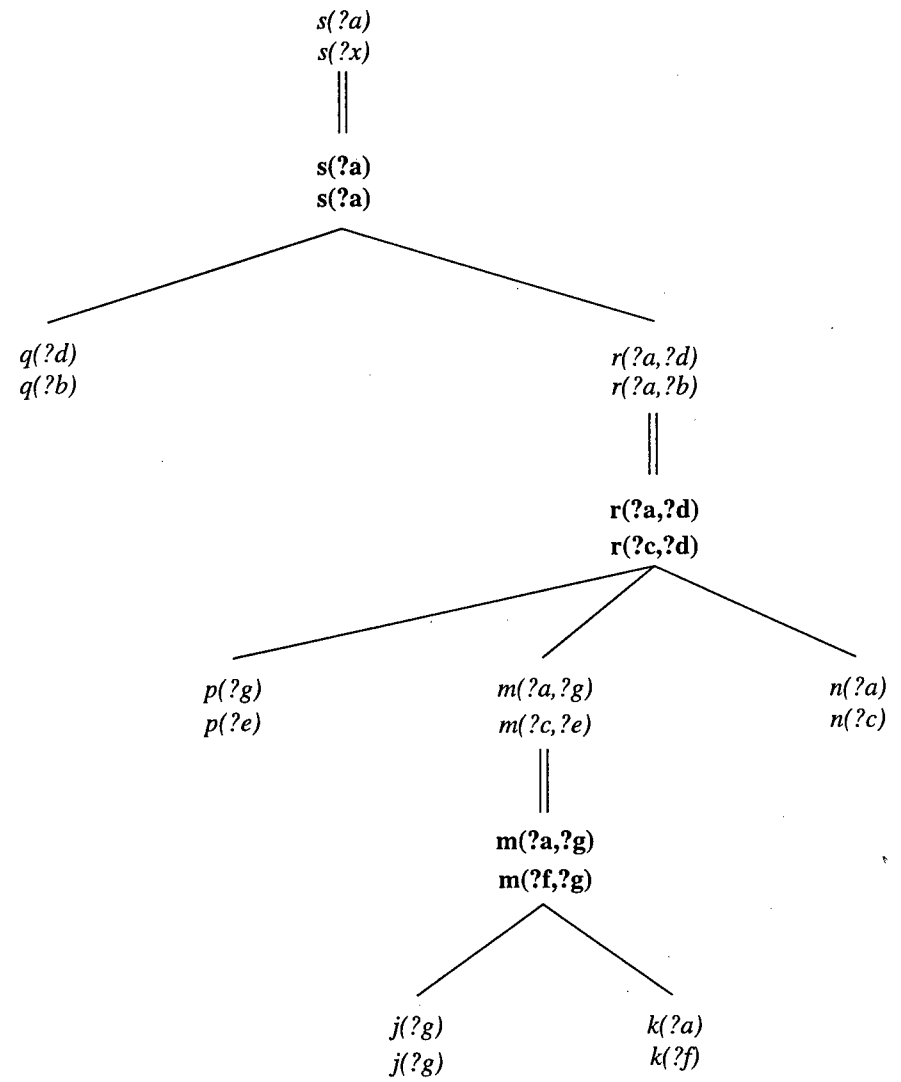

Figure 4.1: Sample proof of Figure 2.1 after applying procedure *ebl*. The new macro-operator $s(?a) \leftarrow q(?d) \wedge p(?g) \wedge j(?g) \wedge k(?a) \wedge n(?a)$ is produced by *chunk* from this transformed proof.

In order to reduce the frequency of inappropriate uses, one might even prefer to learn a more specific version of the macro-operator, such as:

$$s(h(?w)) \leftarrow n(h(?w)).$$

This last macro-operator may not be as useful as the previous one, since its less-general consequent expression means it will be applicable in fewer situations. For this same reason, however, it may less often be used inappropriately. Even when used inappropriately, it will entail a smaller performance penalty, because its single antecedent is a more-specific and therefore easier to prove version of one of the two antecedents of the previous macro-operator.

An example of where the utility problem is serious is the propositional calculus domain of *Principia Mathematica* [115] used by Newell, Shaw and Simon in their landmark work on the *Logic Theorist* (LT) [77], as adapted for definite-clause theorem provers by [78] and later [72]. The traditional EBL algorithm of Section 4.2 performs poorly in the LT domain. In this domain, an EBL algorithm should acquire new macro-operators of a very specific type. For example, from a proof of $thm(or(P, not(P)))$, we want to learn

$$thm(or(?x, not(?x))) \leftarrow .$$

Given the absence of antecedents, this is essentially a new domain theory fact. We call this type of macro-operator a *generalized cache entry* by virtue of its similarity with success caching (*i.e.*, rote learning), which, for this example, would acquire the strictly more-specific (and therefore less useful) entry:

$$thm(or(P, not(P))) \leftarrow .$$

Generalized cache entries minimize the utility problem: the only overhead in using a generalized cache entry is the added cost of indexing the entry. Yet no existing general-purpose EBL algorithm is capable of recognizing special situations where generalized caching is appropriate.[3]

4.4 Learning from Determinations

A second problem inherent in traditional EBL algorithms can best be introduced by an example. Suppose an artificially intelligent accountant knows that if two stores are located in the same state, then the sales tax rate at both stores must be the same:

$$rate(?y, ?r) \leftarrow state(?x, ?u) \wedge state(?y, ?u) \wedge rate(?x, ?r).$$

Given the common location of *Gucci* and *Cartier* and the sales tax rate at *Gucci*, one can find the rate at *Cartier* as the proof tree in Figure 4.2 shows.

A useful special-purpose version of the original rule:

$$rate(?x, 7\%) \leftarrow state(?x, NY)$$

states that the sales tax rate at any store in New York is seven percent. This new rule not only follows deductively from the original domain theory, but is also a useful rule in practice: subsequent queries referring to New York state stores can be handled more efficiently using the new rule than the original domain theory rule. A rule of the form "stores in the same state pay sales tax at

[3]The term *generalized caching* is used informally in [35] as a synonym for explanation-based learning. Here, by analogy with subgoal caching, we use the term in a more restricted sense to mean only those situations where the acquired macro-operator is a more general version of the original query expression.

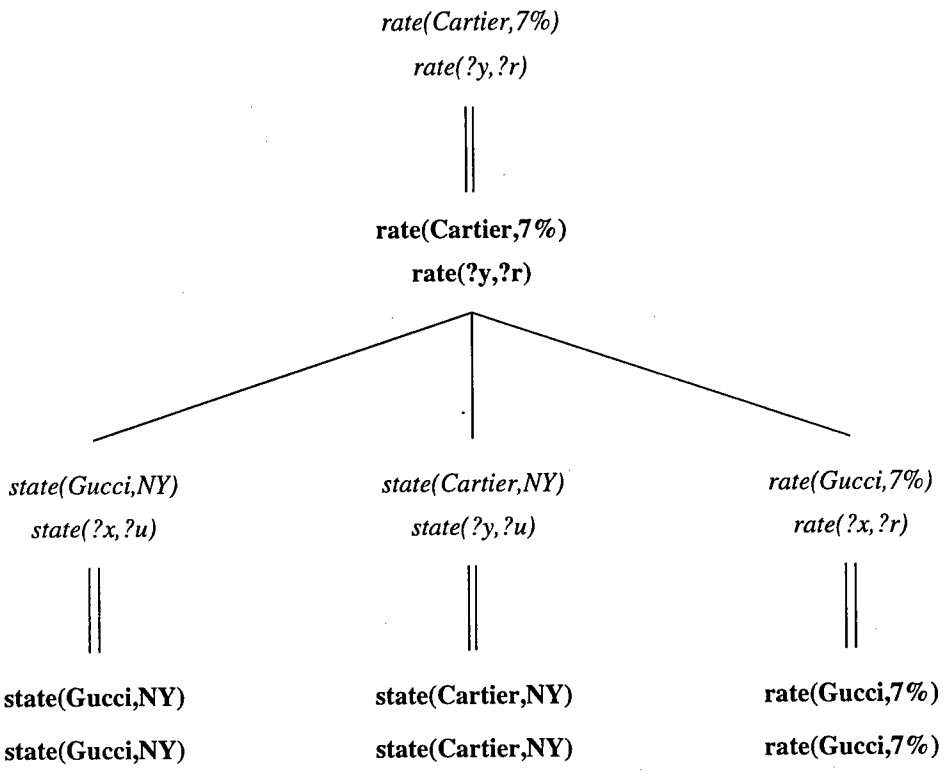

Figure 4.2: Determining the sales tax rate at Cartier in New York.

the same rate" is called a *determination*: a higher-order regularity that by itself is useless in reasoning, but which together with some premises leads to useful conclusions [28]. One can think of a determination as expressing information about similarities between situations in a certain class. Determinations allow the characteristics of a single situation to be extrapolated with confidence.[4]

Traditional EBL algorithms are incapable of acquiring any interesting new macro-operator from a proof involving a determination. If applied to the example proof above, a traditional EBL algorithm yields a macro-operator identical to the original domain theory determination.

4.5 Two Additional Explanation Transformation Operators

Given the inadequacies of traditional EBL algorithms just described, we would like to extend the framework of Section 4.2 so that new, more powerful, EBL algorithms (in particular, ones capable of learning from determinations, and ones capable of performing generalized caching under appropriate conditions) can be constructed.

The following two operators complete the EBL* family of proof-transformation operators.

Definition 10 : Operator 4 (*Match Edge Grafting*). Given a leaf subgoal node n_s and a consequent node n_c whose formulae unify, graft the proof rooted at n_c at the leaf subgoal n_s:

$$Op4(n_s, n_c) : \textbf{if } m(n_s) = \emptyset \wedge f(n_s) \circ f(n_c) \textbf{ then } m(n_s) \Leftarrow n_c.$$

Definition 11 : Operator 5 (*Rule Edge Pruning*). Given a subgoal node n_s and a substitution θ, delete the subtree rooted at n_s while applying the bindings θ to the label of the parent node $p(n_s)$:

$$Op5(n_s, \theta) : r(p(n_s)) \Leftarrow r(p(n_s)) - \{n_s\} \textbf{ and } l(p(n_s)) \Leftarrow l(p(n_s))\theta.$$

Operator 4 can be used (in concert with Operator 3) to emulate the IMEX algorithm [7], which "unravels" sections of a proof corresponding to previously acquired macro-operators by suturing in the proof from which the macro-operator was originally derived. Operator 5 enables pruning at rule edges: in particular, when given an appropriate θ, it is this operator that permits us to construct EBL algorithms that can learn the desired macro-operator for the determination example of the previous section.

4.6 A Domain-Independent EBL* Algorithm

As the proof of the completeness theorem suggests, the EBL* operators define the space of all alternative partial explanations. Recall that the general idea behind EBL is to transform the proof in some fashion, producing a maximally general — yet still valid — partial proof from which a new macro-operator is extracted. EBL algorithms differ in the control heuristics they use to guide the transformation process.

The operationality heuristics used in traditional EBL algorithms (*e.g.*, the *trim* procedure of Section 4.2) are examples of domain-independent control heuristics. While it may be the case that

[4] Determinations have been previously used in EBL work in a quite different way, in order to extend incomplete domain theories. The simplest case of the incomplete domain theory problem occurs when a query is known to be true, yet no proof of the query can be found. When the domain theory gives rise to a single failed proof involving a determination, the PROLEARN-ED algorithm [67] uses the determination to suggest a plausible assumption that permits the proof to go through. After asserting this assumption, PROLEARN-ED uses a traditional EBL algorithm to chunk the patched proof. In the special case where the original domain theory is assigned the Clark completion semantics [25], the PROLEARN-ED algorithm is deductively sound. Otherwise, the algorithm is doing a form of abduction, jumping to unsound but plausible conclusions guided by determinations present in the domain theory.

domain-dependent control heuristics are required in order to produce the best possible speedup, here we propose five general heuristics that can be used in constructing domain-independent EBL* algorithms. In particular, these heuristics not only acquire the desired macro-operator in the example of Section 4.4, but also automatically perform generalized caching where appropriate.

The first heuristic, also suggested in [73], recognizes that chains of reasoning based on single antecedent rules often express taxonomic *isa* relationships, which should not be compiled into the result of learning lest it become over-specific. Intuitively, the idea is to make the "highest" consequent node in a chain look like a domain theory fact.

Definition 12 : Heuristic 1 (*Trim single-antecedent chains*). If a leaf subgoal node n_s is an only child, then apply Operator 5 to prune the subtree rooted at n_s while preserving the binding constraints implicit therein:

$$H1(n_s) : \textbf{if } |r(p(n_s))| = 1 \wedge m(n_s)) = \emptyset \textbf{ then } Op5(n_s, l(n_s) \circ f(\mathit{lift}(n_s))).$$

The lifting step applied to n_s before its removal ensures that only those binding constraints contributed by the pruned subtree are retained, as opposed to all of the bindings of variables contained in $l(n_s)$.

We may have to apply Heuristic 1 several times to nibble away all single-antecedent structures from the proof. Also note that, since subsequent heuristics may apply Operator 5 (producing consequent nodes with one child that originally had more than one child), it is important to apply Heuristic 1 first so that it is not fooled into identifying such reduced portions of the proof as single antecedent chains. This also ensures that we retain the binding constraints implicit in the pruned subtree, since Heuristic 1 is applied before any significant changes are made to that subtree (*e.g.*, before anything like procedure *trim* is applied). Heuristic 1 is easily implemented as a simple recursive-descent algorithm that eats away at the leaves of the proof.

```
function trim-single-antecedent-chains(n : node) : boolean;
  begin
    if subgoal-node?(n) then
      if trim-single-antecedent-chains(m(n)) then
        begin
          lift(n);
          Op5(n, l(n) ∘ f(n));
          return t;
        end
      else return f;
    elseif consequent-node?(n) then
      if r(n) = {s} ∧ trim-single-antecedent-chains(s)) then return t;
      else
        begin
          for s ∈ r(n) do trim-single-antecedent-chains(s);
          return f;
        end
  end
```

The second heuristic governs another application of Operator 5. The insight is that certain subgoals provide background information that should be compiled into the learned macro-operator; essentially a form of *partial evaluation* [112].

Definition 13 : Heuristic 2 (*Trim alien subgoals*). If the label $l(n_s)$ of a subgoal node contains only variables not present in the label $l(p(n_s))$ of its parent, then the subtree rooted at n_s should be deleted using Operator 5, while preserving the binding constraints implicit therein.

$$H2(n_s) : \textbf{if } \textit{variables}(l(p(n_s))) \cap \textit{variables}(l(n_s)) = \emptyset \textbf{ then } Op5(n_s, l(n_s) \circ f(\textit{lift}(n_s))).$$

The function *variables(x)* returns the set of variables mentioned in its argument x.

As we shall see later, Heuristic 2 is useful in obtaining the desired generalization for the determination example. Like Heuristic 1, since we are interested in retaining the binding constraints implicit in the pruned subtree, it is important that Heuristic 2 also be applied before anything resembling procedure *trim*. Heuristic 2 is also easily implemented.

```
procedure trim-alien-subgoals(n : node);
  begin
    if consequent-node?(n) then for s ∈ r(n) do trim-alien-subgoals(s);
    elseif subgoal-node?(n) ∧ variables(l(p(n_s))) ∩ variables(l(n_s)) = ∅ then
      begin
        lift(n);
        Op5(n, l(n) ∘ f(n));
      end
    else trim-alien-subgoals(m(n));
  end
```

The third heuristic is a refinement of the traditional EBL operationality heuristic that only removes consequent nodes corresponding to domain theory facts if those nodes actively serve to impose binding constraints on the proof.

Definition 14 : Heuristic 3 (*Trim axioms selectively*). If a consequent node n_c is a leaf node whose label and formula are equally specific (*i.e.*, identical subject to variable renaming), apply Operator 3 to prune the subtree rooted at n_c:

$$H3(n_c) : \textbf{if } r(n_c) = \emptyset \wedge f(n_c) = l(n_c) \textbf{ then } Op3(p(n_c)).$$

Heuristic 3 is more selective than the operationality heuristic of Section 4.2 and plays a critical role in obtaining generalized caching behavior in the LT domain. Its implementation is a straightforward modification of procedure "*trim*".

```
procedure trim-axioms-selectively(n : node);
  begin
    if consequent-node?(n) then for s ∈ r(n) do trim-axioms-selectively(s);
    elseif subgoal-node?(n) ∧ r(m(n)) = ∅ ∧ f(n_c) = l(n_c) then Op3(n);
    else trim-axioms-selectively(m(n));
  end
```

The fourth heuristic is deceptively simple to state but, unfortunately, quite expensive to implement.

Definition 15 : Heuristic 4 (*Trim universally true subproofs*). If a single answer substitution θ subsumes all other possible answer substitutions for the formula of a subgoal node $f(n_s)$, then the subtree rooted at n_s should be deleted using Operator 5 to preserve θ:

$$H4(n_s) : \textbf{if } \exists\theta \in \mathrm{A}(f(n_s))\forall\sigma \in \mathrm{A}(f(n_s))f(n_s)\sigma \subseteq f(n_s)\theta \textbf{ then } Op5(n_s, \theta).$$

Heuristic 4 recognizes that if something can be shown true in the general sense at macro-operator construction time, there is no need to require any verification of the fact at macro-operator application time; simply remove the subgoal in question. It is the problem of recognizing that something is true in the general sense that is so expensive: Heuristic 4 as stated relies on non-resource-bounded proof enumeration to find a substitution that subsumes all other substitutions. We present no implementation of Heuristic 4, however, later we shall see two examples of efficient approximations of Heuristic 4 that can be used to construct practical generalization algorithms.

The last heuristic is perhaps the simplest of all. It recognizes that in some domains there is a certain amount of redundancy in the construction of a proof, such as might occur with frame axioms in situation calculus formulations of planning problems [44].

Definition 16 : Heuristic 5 (*Trim redundant subgoals*). If two subgoal nodes in a proof have identical formulae (note variable renaming substitutions are *not* allowed), then one of the subgoal nodes should be deleted along with its subtree:

$$H5(n_{s1}, n_{s2}) : \textbf{if} f(n_{s1}) \equiv f(n_{s2}) \textbf{ then} Op5(n_{s2}, \emptyset).$$

Heuristic 5 avoids extracting new macro-operators with redundant antecedents: antecedents that would cause later proofs using the macro-operator to perform the same work more than once. It should only be applied at the end of the proof transformation process, right before the new macro-operator is extracted.

procedure *trim-redundant-subgoals*(n : *node*, a : *set of atoms*);
  begin
    if *consequent-node?*(n) **then for** $s \in r(n)$ **do** *trim-redundant-subgoals*(s, a);
    elseif *subgoal-node?*(n) $\wedge$ $f(n) \in a$ **then** $Op5(n, \emptyset)$;
    else *trim-redundant-subgoals*($m(n), a \cup \{f(n)\}$);
  end

To see how these heuristics might be used, let us return to the tax rate example of Section 4.4. Recall the original proof (Figure 4.2) is simply an instantiation of the following domain theory rule:

$$rate(?y, ?r) \leftarrow state(?x, ?u) \wedge state(?y, ?u) \wedge rate(?x, ?r).$$

Applying Heuristic 2, we note that none of the variables of the left-most subgoal (*i.e.*, $?x$ and $?u$) appear in the rule consequent. This subgoal is pruned using Operator 5 and the bindings $?x/Gucci$ and $?u/NY$ are retained. We next apply Heuristic 3 to remove the two remaining consequent leaf nodes (Operator 3), and use "*lift*" (Operators 1 and 2) to obtain a maximally general, valid, proof tree. Given that this particular proof is the product of only one rule, the valid proof tree reflects exactly the structure of the original rule, less the pruned subgoal.

The new macro-operator that can be extracted from this partial proof tree is:

$$rate(?y, ?r) \leftarrow state(?y, NY) \wedge rate(Gucci, ?r).$$

While this rule is more useful than the original rule due to the reduction in number of subgoals and variables that must be bound, we can still do better. The rightmost subgoal $rate(Gucci, ?r)$ has an answer substitution $\{?r/7\%\}$ that subsumes all other answer substitutions for that subgoal within this domain theory. By Heuristic 4, it can thus be removed via application of Operator 5 while "compiling in" the substitution $\{?r/7\%\}$:

$$rate(?y, 7\%) \leftarrow state(?y, NY)$$

which is the desired macro-operator.

While the heuristics just described do produce the desired determination, we note that the application of Heuristic 4 in the general case entails enumerating all possible proofs for an expression. Fortunately, two special cases of Heuristic 4 can be implemented efficiently and, when combined, provide much of the power of Heuristic 4.

The first special case of Heuristic 4 recognizes that the null substitution $\emptyset$ subsumes any other substitution. If we can prove a skolemized version of the formula of the node (within some reasonable resource bound), then the formula is universally true, *i.e.*, $\emptyset$ is a valid answer substitution for the original formula.

Definition 17 : Heuristic 4a (*Trim universal subproofs*). If $f(n_s)$ is universally true, then delete the subtree rooted at subgoal n_s:

$$H4a(n_s) : \textbf{if } \emptyset \in \mathrm{A}_R(skolemize(f(n_s))) \textbf{ then} Op5(n_s, \emptyset).$$

The function $skolemize(x)$ returns a copy of its argument x with each variable replaced by a fresh constant.

Unlike the general statement of Heuristic 4, Heuristic 4a employs resource-bounded search and does not rely on proof enumeration. A reasonable resource bound might be determined by inspecting the resources required to construct the original proof. For example, a resource bound based on the depth of the proof tree rooted at n_s might reasonably be used to limit the depth of search for a universally true equivalent.

procedure *trim-universal-subproofs*(n : *node*);
  begin
    if *consequent-node?*(n) **then for** $s \in r(n)$ **do** *trim-universal-subproofs*(s);
    elseif *subgoal-node?*(n) $\wedge$ *prove*(*skolemize*($f(n)$), $1 +$ *depth*(n)) $= \emptyset$ **then** $Op5(n, \emptyset)$;
    else *trim-universal-subproofs*($m(n)$);
  end

Heuristic 4a can be used to automatically recognize situations where generalized caching is appropriate, as in the LT domain. An EBL* strategy that includes this approximation of Heuristic 4, in addition to functioning as a normal EBL algorithm, is able to reduce any valid LT proof tree to its maximally general root node; thus recognizing and automatically performing generalized caching as a special case.

The second special case recognizes that if there is only one substitution θ that makes $f(n_s)$ true, then that substitution subsumes all other substitutions.

Definition 18 : Heuristic 4b (*Trim singleton subproofs*). If $f(n_s)$ has only one true substitution, then delete the subtree rooted at subgoal n_s:

$$H4b(n_s) : \textbf{if } \Pi_R(f(n_s)) = \langle \rho_1, fail \rangle \wedge \mathrm{A}_R(f(n_s)) = \langle \theta \rangle \textbf{ then} Op5(n_s, \theta).$$

As with Heuristic 4a, a reasonable resource bound might again be determined by examining the original proof structure.

procedure *trim-singleton-subproofs*(n : *node*);
  begin
    if *consequent-node?*(n) **then for** $s \in r(n)$ **do** *trim-singleton-subproofs*(s);
    elseif *subgoal-node?*(n) $\wedge$ *prove*($f(n)$, $1 +$ *depth*(n)) $= \theta \wedge$ *continue*() $=$ *fail* **then** $Op5(n, \theta)$;

```
    else trim-singleton-subproofs(m(n));
  end
```

It is this second heuristic, which performs a very limited form of resource-bounded proof enumeration, that can be used to prune the $rate(Gucci, ?r)$ subgoal in the tax example. Note, however, that in practice we must temporarily remove the recursive rule for *rate* and perform a resource-limited search only for alternative base cases in order to recognize the proof's singleton nature [105].

Given the domain-independent heuristics just described, we are now ready to construct a domain-independent EBL* algorithm. Our algorithm, denoted EBL*DI, is easily expressed as a composition of the heuristics introduced previously.

```
function ebl*di(ρ : proof) : rule;
  begin
      trim-single-antecedent-chains(root(ρ));
    trim-alien-subgoals(root(ρ));
    trim-axioms-selectively(root(ρ));
    lift(root(ρ));
    while trim-universal-subproofs(root(ρ)) do lift(root(ρ));
    while trim-singleton-subproofs(root(ρ)) do lift(root(ρ));
    trim-redundant-subgoals(root(ρ), ∅);
    return chunk(ρ);
  end
```

Note that some heuristics are applied only once, while others are applied repeatedly as long as they continue to alter the proof. In addition, each heuristic that either violates the validity of the proof or leaves it overly constrained is followed by an application of "$lift$" to restore the validity and generality of the resulting partial proof.

As with a traditional EBL algorithm, the validity of learned knowledge is dependent on the validity of the original domain theory. Traditional EBL formulations expect the domain theory to be both correct and stable. Retracting rules or facts from the original domain theory after learning may compromise the validity of any learned rules. Similarly, some EBL* transformation strategies that rely on Heuristic 4 are implicitly dependent on a form of the *closed-world assumption* [85]. For example, Heuristic 4b is a form of *negation as failure* [64]; subsequent addition of a new fact to the domain theory may change the usefulness of the learned rule.

The EBL*DI algorithm is truly a domain-independent learning algorithm in the sense that it is useful over a broad range of domains. Unlike traditional EBL, EBL*DI is not only capable of handling the determination in the tax rate example, but also reduces to performing generalized caching in domains where appropriate, such as the LT domain. Perhaps more interesting, however, is the fact that EBL*DI is free to mix generalized caching on one portion of a proof with the use of a determination in another (or even the same) portion of the proof as appropriate. In Section 4.7, we support the superior performance of EBL*DI in an empirical comparison with traditional EBL across several domains.

In practice, we expect that improved EBL* transformation strategies for learning macro-operators may well be domain dependent. Taking specific knowledge of a particular domain into account should lead to better, more useful, generalizations. For example, in the tax rate generalization above, "knowing" that a store can have only one tax rate (*i.e.*, that *rate* defines a function from

its first argument to its second) would support a very efficient, domain-specific, implementation of Heuristic 4. This kind of information is often easily included in the original domain theory specification: here, for example, as the first-order sentence $\forall ?x, ?u, ?v\ rate(?x, ?u) \wedge rate(?x, ?v) \rightarrow ?u = ?v$. Alternatively, if a nonmonotonic semantics such as the standard minimal model semantics is adopted for facts and rules, then functionality assertions are logically entailed, and they can be proven using special inference rules [38].

4.7 Evaluating EBL*DI

In this section, we present an empirical comparison of our EBL*DI algorithm with the traditional EBL algorithm of Section 4.2. The central question we want to resolve is whether macro-operators produced our EBL*DI algorithm reliably outperform macro-operators acquired by a traditional EBL algorithm across a spectrum of application domains. We are interested in measuring the change in performance on subsequent problems after learning.

Unlike our evaluation of bounded-overhead caching in Section 3, we used an even simpler experimental method for this comparison. Each experiment reported here used a different domain theory and problem set. The general idea is to partition the problem set, originally of size N, into two mutually exclusive subsets, a training set of size k and a test set containing the remaining $N - k$ problems. Two otherwise identical depth-first unit-increment iterative-deepening theorem provers were allowed to learn from each problem in the training set using different learning algorithms, and then are tested on the test set.[5] We recorded CPU time required and number of nodes explored for each problem in the test set. We then compared these numbers to those obtained with an identical, non-learning, theorem prover on the same test set. As is normal practice, we assumed that the cost of learning is negligible in the sense that it can be amortized over many subsequent problems.

4.7.1 Experiment 7

In our first EBL experiment, we compared the performance of our EBL*DI algorithm and the traditional EBL algorithm in the same blocks microworld used in the caching experiments of the previous section.

The blocks-world domain theory is highly recursive and the problem-solving search space is highly redundant. A typical new rule serves only to increase the branching factor of the search space and is of negative utility. In fact, training on a single problem yields useful rules (*i.e.*, produces some overall speedup on the remaining 25 problems) only 8 of 26 times with traditional EBL. The EBL*DI algorithm is superior, in that it produces net speedup for 5 additional problems, for a total of 13 of 26 cases. We used only those 8 problems from which even traditional EBL could extract rules of positive utility to investigate the extent to which the utility of multiple acquired rules is cumulative. For this experiment, our hypotheses are that (*i*) EBL*DI learns macro-operators that give greater speedup than those learned by traditional EBL, and (*ii*) the greater the number of problems from which EBL*DI can learn, the greater the speedup.

The results of experiments testing these hypotheses are illustrated in Figures 4.3 and 4.4. The EBL*DI system is significantly faster on average than the EBL system for all tested training sets and training set sizes. Furthermore, learning from more problems makes both systems run faster.[6]

[5] As noted previously, unit increment may well produce the worst-case performance for iterative deepening.

[6] It is technically possible for a node exploration ratio under one to correspond to a CPU time ratio over one, since the overhead of using an additional rule may increase the average node exploration cost [109].

This "expensive chunk problem" does not appear to be an issue in this experiment, as the general shapes of the curves in Figures 4.3 and 4.4 are quite similar. More precisely, we observed that average node exploration cost is

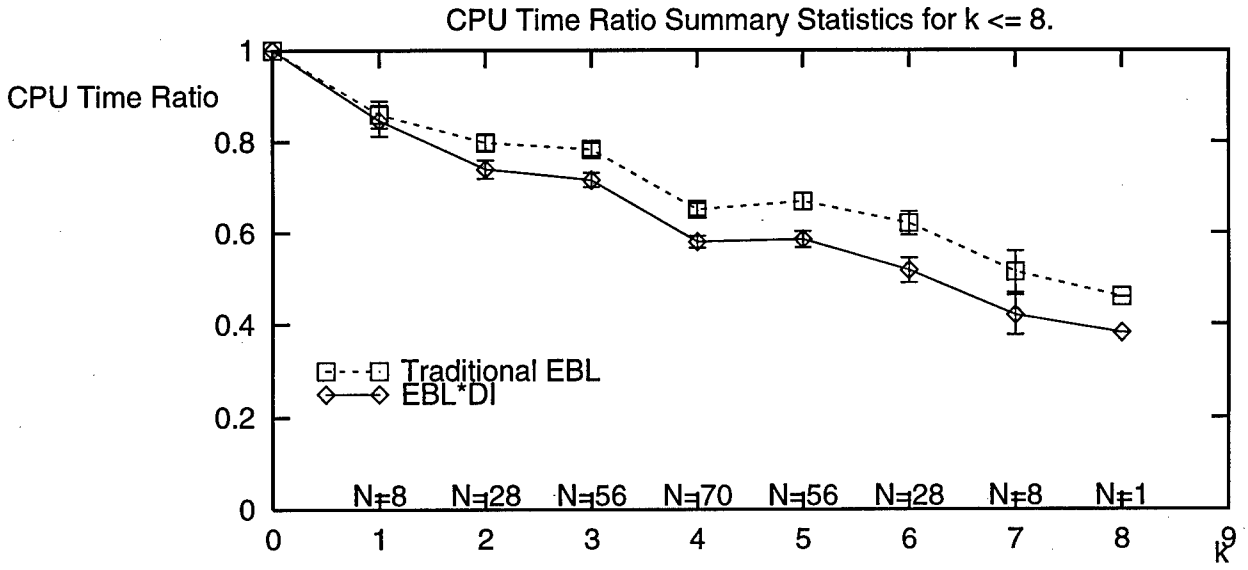

Figure 4.3: Average CPU time ratios for selected training sets with $k \leq 8$.

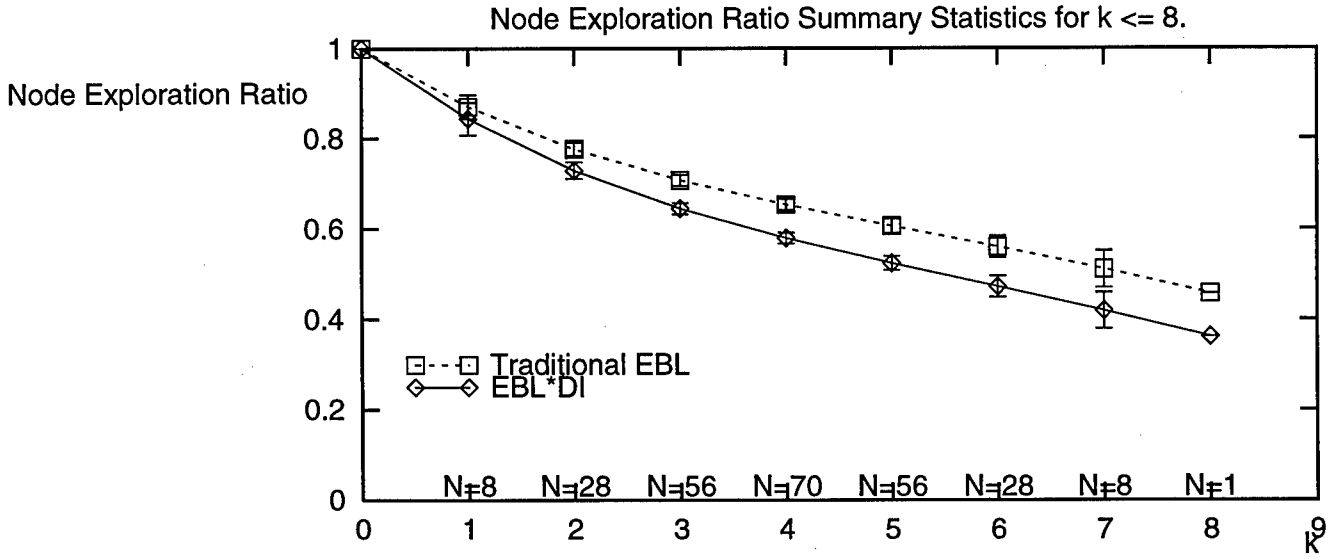

Figure 4.4: Average node exploration ratios for selected training sets with $k \leq 8$.

These results are quite impressive: by learning from appropriate problems, the EBL*DI system can solve the remaining test problems can be solved in as little as 32% of the time while searching as few as 33% of the nodes searched by an otherwise equivalent non-learning system. However there are two important caveats. The first is that the results are specific to this particular microworld and problem distribution. We hope that similar results hold in other domains, but no experimental evaluation can prove this.

More important, this experiment finesses the problem of *what* to learn. Learning from random examples is fundamentally more difficult than learning from well-chosen examples supplied by a teacher [111]. The training sets used in the experiment here consist entirely of problems from which traditional EBL can learn useful macro-operators. Training sets consisting of randomly chosen problems typically do not give speedup in this domain.

4.7.2 Experiment 8

In this next experiment, we revisited the classic LT experiment [77], using an updated version of the original LT domain theory. Queries in the LT domain are statements in the propositional calculus, that is, fully ground expressions, such as $thm(or(not(or(not(P), not(P))), not(P)))$. We rewrote the 92 propositional calculus problems from Chapter 2 of *Principia Mathematica*, replacing *implies* with *or* and *not*: 87 unique problems remain after rewriting. Unlike the blocks microworld problems of the first experiment, the LT problems were originally ordered by the authors of *Principia Mathematica* to maximize their pedagogical utility.

The domain theory consists of three rules and five facts, which correspond to the first five theorems from Chapter 2 of *Principia Mathematica*. Domain theory facts are generalized propositional statements such as $axm(or(not(or(?a, ?a)), ?a))$ that rely on universally quantified variables to allow for constant renaming in the query expressions. In this fashion, both the queries $thm(or(not(or(P, P)), P))$ and $thm(or(not(or(Q, Q)), Q))$ will eventually match the same domain theory fact. The three domain theory rules are

$$\begin{aligned} thm(?x) &\leftarrow axm(?x) \\ thm(?x) &\leftarrow axm(or(not(?y), ?x)) \wedge thm(?y) \\ thm(or(not(?x), ?z)) &\leftarrow axm(or(not(?x), ?y)) \wedge thm(or(not(?y), ?z)). \end{aligned}$$

Since each of these rules has at most one recursive *thm* subgoal, they give rise to the same kind of linear proof structure produced by the original LT system (see Figure 4.5).

The hypotheses tested in this experiment are that (*i*) learning from early problems helps in solving later problems, and (*ii*) EBL*DI outperforms both traditional EBL and rote learning. The experiment consists of four trials. The first trial is a simple non-learning trial: all 87 problems are attempted, and statistics describing whether or not each problem is solved as well as solution characteristics are recorded. The remaining three trials are learning trials, where a learning algorithm (rote learning, traditional EBL, or EBL*DI) is applied to each solved problem with a proof larger than one node; the results of learning are then available for use on subsequent problems.

The protocol just described differs significantly from that used in our first experiment, where separate training and test sets were used. The protocol used here better suits the sequential nature

almost independent of training set size for both learning algorithms. For the traditional EBL system, a very small but statistically significant positive correlation between training set size and average node exploration cost node was observed. However, for the EBL*DI system, we observed a very small but statistically significant *negative* correlation.

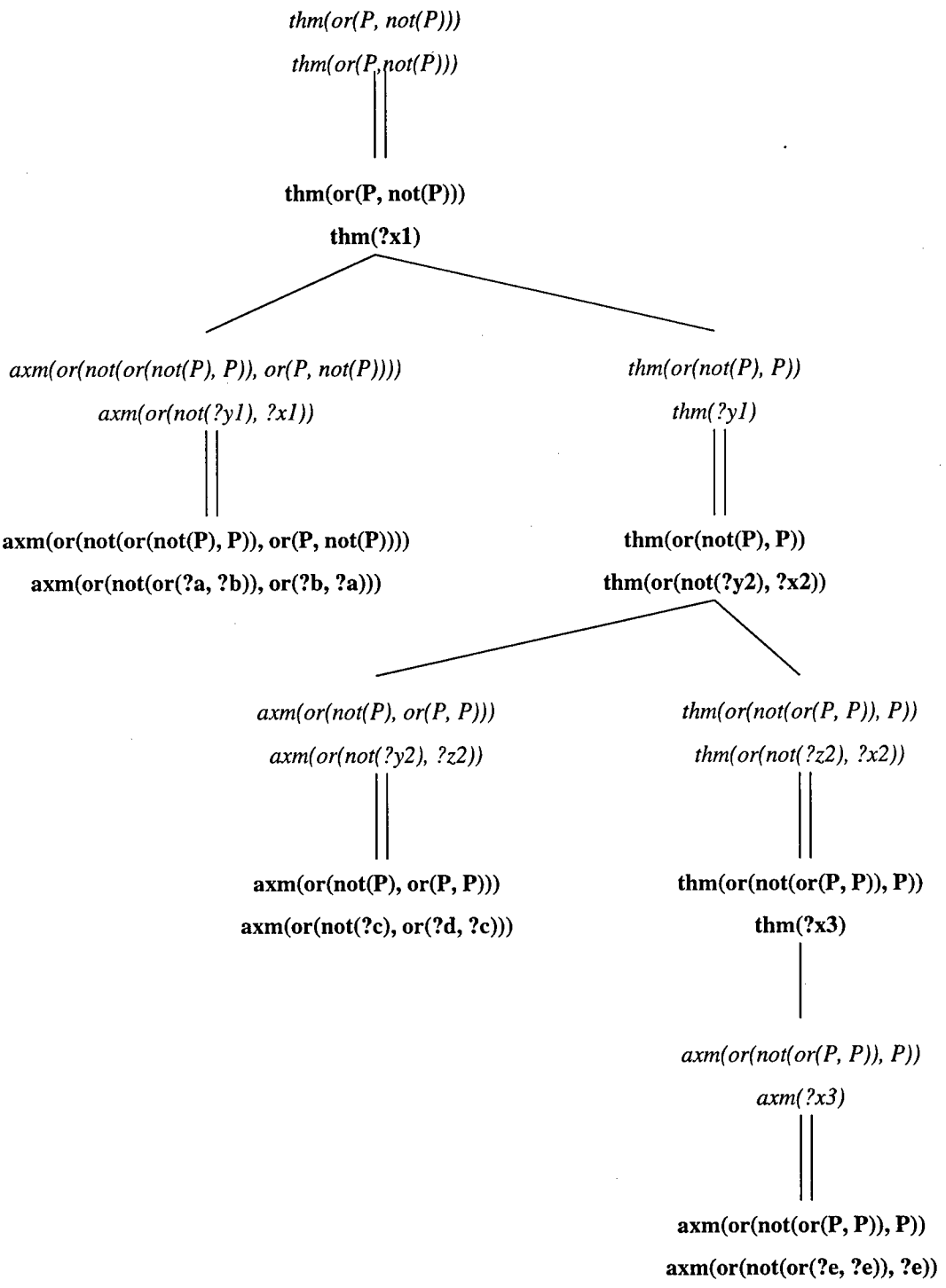

Figure 4.5: Proof of $thm(or(P, not(P)))$.

	No Learning	*Rote*	*EBL*	*EBL*DI*
Problems Solved	34	38	30	44
CPU Time Ratio	1	0.72	16.62	0.13
Node Exploration Ratio	1	0.76	3.13	0.17

Table 4.1: Summary Results for LT Domain.

of the LT problem set, and is in the spirit of the original LT experiments.[7] Unfortunately, this protocol presents some interesting statistical problems when one tries to apply the experimental analysis methods used for the other experiments. For this reason, we restrict ourselves to qualitative comparisons of the different systems.[8]

Summary statistics for the four trials are shown in Table 4.1. Each trial was performed under an identical resource bound of 50,000 node explorations per problem. Any problem left unsolved by the non-learning system that was subsequently solved by one of the learning systems was reattempted using the non-learning system with an extended resource bound in order to compute CPU time and node exploration ratios. Such problems are, of course, not included in the results reported for the non-learning system.

The results are generally in line with those reported in [78]:

1. The non-learning system solved 34 of the 87 problems within the resource bound. Its CPU time ratio and node exploration ratios are, by definition, 1.

2. The rote learning system solved 4 additional problems for a total of 38 problems solved. On average, the rote learning system searched fewer nodes (76% of those searched by the non-learning system) and required less time (72% of the CPU time required by the non-learning system).

3. The traditional EBL system failed to solve 5 problems that were solved by the non-learning system within the resource bound. In return, it was able to solve 1 additional problem not solved by either the non-learning or rote-learning systems. On average, however, the traditional EBL system searched a far greater number of nodes (over 300% of those searched by the non-learning system) and was also much slower than the non-learning system (CPU time ratio of over 1600%) for those problems that it did manage to solve.

4. The EBL*DI system solved every problem solved by any other tested system. It solved 10 problems more than the non-learning system, 6 problems more than the rote learning system and 14 more than the traditional EBL system. It searched far fewer nodes (17% of those searched by the non-learning system) and was also faster (CPU time ratio of about 13%) than any other system.

[7]The protocol also differs from the original LT protocol of [77], which allowed rote learning (*i.e.*, caching) of unsuccessful problems as new domain theory elements. In general, learning from unproven propositions may augment a domain theory with untrue facts, and it makes the performance contribution of a particular learning algorithm difficult to isolate. Therefore our protocol follows that of [78].

[8]These interesting statistical problems are the topic of [43], where the data from this experiment is evaluated using a more sophisticated nonparametric statistical test specifically designed for censored data. While the statistical foundations of [43] are well beyond the scope of this report, we should note here that the qualitative observations reported for this experiment are in fact in agreement with the more rigorous conclusions discussed in [43].

These results support our experimental hypotheses. In particular, we see that the traditional EBL algorithm often acquires macro-operators of negative utility. The branching factor of the search explodes as the acquired macro-operators are added to the domain theory. Two factors account for this explosion: the generality of the macro-operator consequents and the number of macro-operator antecedents. The macro-operators acquired by rote learning are nothing more than cached axioms, with very specific consequents and no antecedents. Their specificity guarantees their low overhead, but also makes them less useful.[9] In contrast, since the EBL*DI macro-operators are more general than those acquired by rote learning, they are more widely useful. At the same time, their consequents are not general enough to cause the branching factor explosion observed with traditional EBL.

The principal result is that the EBL*DI algorithm *automatically* performs generalized caching in the LT domain: it is a general-purpose EBL algorithm, and not a special-purpose generalized caching algorithm. In fact, it is precisely the same algorithm used in both the previous experiment and the next experiment.

4.7.3 Experiment 9

In this third EBL experiment, we tested one of the original intuitions motivating EBL, that is, that problem solvers are typically posed a series of problems selected according to some skewed yet *a priori* unknown distribution. The desired effect of learning is then to "tune" the problem solver to this particular query distribution, improving its overall expected performance as a consequence. In this experiment, we used a synthetic domain theory to test the performance of traditional EBL and EBL*DI on two different problem distributions, where one of the distributions is uniform and one is weighted to a subspace of problems. Our hypotheses are that (*i*) both EBL algorithms perform better on the skewed distribution, and (*ii*) the EBL*DI algorithm performs better than the traditional EBL algorithm.

In order to have control over the query distribution, we used an artificial theory. The synthetic domain theory used in our experiment consists of 330 rules and 38 facts. It was generated in a restricted first-order language without function symbols: thus, while the theory may entail an infinite number of valid proofs, there are only a finite number of atomic formulae within the deductive closure. Restricting the language in this fashion allows us to efficiently compute the deductive closure in a forward-chaining fashion. For the theory used here, there are 976 unique maximally general atomic formulae within the deductive closure.

We generated two 135-element problem sets from this 976-element deductive closure. The first set was generated from a seed set sampled from the deductive closure according to a uniform probability distribution, while the second set was generated from a seed set drawn according to a skewed probability distribution. Queries are either identical to the seed formula, more specific instances of the seed formula (if it contains universally quantified variables), or copies of the seed formula with new existentially quantified variables replacing randomly selected seed formula constants. Thus while the queries so generated are guaranteed to have corresponding instances within the domain theory, finding proofs of these queries may involve substantial search (a number of the queries generated do not yield a solution within $2 \cdot 10^7$ node explorations by the non-learning system). The first quasi-uniformly distributed set contains 113 unique queries, while the second set contains 96 unique queries (queries are duplicated due either to repeated seeds or to the introduction of existentially quantified variables in the seed).

[9] One of the techniques proposed in [72] was, in fact, to prohibit chaining on the antecedents of acquired macro-operators in order to reduce the branching factor explosion.

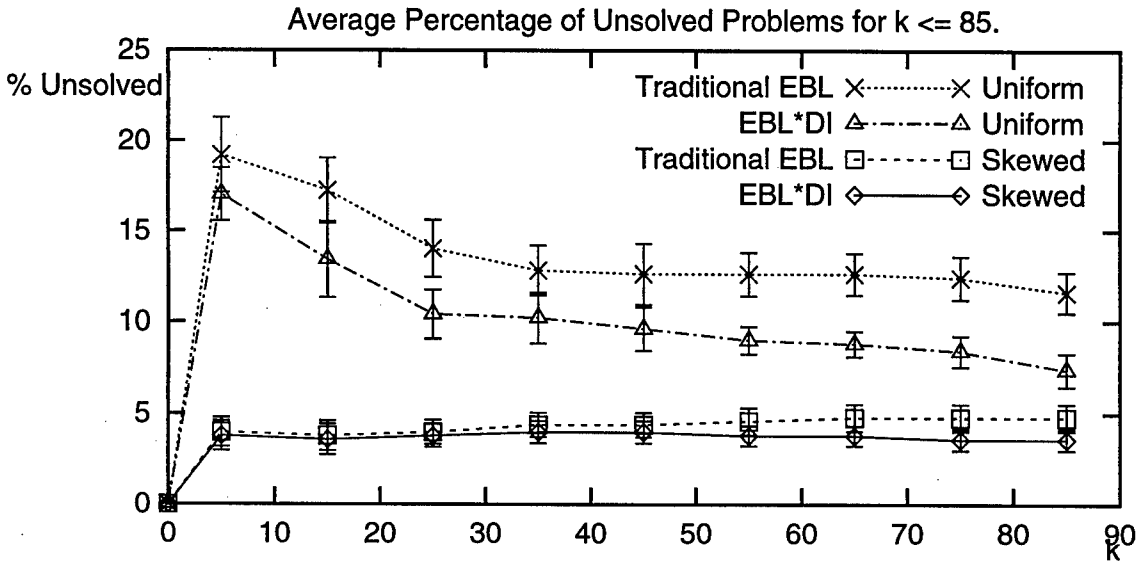

Figure 4.6: Average percentage of unsolved test problems for nested training sets with $k \leq 85$.

We randomly partitioned each 135-element problem set into an 85-element training set and a 50-element test set. Ten such random partitions were generated. For each partition, we performed nine trials for each learning system. The first trial involved learning from 5 randomly selected problems from the training set and testing on all 50 test problems. Each subsequent trial involved training on 10 additional randomly selected training problems. We performed all trials with a 200,000 node exploration resource limit, and we compared the results to the performance of a non-learning system operating on identical test sets.

Figure 4.6 plots the average percentage of unsolved test problems and makes clear the effect of the utility problem. The non-learning system is able to solve every test problem within the 200,000 node exploration resource limit, while both learning systems lose the ability to solve a certain percentage of the test problems within the same resource bound. The ability of the EBL*DI system's intrinsically more useful macro-operators to mitigate the adverse effects of the utility problem are especially clear in the quasi-uniform distribution case, where the effects of the utility problem are more pronounced. For the skewed distribution case, the difference is much less striking, especially for smaller training sets. Both learning systems suffer less in the skewed distribution case, where the training problems are by construction more likely to reflect the composition of the test problem set.

Of course, the adverse effects shown in Figure 4.6 should be balanced against any performance improvement provided by the macro-operators on the remaining test problems. Figure 4.7 plots average CPU time ratio for solved problems. As for Figure 4.5, we again see our hypotheses are supported. First, it is clear that, regardless of learning algorithm employed, the skewed problem distribution case initially yields more CPU time reduction than the quasi-uniform problem distribution case. This is due to the greater "predictive power" of the training set with respect to the test set in the skewed distribution case; as the training set size increases, this advantage disappears. Second, we see that the EBL*DI learning system provides greater speedup than the traditional EBL algorithm for both problem distributions. Even where the difference is not large, since the EBL*DI system leaves significantly fewer unsolved problems than the traditional EBL system (Figure 4.6), this supports our hypothesis that EBL*DI macro-operators are intrinsically

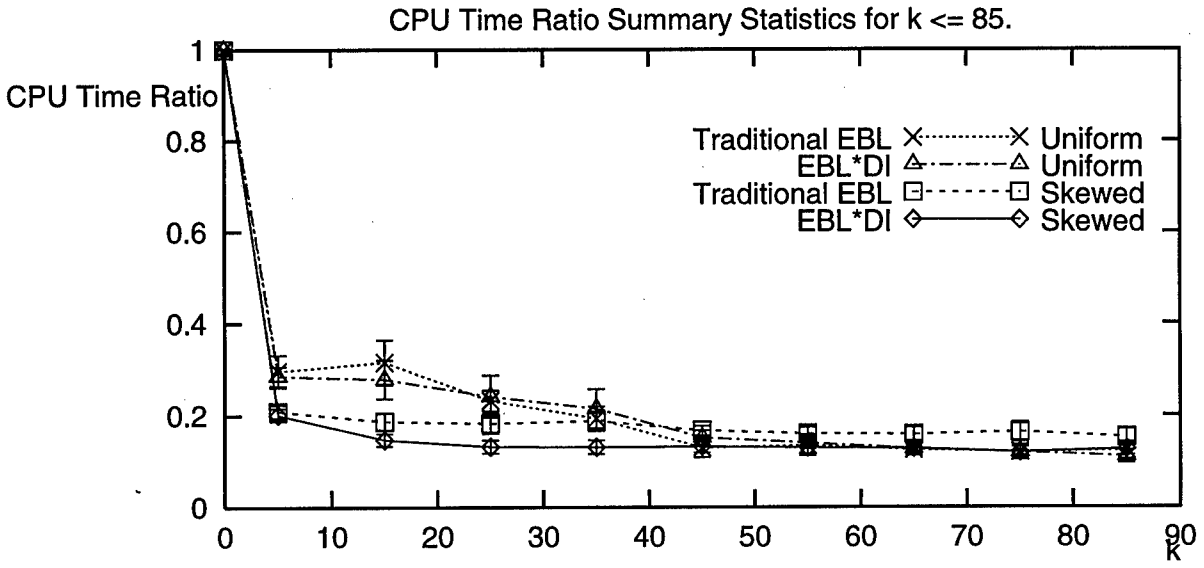

Figure 4.7: CPU time ratios for nested training sets with $k \leq 85$.

more useful than macro-operators produced by a traditional EBL algorithm. Similar conclusions are supported when one examines the node exploration ratios (Figure 4.8). In summary:

1. Independent of the distribution tested, the EBL*DI system solves more problems faster and with fewer nodes explored than does the traditional EBL system within the same resource limit.

2. The performance of both learning algorithms is better on the skewed problem distribution than on the quasi-uniform problem distribution. As expected, the proper selection of training problems has an enormous impact on the overall usefulness of explanation-based learning, regardless of the actual learning algorithm applied.

It should be mentioned that there is one source of experimental bias present in the results just reported. As should be clear from Figure 4.6, due to the large size of the problems in both problem sets, both systems often did not solve all of the test problems within the resource bounds imposed. For this reason, the results shown in Figures 4.7 and 4.8 are *optimistic* estimates of the real values: increasing the resource limits so that more problems are solved would most probably *increase* average CPU time ratios as well as average node exploration ratios. The bias is more pronounced in the quasi-uniform distribution case, where a larger number of problems were left unsolved. Thus, as shown in [94], it is theoretically possible for the apparent advantage of the EBL*DI system over the EBL system to erode or even to vanish entirely as the resource limit is increased. This eventuality is rather unlikely, however, as over all trials and for both distributions the problems solved by EBL*DI and unsolved by traditional EBL outnumber the problems solved by traditional EBL and unsolved by EBL*DI by a margin of greater than 10 to 1.

4.8 Summary of EBL*DI

The EBL*DI algorithm is superior to traditional EBL algorithms in at least three ways. First, it is able to acquire useful macro-operators in situations where traditional algorithms cannot, such as

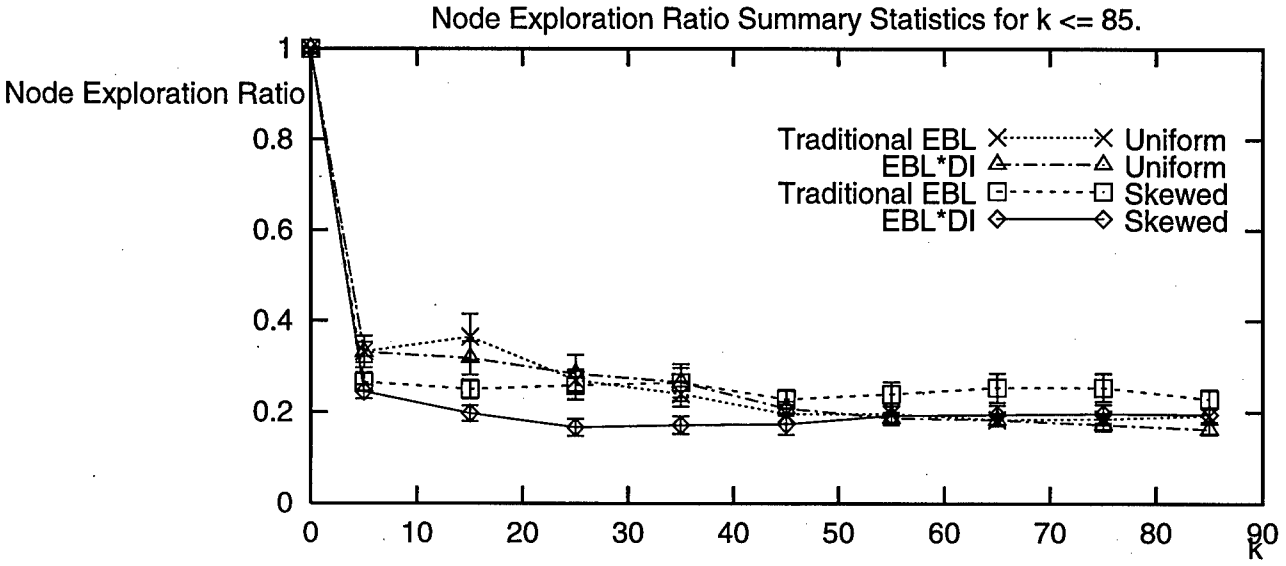

Figure 4.8: Node exploration ratios for nested training sets with $k \leq 85$.

in the determination example of Section 4.4. Second, it produces macro-operators of significantly greater utility than those produced by traditional EBL algorithms, a claim supported empirically by the experimental results of Section 4.7. Finally, the generality of EBL*DI's control heuristics — which are declaratively specified — allow the same algorithm to be used effectively across a broad spectrum of application domains, including the LT domain for which a special-purpose EBL algorithm was previously proposed.

Chapter 5

Nagging and the DALI Inference Engine

This chapter introduces a parallel search-pruning technique called nagging. *Nagging is sufficiently general to be effective in a number of domains; here we focus on an implementation for first-order theorem proving, a domain both responsive to a very simple nagging model and amenable to many refinements of this model. Nagging's intrinsic fault tolerance and exceptional scalability make it particularly suitable for application in commonly available, low-bandwidth, high-latency distributed environments. We present several nagging models of increasing sophistication, demonstrate their effectiveness empirically, and compare nagging with related work in parallel search.*[1]

5.1 Introduction

Combinatorial search is among the most rudimentary strategies for problem solving. Unfortunately, while search is the only known approach for many interesting problems, it is fundamentally inconsistent with efficient computation. This in no way diminishes the importance of these problems. It just means that we can't expect to solve them without some trial and error.

In its most naive form, combinatorial search involves generating and then testing each candidate solution. Of course, if suitable problem-specific knowledge is available, one can do considerably better than blindly trying all possibilities. Exploiting such knowledge can prune away whole regions from the space of candidate solutions, dramatically reducing solution time. Thus while we may not be able to solve search problems in polynomial time, improvements in search technology can significantly increase the size of the largest problem solvable within a limited amount of time. This is the goal of our research; we are interested in applying various types of problem-specific information to improve the resource-bounded performance of a search-based problem solver.

Unfortunately, exploiting a particular aspect of problem-specific structure may not reduce search uniformly across all problem instances. This is further confounded by the fact that structural properties common among problems of interest are not always well understood. Thus, a search mechanism that is effective in one case may be substantially less effective on other, seemingly very similar problem instances.

An example will help to illustrate this point. Consider a variant of the classic N-queens problem which we call the $M:N$-queens problem.[2] As usual, a solution represents a placement of N queens

[1] This chapter is adapted from [102].

[2] Although N-queens is not a particularly good example of a search problem, we use it because of its simplicity

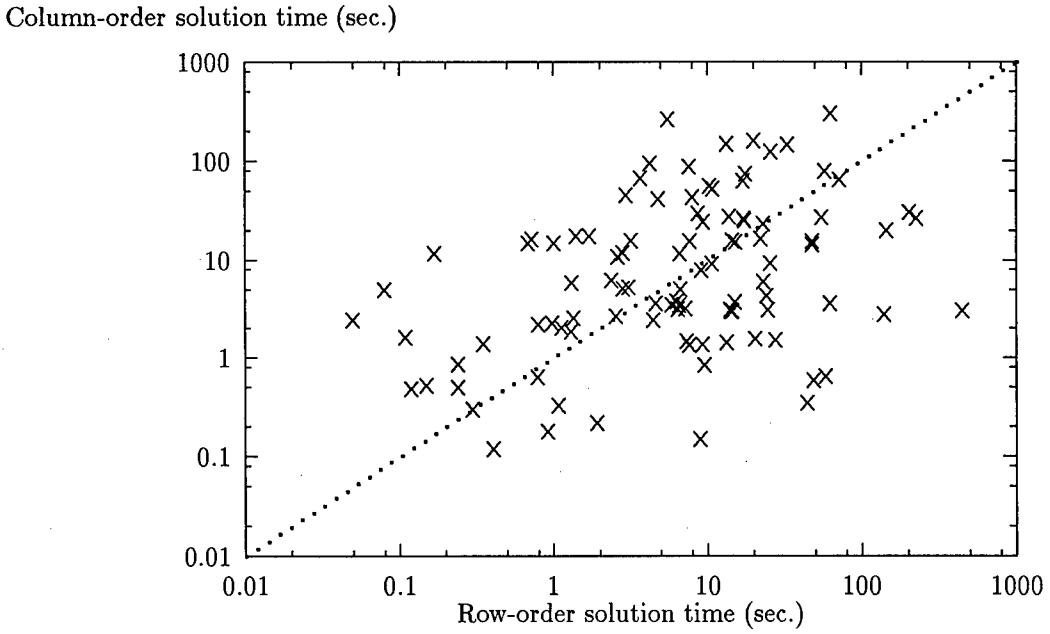

Figure 5.1: Performance comparison of two closely related search procedures on 100 randomly generated instances of the 60:80-queens problem. While the average performance of these systems should be identical, their comparative proficiency on individual problem instances varies greatly.

on a $N \times N$ chess board such that no queen is threatened. Unlike the standard N-queens problem, however, some number $M < N$ of queens are placed on the board in advance in an initial, threat-free configuration.

We now compare the effectiveness of two slightly different search procedures on a collection of 60:80-queens problems. One procedure searches for a solution by trying to place a queen on each unoccupied row of the board, from top to bottom. The other fills the board one column at a time from left to right. Both procedures reject partial solutions in which a queen is already threatened. Figure 5.1 plots the search time for the column-by-column system against the row-by-row system on 100 problems. Each point represents the solution of a randomly-generated 60:80-queens problem. Each system was required to complete the solution, or to report failure if no solution was possible.

On standard N-queens problems, these systems would be equally proficient, and all datapoints in Figure 5.1 would lie close to the diagonal. On $M:N$-queens problems, however, the two systems exhibit radically varying performance from one problem to the next. This is representative of the behavior of many search procedures in general; it is difficult to know beforehand how effective a given approach will be on a given problem instance. Two techniques that offer comparable performance on one problem may differ dramatically on another problem instance.

Nagging is a parallel search-pruning technique specifically designed to exploit this problem-to-problem variation in search behavior. Conventional approaches to reducing search (*e.g.*, subgoal caching) pay a polynomial amount of overhead for the chance to avoid exponentially-sized portions of the search; essentially, they try to trade inefficient computation for efficient whenever possible. Nagging attempts to avoid exploring parts of the search space by examining them in parallel under an alternative formulation or search procedure. Although still exponential, these alternative search problems may be dramatically smaller than the original, so a parallel nagging process may come up with an answer much more quickly than the standard search. Such an approach would be quite effective on the problems in Figure 5.1, where solution times differ by more than a factor of two on

and likely familiarity to the reader.

74 of the 100 problem instances.

In Section 5.2 we describe a general framework for nagging. In Section 5.3, we instantiate this framework as part of the *Distributed Adaptive Logical Inference* (DALI) search engine, a first-order theorem prover based on model elimination. Next, we describe several technical refinements to both nagging and the internals of DALI designed to enhance performance in a large class of domains. In Section 5.6, we present an empirical evaluation of DALI and the performance advantage of nagging on problems drawn from Version 1.1.1 of the *Thousands of Problems for Theorem Provers* (TPTP) problem set. Finally, we compare nagging to related work in parallel search and theorem proving and outline the direction of our continuing research.

5.2 Nagging

We begin by introducing some relevant notation. In general, T will be used to denote the search tree and δ to represent the individual nodes that comprise T. For node δ, $c(\delta)$ represents the set of children of δ. With some overloading of the symbol T, $T(\delta)$ is used to denote the subtree rooted at δ.

Nagging depends on a *problem transformation function*, f, mapping search trees to alternative search trees:

DEFINITION (Problem Transformation Function). f is a problem transformation function if for every search tree T, $f(T)$ is another search tree such that $f(T)$ contains solutions whenever T does. The class of all such functions is denoted by $\mathcal{F}$.

Intuitively, functions in $\mathcal{F}$ exchange a search problem in one domain for a "simpler" problem in some new domain. We say the transformed problem is simpler because it must have a solution whenever the original does.

The most useful consequence of this definition is that knowledge about $f(T)$ can sometimes obviate the need to explore T itself. In particular, if $f(T)$ is known to contain no solutions, then T cannot contain a solution. When the cost of exploring $f(T)$ is small compared to T, it may be beneficial to use $f(T)$ as an indicator of whether or not searching T would be productive.

Nagging is designed to take maximal advantage of this property. Two types of processes are used. A *master process* explores the search tree T under some serial search discipline. One or more *nagging processes* operate asynchronously and in parallel to the master. Each nagging process monitors the operation of the master and periodically selects some node $\delta \in T$ such that the master has started but not yet finished exploring $T(\delta)$. The nagger applies a transformation $f \in \mathcal{F}$ to $T(\delta)$ and attempts to solve the resulting transformed search problem. If the nagger exhaustively explores $f(T(\delta))$ without finding a solution, the definition of $\mathcal{F}$ guarantees that the master's search of $T(\delta)$ is futile and can be abandoned without risk of missing a solution. If, however, the nagging process finds a solution in $f(\delta)$, no search pruning is warranted by the definition of $\mathcal{F}$, but the nagger is free to work on a new transformed subproblem.[3]

Figure 5.2 gives a simple example of how nagging might reduce the master's search on N-queens problems.[4] Assume that the master process is attempting to solve the N-queens problem by filling the board one row at a time. In the figure, the master has closed all avenues to solution after

[3] Later, we'll see how some classes of transformations more specific than $\mathcal{F}$ permit a nagger's solution can be used to advantage under certain conditions.

[4] This particular example does not illustrate all the aspects of transformation tolerated by nagging. Here the nagger's search tree always contains the same number of solutions as the master's. Later we introduce transformations that do not have this property.

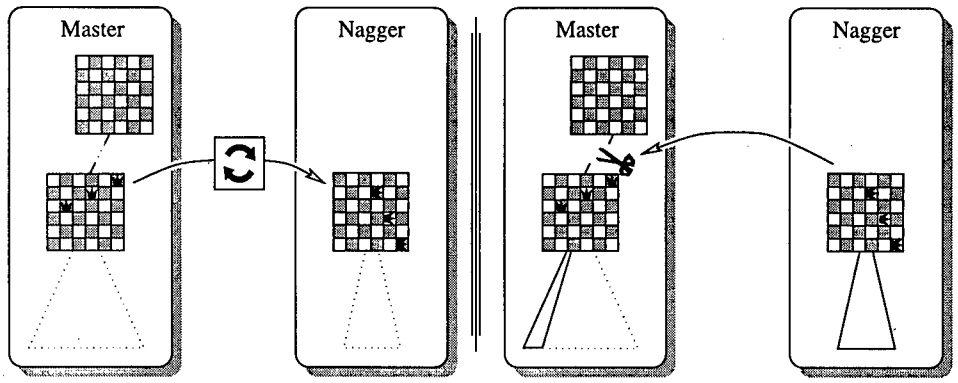

Figure 5.2: Search pruning via nagging. Here, the nagging process transforms the master's search problem by rotating the chess board 90 degrees. Under the same row-by-row search procedure the nagger's search space is much smaller and will be exhausted much more quickly than the master's.

placing only three queens; although no queens are threatened, every space in the leftmost column is. If a nagger transforms the problem via a 90-degree board rotation, its attention is immediately focused on the threatened column. While the master may explore many futile search paths before reconsidering the placement of one of the first three queens, the nagger can be expected to exhaust its transformed search space very quickly. The hope is that the master will explore only a small portion of its own space before the nagger prunes it.

5.2.1 Basic Protocol

In its most elementary form, the nagging protocol is defined by the following three message types:

`idle` When a nagging process becomes idle, it reports to the master with an `idle` message. After sending this message, the nagger waits to be assigned a new search problem.

`problem` The `problem` message is the means by which the master distributes work to available nagging processes. The master is licensed to send `problem` messages to any nagger at any time, but, in practice, they are sent only after one of the following conditions is met:

- The master receives an `idle` message from a nagger.
- The master completes its search of $T(\delta)$ while the nagger is still exploring some $f(T(\delta))$.

The `problem` message specifies a node $\delta \in T$ such that the master has expanded δ but not finished exploring $T(\delta)$. After sending a `problem` message, the master continues its search. When the nagger receives the message, it discards any search in progress, selects a transformation $f \in \mathcal{F}$ and begins exploring $f(T(\delta))$.

`prune` Whenever a nagging process exhausts its transformed search problem without finding a solution, it issues a `prune` message to its master. When the master receives a `prune` message for subtree $T(\delta)$, it knows that further search in $T(\delta)$ would be futile and simply discards any unexpanded nodes in the subtree.

Each type of message also carries a unique time-stamp to prevent the receiver from misinterpreting the state of the sending process.

5.2.2 Properties of the Nagging Protocol

Although the nagging protocol is quite simple, it enjoys properties (stated in the theorems below) that make it attractive as both a general-purpose search pruning technique and a parallelism scheme. These properties contribute to two design goals: similarity to the underlying serial search procedure and suitability for a distributed computing environment.

One advantage of the nagging protocol is that it does not directly affect the master's serial search order. This can be convenient if the master's search is heuristically guided or if the serial search order conveys some measure of optimality (*e.g.*, shortest solutions first). It also bounds the performance of the parallel system with respect to its serial counterpart.

DEFINITION (Myopic Search Procedure). Let F be the set of nodes on the search fringe (the nodes generated but not yet expanded) and let δ be a member of F. A search procedure is said to be *myopic* if the following conditions are met for all F and δ:

- The set of children generated when δ is expanded is a function of δ only.
- If δ is selected from F as the next node for expansion and $F' \subseteq F$ with $\delta \in F'$, then δ must also be selected from F' as the next node for expansion.

The myopic property excludes a number of techniques by which information gained in one part of T can be used to prune or reorder search elsewhere. These techniques include various search pruning mechanisms like intelligent backtracking [9, 58] as well as some search-reordering policies [5]. Some learning schemes (*e.g.*, caching and lemmaizing [89]) also compromise myopia.

When myopia is maintained, nagging exerts a well-defined influence on the search order:

THEOREM 1 (Solution Ordering) *If the master's search procedure is myopic, a nagged search will discover all solutions that are found by an equivalent serial search, and solutions will be found in the same order.*[5]

Theorem 1 does not guarantee that nagged and serial searches discover the same solutions because, in special cases, nagging will find solutions where the serial search will not. Obviously, this can only occur when the serial search is incomplete. For example, nagging can extricate a depth-first search from an infinite subtree as long as transformation makes this subtree finite. Because of this possibility, the potential search reduction achievable through nagging is theoretically unbounded. More generally, nagging will not cause the master to explore more of T than it otherwise would:

THEOREM 2 (Non-Increasing Search) *If the master's search procedure is myopic, then, for any solution δ_s, if δ_s is the i^{th} node expanded without nagging then δ_s will be found before the master expands $i+1$ nodes in a nagged search.*

Of course, this result has only indirect bearing on performance, since nagging overhead changes the average cost of node expansion. There is some risk that nagging-induced overhead will actually increase solution time even when the search space is reduced. As with serial search-reduction mechanisms, the hope is that the savings will outweigh the overhead.

For this reason, the nagging protocol has been designed to minimize overhead, with demand on the master process being given special attention. Under the basic protocol, communication occurs infrequently, about as often as naggers need new subproblems. Additionally, the content of each

[5] Proofs of all theorems given here can be found in [103].

message may be kept reasonably small. The `idle` and `prune` messages require only a time-stamp and an indication of the relevant nagging process. The `problem` message must encode an entire subtree, but, with a little communication in advance, this message also may be concisely represented [4, 26, 87].

Nagging is also designed to inconvenience idle processes rather than busy ones whenever possible. For example, the problem transformation is always computed by an idle nagging process. Likewise, if a nagger is idle, it may have to wait for a new problem from the master, but the master is never required to wait for messages from its naggers. From the master's point of view, nagging is completely asynchronous. This property fosters a form of fault tolerance which makes nagging well suited to a distributed computing environment:

THEOREM 3 (Fault Tolerance) *Theorems 1 and 2 apply even if messages under the nagging protocol are delayed or lost.*

Theorem 3 implies that nagging can even tolerate quiet failure of a nagging process. In fact, if T is finite, then a nagging process will eventually be reintegrated even if its connection to the master is temporarily broken. Interruptions in communication may cause the master process to explore more of T than it would with reliable communication, but it will never fail, overlook solutions or generate invalid ones.

5.3 Nagging in First-Order Inference

Although the N-queens examples given so far have been useful for expository purposes, solutions to this problem are of limited practical value. The richer language of first-order logic makes a much more compelling target for nagging. Since a large number of interesting problems, including N-queens, have obvious first-order encodings, effective nagging in this domain can benefit many applications.

Nagging has been implemented as part of the *Distributed, Adaptive Logical Inference* (DALI) theorem prover. DALI is a search engine for first-order logic based on the model-elimination proof calculus [66]. It was designed as a framework for combining a variety of search reduction techniques, and it features a number of serial performance enhancements in addition to its parallel component.

5.3.1 Proof Calculus

We assume that the reader is familiar with the essentials of first-order logic and theorem proving [10, 22]. Model elimination is a first-order inference procedure that, although not properly a resolution procedure, is closely related to resolution and its variants [65]. Model elimination has been popular among theorem proving systems because its component operations can be implemented very efficiently [100]. Here, we focus on the characterization of model elimination within the *connection tableau* framework [63]. The connection tableau makes explicit much of the structure of model elimination proof objects and simplifies the discussion of problem transformations needed for nagging.

A connection tableau $\Delta = \langle \tau, \mu \rangle$ consists of a finite tree τ along with a labeling function μ defined on the nodes of τ. The function μ associates a literal with each node of τ except the root. This labeling must satisfy two conditions. For each non-root node $n \in \tau$, either $c(n) = \emptyset$ or $\{\mu(n') \mid n' \in c(n)\}$ is an instance of a clause from the theory. Also, for any non-root, non-leaf $n \in \tau$ there must be some $n' \in c(n)$ such that $\mu(n')$ is the negation of $\mu(n)$.

DEFINITION. A tableau *branch* β is the sequence of nodes on some simple path from the root of the tableau to a leaf. The member of β farthest from the root is indicated by $\omega(\beta)$.

DEFINITION. A branch is considered *closed* if it contains nodes n_a and n_b such that $\mu(n_a)$ and $\mu(n_b)$ are logical complements. A tableau is closed when all of its branches are closed. A branch or tableau that is not closed is considered *open*.

Model elimination uses two inference operations, *extension* and *reduction*. Let $\Delta = \langle \tau, \mu \rangle$ be a tableau containing an open branch β.

DEFINITION (Reduction). If there exists a node $n \in \beta$ such that $\mu(n)$ and $\neg\mu(\omega(\beta))$ have a most general unifier, θ, then *reduction* of β by n entails applying θ to every label in Δ.

When it is applicable, reduction effectively closes branch β. Extension creates new, potentially open, branches.

DEFINITION (Extension). If $\{l_1, \ldots, l_k\} = C$ is a clause in the theory and $\mu(\omega(\beta))$ and $\neg l_m$ have a most general unifier, θ, then *extension* of β by $l_m \in C$ results in a new tableau $\langle \tau', \mu' \rangle$. The tree τ' is identical to τ except for the addition of nodes $n_1, \ldots n_k$ as children of $\omega(\beta)$. The new labeling function μ' is defined by $\mu'(n) = \mu(n)\ \theta$ for all $n \in \tau$ and $\mu(n_i) = l_i\ \theta$ for $i = 1, \ldots k$.

The *empty tableau* consists of a single, unlabeled root node. According to the definition above, the empty tableau may be extended by a literal of any clause in the theory. A model elimination proof for theory S is a sequence of inference operations that yields a closed tableau when applied to the empty tableau. Theorem proving in model elimination involves a search for such a proof. Thus, the tableaux are the nodes of the model elimination search tree and the symbol Δ will henceforth be used instead of δ to denote nodes of T.[6] We say that a sequence of operations is a *subproof* for branch β in tableau Δ if the sequence closes β when applied to Δ. A sequence of operations is said to *bind* a variable X occurring in Δ if the substitutions applied by the sequence replace X with a non-variable term or cause X to codesignate with another variable that is distinct in Δ.

5.3.2 Search Engine

Like many other theorem provers built around model elimination, DALI's basic inference mechanism is modeled after the *Warren Abstract Machine* (WAM) [1, 113]. This efficient search mechanism was developed within the Prolog community to support efficient execution of definite-clause programs. Fortunately, since model elimination is operationally very similar to Prolog, it necessitates relatively few modifications to the WAM [100].

The WAM traverses the search space by incrementally modifying a single representation of the tableau. This results in a low node-expansion cost and makes depth-first the most natural search order. Since the infinite search trees common in first-order logic are problematic for simple depth-first search, DALI uses iterative deepening [57]. This entails some duplication of work at each iteration but, in most cases, does not seriously handicap performance [101]. By default, DALI bounds search at each iteration by bounding the height of derived tableaux.

DALI offers some flexibility in its ordering of search within each iteration. Ordinarily it uses Prolog's policy of always choosing the leftmost open branch as the target for extension or reduction. When operating on a branch, reduction is tried first, with reduction by nearer ancestors given

[6] This can be a bit confusing since the tableaux composing the search tree are, themselves, trees of labeled nodes. To help to reduce this confusion, we adhere closely to our chosen notation: T represents the search tree, $\Delta = \langle \tau, \mu \rangle$ stands for the nodes in T, and $T(\Delta)$ denotes a subtree of T.

precedence. If no solution is found via reduction, extensions of the selected branch are considered. Extensions by unit clauses are tried before non-unit clauses, but, otherwise, clauses are simply ordered according to their appearance in the theory.

5.3.3 Nagging Component

Critical to the success of nagging is the design of effective problem transformation functions. In the simple example of Figure 5.2, the "rotation" transformation converts one instance of M:N-queens into a different instance of the same problem, permitting master and nagger to use the same search procedure. Not only does this significantly simplify the implementation, but it also ensures that any improvements made to the serial search procedure will automatically benefit both master and nagging processes.

In a similar manner, we would like DALI's transformations to trade one theorem-proving problem for a different theorem-proving problem that presents a different solution profile under an identical search procedure. Of course, membership in $\mathcal{F}$ means that $f(T)$ must contain a proof whenever T does. Unlike the N-queens example, however, the model elimination proof calculus provides a very rich framework for designing problem transformations.

In practice, membership in $\mathcal{F}$ is sufficient to guarantee correctness but is not sufficient to insure performance improvement. Designing effective transformation functions with a realistic potential for search reduction is a nontrivial task. Consider the two sample model elimination transformation functions, f_1 and f_2. Function f_1 works by modifying the tableau, while f_2 works by modifying the theory, S.

$f_1(T) = T(\Delta')$ where Δ' is the result of deleting the leaf nodes of every open branch in the tableau at the root of T.

$f_2(T) = T'$ where T' is a model elimination search tree with the same root as T, but defined by theory $S' = S \cup C'$ for some new clause C'.

Both of these functions are legal members of $\mathcal{F}$, but neither will contribute to pruning the master's search in practice. Pruning occurs only when a nagger exhausts its search space without finding a solution. A nagger using f_1 would always find a solution (a closed tableau), while the extra clause introduced by f_2 can only increase the nagger's search space when there is no solution to be found.

These functions serve to illustrate two additional properties that are crucial to the design of effective problem transformation functions:

DEFINITION (Informative). A function $f \in \mathcal{F}$ is *informative* if there is a search tree T such that $f(T)$ contains no solutions.

DEFINITION (Reductive). A function $f \in \mathcal{F}$ is *reductive* if there is a search tree T such that $|f(T)| < |T|$.

Function f_1 is not *informative*; function f_2, while *informative*, is not *reductive*.

While the *informative* property is a necessary condition for successful nagging, the *reductive* property is not. If, for example, the master's search procedure is not depth-first, a nagger may be able to exhaust $f(T(\Delta))$ before its master completes $T(\Delta)$ even if $f(T(\Delta))$ is larger. Alternatively, the nagger may simply be running on a faster or less loaded machine. In this paper, however, the reductive property will be taken as a practical requirement for any candidate transformation.

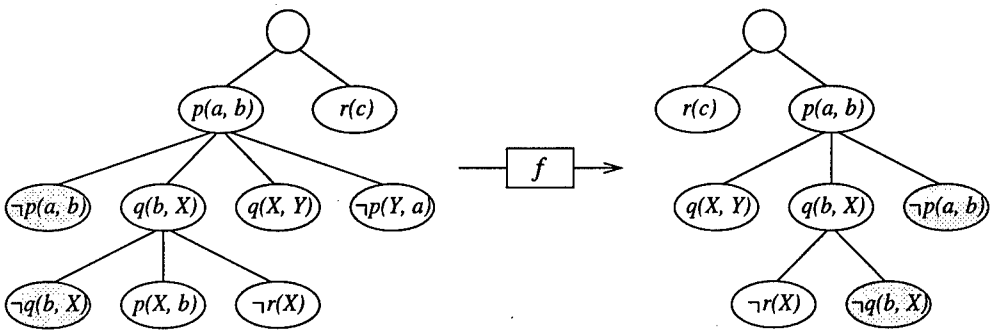

Figure 5.3: Example of transformation under $\mathcal{P}$. Closed branches are indicated by shading. Not only has the left-to-right ordering of the tableau been perturbed, but two of the open branches have been discarded.

Nagging in DALI has focused on two classes of transformations. Functions in the *permutation class* work by reordering and discarding tableau branches. Functions in the *abstraction class* map distinct symbols of the original domain to indistinguishable first-order terms.

DEFINITION (Permutation Transformation). Function f is a *permutation transformation* if $f(T(\Delta)) = T(\Delta')$ where Δ' is identical to Δ except for the possible deletion of the leaves of some open branches and permutation of the left-to-right ordering of children at each node. The set of all such functions f is denoted as $\mathcal{P}$.

Figure 5.3 illustrates the effect of a typical transformation in $\mathcal{P}$. Reordering children is simply a means of altering the order in which the search attempts to close open branches. Deleting a leaf node permits closing of the tableau without a subproof for the truncated branch.

Membership in $\mathcal{P}$ does not insure that a function is informative or reductive. Indeed, both f_1 and the identity transformation satisfy the definition of $\mathcal{P}$. In general, however, functions in $\mathcal{P}$ exploit two opportunities for reducing the size of the transformed search space. The most obvious is the deletion of tableau branches. Tableaux that are distinct in the original search space may have a single, abbreviated representative in the transformed search. Reordering the remaining tableau branches also contributes to search reduction. The order in which branches are selected may have substantial influence on the size of the search space (as evident in the N-queens example of Figure 5.1) without affecting completeness. In general, determining an optimal conjunctive ordering is, itself, a search problem. Naggers using functions in $\mathcal{P}$ operate under the assumption that the master's default ordering is suboptimal. A nagger's permutation of the tableau can be seen as an attempt to find a better ordering.

The definition of the *abstraction class* is a bit more involved. Let $=_a$ be an equivalence relation on the constant and function symbols appearing in the theory, S. Each equivalence class in $=_a$ comprises a set of symbols that may be rendered indistinguishable after transformation. DALI uses a simple syntactic device to enforce this mapping.

DEFINITION (Abstraction Mapping). Given S and $=_a$, the abstraction mapping g_a associates with each first-order formula a set of abstracted formulae. Given formula F, $g_a(F)$ is the set containing all formulae derivable from F via the following:

- Each occurrence of constant symbol c in F is replaced by either $f_{[c]}(c)$ or $f_{[c]}(V)$ where $f_{[c]}$ is a new symbol representing the equivalence class containing c, and V is a new variable.

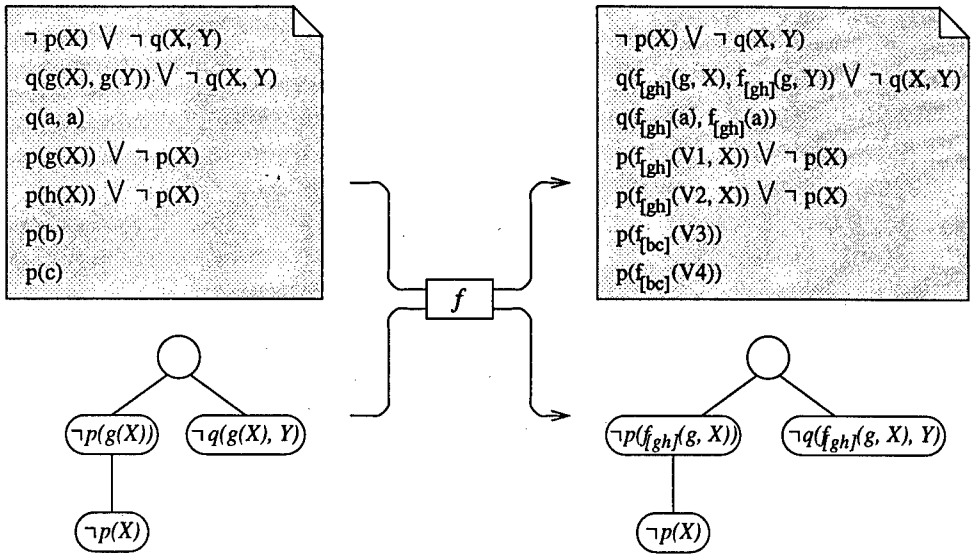

Figure 5.4: Example of transformation under $\mathcal{A}$. Here, the constants b and c are identified, as are the functions f and h. This results in a simplified theory where four of the original clauses collapse to two.

- Each occurrence of function symbol h of the form $h(t_1, \ldots t_n)$ is replaced by either $f_{[h]}(h, t_1, \ldots t_n)$ or $f_{[h]}(V, t_1, \ldots t_n)$ where $f_{[h]}$ is a new symbol representing the equivalence class containing h and V is a new variable.

DEFINITION (Trivial Abstraction). Given abstraction mapping g_a, the *trivial abstraction* of formula F, written $g_{\bar{a}}(F)$, is the member of $g_a(F)$ that introduces no new variables.

The set of abstracted formulae generated by g_a represents a choice of where abstraction might be applied. Intuitively, abstraction discards some information in the original formula. For formula F, selecting among the members of $g_a(F)$ controls where and how much information is lost. If constant b is replaced by a term like $f_{[b]}(V)$, it will match the abstraction of any other constant c for which $[b] = [c]$. If b is replaced by $f_{[b]}(b)$, it will only match $f_{[b]}(V)$ or other occurrences of $f_{[b]}(b)$.

DEFINITION (Abstraction Transformation). Let g_a be an abstraction mapping. Function f is an *abstraction transformation* if $f(T(\langle \tau, \mu \rangle)) = T'$ where, T' is the model elimination search tree defined by some modified theory $S' \in g_a(S)$ and rooted at tableau $\langle \tau, \mu' \rangle$. The labeling function μ' is defined as $\mu'(n) = g_{\bar{a}}(\mu(n))$ for all $n \in \tau$. The set of all such transformation functions f is denoted by $\mathcal{A}$.

Figure 5.4 demonstrates the effect of a transformation in $\mathcal{A}$. Here, constants b and c are identified. As a result, the last two clauses are rendered logically equivalent, and one of them can be discarded without changing the deductive closure of the theory. Similarly, association of function symbols f and h makes two other clauses identical. In general, clause C may be removed from the abstracted theory as long as some other clause C' is retained such that C' is *as general* as C:

DEFINITION (Clause Generality). Clause C' is *as general* as clause C if there exists a substitution θ and an isomorphism g_C from C onto C' such that for $l \in C$, $l = g_C(l)\,\theta$.

This generality relation on clauses is stronger than conventional subsumption [10]. Subsumption would be sufficient if the nagger was only required to find a proof starting from the empty tableau; however, $f(T(\Delta))$ is rooted at a transformed version of the master's tableau. To ensure that $f(T(\Delta))$ contains solutions whenever $T(\Delta)$ does, it is not always permissible to remove all clauses that are logically subsumed.

Like $\mathcal{P}$, membership in $\mathcal{A}$ does not guarantee that a transformation is reductive, since, for example, the definition of $\mathcal{A}$ does not exclude the trivial abstraction. Thus, the abstracted search space may be isomorphic to the original. In general, however, abstraction has power to reduce search by eliminating clauses from the abstracted theory. In Figure 5.4, both the original and transformed theories entail no proofs, but the size of the abstracted search space is only linear in its depth while the original is exponential.

The classes $\mathcal{P}$ and $\mathcal{A}$ prescribe functions of fairly general applicability. Both do, however, rely on some assumptions about the first-order formulation of a problem. The functions in $\mathcal{P}$ transform the search space by interchanging and deleting branches of the tableau. This approach would be of limited value for theories where every clause contains at most two literals. Likewise, nagging with $\mathcal{A}$ would be ineffective for theories having only a small number of distinct symbols. Thus, the applicability of nagging is a function of both the problem and its chosen formulation. The hope is that most natural problem formulations will lend themselves to nagging under $\mathcal{P}$, $\mathcal{A}$ or some combination of the two.

5.4 Refinements to Nagging

The basic nagging protocol is attractive because of its simplicity and its consistency with a distributed model of parallel computation. This framework also admits many natural refinements and extensions that support greater utilization of nagging processes and additional opportunities for search pruning. These refinements fall into two broad categories: completely general refinements applicable to nagging in any search problem and with any transformation function, and refinements specific to nagging in model elimination.

5.4.1 Recursive Nagging

One limitation of the basic nagging protocol is that all nagging processes must communicate directly with the single master process. By design, each nagger imposes only a small amount of overhead on the master, but this centralized approach is inherently inconsistent with an interest in scalability.

Recursive nagging is a strategy for reducing this bottleneck while increasing the effectiveness of individual nagging processes. In attempting to prune the master's search, each nagger must complete its own search problem in a similar or identical domain. This presents an obvious opportunity for naggers to be nagged in turn. If nagging is effective at reducing search on the master, it may be similarly effective at pruning naggers' search problems. Figure 5.5 shows how recursive nagging may utilize a large number of processes without requiring a single administrator to directly control them all. The top-level process acts as the master for a small number of naggers; each non-terminal nagger also serves as the master for its subordinates. These subordinate naggers advance the master's search only indirectly, by speeding the operation of their parent naggers.

All of the transformation functions introduced thus far map between instances of the same problem. This permits uniform operation throughout a nagging hierarchy. While a first-level nagger searches some tree of the form $f(T(\Delta))$, its second-level naggers explore trees of the form $f'(T(\Delta'))$ for $\Delta' \in f(T(\Delta))$. For each problem assigned to the top-level master, a nagger at the first level may undertake several transformed subproblems; for each problem given to a first-level

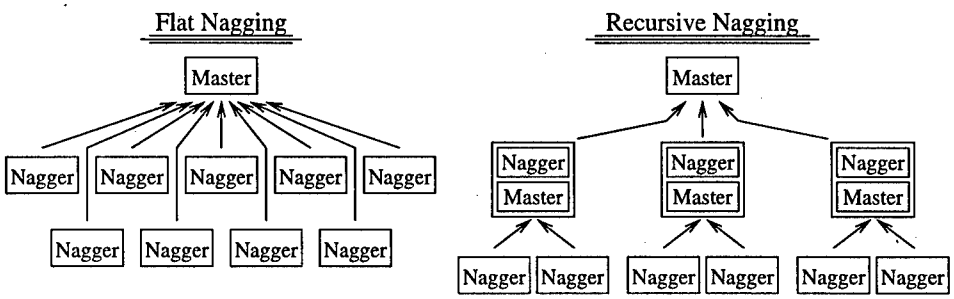

Figure 5.5: Recursive nagging model.

nagger, its second-level naggers can expect to see several subproblems. This nonuniform granularity yields a nonuniform demand for interprocess communication, which can be structured according to the layout of the communication facility. Processes demanding more frequent communication can be placed locally in the distributed computing architecture.

5.4.2 Informed Selection of Nagging Targets

The discussion of nagged subproblems thus far has concentrated on the action of the transformation, with little attention given to choosing an appropriate target subtree. A policy for selecting this subtree is much akin to an OR-ordering heuristic. A perfect mechanism for selecting nagging targets must reject all subtrees that contain solutions. Thus, any reasonable rule for selecting feasible nagging targets must be approximate in nature.

In many cases it is possible to use knowledge about the theory or the transformation to make informed choices about where nagging would be most useful. For example, when a node has only one child, it is generally preferable to nag on the subtree rooted at that child rather than its parent. Both nodes offer similar potential for search pruning, but, as a rule, the child yields a smaller transformed search space. Enforcing this preference can be particularly effective on theories with logic programming qualities, where OR choice points occur infrequently and are embedded in liberal stretches of supporting computation. DALI uses a combination of compile-time and run-time techniques to avoid nagging on a node that has only one child.

5.4.3 Completed Subproofs

Apart from the use of problem transformation functions specific to its operation, there are opportunities to exploit features of the model elimination proof calculus within the nagging protocol. Under both $\mathcal{P}$ and $\mathcal{A}$, master and nagger perform search in closely related domains. For $f \in \mathcal{P} \cup \mathcal{A}$, search in $f(T(\Delta))$ may reveal more information about $T(\Delta)$ than is guaranteed under the definition of $\mathcal{F}$.

Under the basic protocol, naggers reduce the master's search only when they fail to solve their transformed search problems. If a nagger finds a solution, it simply discards it and reports idle. In the general case, there is no simple relationship between solutions to the master's problem and the nagger's transformed problem. For functions in $\mathcal{P}$, however, there are opportunities to exploit solutions found in the transformed space. In Figure 5.6, for example, transformation discards all but the rightmost branch. If the nagger discovers a solution, it has found a subproof for $r(c)$, one of the open branches in the master's tableau. If the master were permitted to use this partial solution, it would not have to re-derive a subproof for $r(c)$.

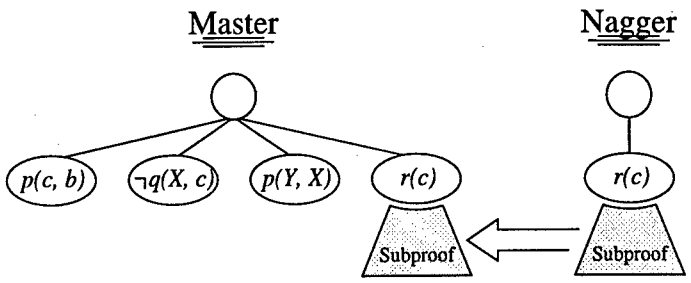

Figure 5.6: The nagger's discovery of a subproof for $r(c)$ may permit the master to avoid proving it a second time.

In general, some restraint must be exercised in grafting a nagger's subproof into the master's tableau. While the nagger searches in $f(T(\Delta))$, the master continues its attempt to close Δ. By the time the nagger completes a subproof, the master will have already applied some inference operations to Δ. The nagger's subproofs will be applicable to Δ, but may not be consistent with the master's current node in $T(\Delta)$. Even when consistent, adopting such a subproof is tantamount to permuting the master's conjunctive goal ordering.

Notwithstanding these difficulties, there are situations where naggers produce unquestionably useful subproofs. For example, the nagger's subproof of $r(c)$ in Figure 5.6 can be applied to the master's tableau without risk of adversely affecting the master's search. This is because the branch it closes shares no variables with the rest of the tableau; it cannot interfere with operations on other branches. DALI uses the following condition to identify subproofs that can be safely used by the master:

DEFINITION (Weak Locality). Let B be the set of open branches in tableau Δ and let $\hat{B} \subseteq B$. A partial proof, p, is a *weakly local subproof* for $\hat{B}$ in Δ if the following are satisfied:

- Subproof p closes all branches in $\hat{B}$ when applied to Δ.
- For any branch $\beta \in (B - \hat{B})$, when p is applied to Δ, no children are added to β.
- The labeling of any $\beta \in (B - \hat{B})$ is changed only by a renaming of variables when p is applied to Δ.

DALI's notion of locality is "weak" because it permits subproofs to bind variables occurring elsewhere in the tableau so long as they do not affect other open branches. One advantage of this class of subproofs is that the master does not have to encourage or even detect weak locality. The nagger may simply check its own solutions *a posteriori* to see if they are weakly local before reporting idle. If a nagger reports that it has found a qualifying subproof, the master can rely on the nagger's partial solution and concentrate on closing the remaining branches. It can be shown that addition of a weakly local subproof has no influence on the closing of other branches. Consequently, the master may defer its integration indefinitely without risk of changing search behavior. When a nagger finds a weakly local subproof, it simply reports the pertinent branches and holds a copy of the solution. The actual subproof is transmitted only if the master succeeds in closing all remaining branches. In this way, the cost of exploiting a nagger-discovered subproof is kept low until it is clear that the subproof is useful.

Exploiting nagger-discovered solutions in this way requires three new messages in the nagging protocol:

`subproof-found` When the nagger finds a solution in its $f(T(\Delta))$, it checks the weak locality conditions. If they are satisfied, the nagger records a copy of its subproof and sends a `subproof-found` message to the master indicating the set of tableau branches it has successfully closed. The nagger retains a copy of its solution in $f(T(\Delta))$ as long as the master continues searching $T(\Delta)$, in case the master needs to integrate this solution into the final answer (see the `local-subproof` message below). Upon receiving a `subproof-found` message, the master discards any attempt it has made to close these branches, and considers them closed as long as it remains in $T(\Delta)$.

`subproof-request` If the master successfully closes all branches not covered by a local subproof, it uses `subproof-request` messages to ask for copies of the missing subproofs from the appropriate nagging processes. The master then searches for solutions to these branches itself while it waits for its naggers to respond.

`local-subproof` When the nagger receives a `subproof-request` message, it transmits its copy of the requested weakly local subproof in a `local-subproof` message. If parts of the desired subproof are held by recursive naggers, they must first be requested and received from these processes. As the master receives local-subproof messages it discards any existing, partial subproofs for the relevant branches and integrates the new ones.

The master's attempt to complete its proof even after issuing a `subproof-request` is in the interests of fault tolerance. Ordinarily the nagger will supply the needed subproof before the master can complete its search. However, if communication with the nagger is interrupted, the master will eventually discover a solution by itself.

Permitting the master to exploit subproofs found by its naggers provides new opportunities for performance improvement. In Figure 5.6, even if nagging never causes the master to backtrack while working on the three leftmost branches, the promise of a subproof for $r(c)$ permits the master to completely avoid search on behalf of the rightmost branch. Previously, the master was assisted only where backtracking was required. On the other hand, while the exploitation of weakly local subproofs may benefit search performance, it compromises the solution ordering property of Theorem 1. A nagger's partial solution may be the result of a different conjunctive goal ordering. By grafting it into the master's tableau, the master's usual search order is violated.

5.4.4 Incremental Search Pruning

Under the basic nagging protocol, pruning occurs only after a nagger finishes its search problem. Under $\mathcal{F}$, this is the point at which the master's and nagger's search spaces are guaranteed to be related. For transformation $f \in \mathcal{P}$, there is a much tighter relationship between these spaces and there is potential for more aggressive search pruning.

Consider the tableau and its transformation given in Figure 5.7. When a nagger explores $f(T(\Delta))$, it begins by trying to close its leftmost open branch. The nagger finds the first applicable inference operation, op_1, performs it, and then goes on to the next open branch. If attempts to close the remaining branches fail, the nagger may be forced to backtrack, reject op_1, and consider other inference operations for its first branch. Information about this rejection of op_1 can be useful to the master process. The master begins its search of $T(\Delta)$ by attempting to close its own leftmost open branch. If successful, it moves on to the second branch, the nagger's leftmost. Like the nagger, the master will first try op_1. The nagger's determination that op_1 can't participate in closing the three rightmost branches implies that it won't lead to a proof of the master's four. In fact, this knowledge may even be of value to sibling and subordinate naggers.

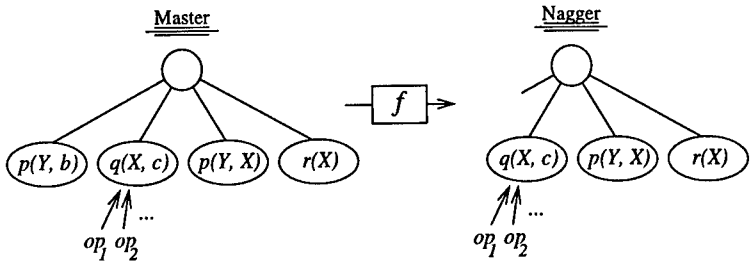

Figure 5.7: Potential for cooperative search pruning. The same inference operations may be applicable to branches in both the master's and nagger's tableaux. Failure of one process to close the tableau using a particular inference operation may have implications for that operation's potential to close the tableau on another process.

An inference operation is considered *infeasible* for tableau Δ if applying the operation to Δ cannot lead to a solution. If operation *op* is known to be infeasible for Δ, any partial proofs in $T(\Delta)$ that contain *op* can be discarded. To facilitate exchange of this finer-grained information about infeasible avenues of proof, the nagging protocol is supplemented with one new type of message:

`infeasible-choice` The `infeasible-choice` message includes a tableau Δ and a set of inference operations $\{op_1, \ldots op_k\}$ that are infeasible for Δ.

Sharing this type of intermediate search information can reduce both master's and nagger's searches in ways not possible under the standard protocol. It does, however, represent a change in the granularity at which nagging takes place. Previously, processes communicated only when a nagger's subproblem was complete. With incremental search pruning, processes may send and receive a number of `infeasible-choice` messages while working on a single subproblem. This additional overhead is somewhat mitigated by the fact that processes never have to wait for these messages. They are simply processed when and if they arrive.

5.5 Refinements to the Search Procedure

DALI's basic model-elimination search has been extended in several respects. These extensions help to reduce search and increase DALI's effectiveness on typical problems. Improvements to the basic search engine also contribute to parallel performance by speeding the nagging processes as well as the master. In principle, these changes to the serial search procedure are orthogonal to nagging. However, certain modifications of the search procedure will jeopardize some of the properties enjoyed by nagging and its refinements. To retain completeness, some mutual accommodation between nagging and serial search-reduction schemes is necessary. On the other hand, there are opportunities for synergy between nagging and the serial search-reduction mechanisms. Exploiting this potential can permit DALI's serial and parallel components to each work more effectively.

5.5.1 Structural Refinements

Although model elimination is refutation complete, there are many restricted forms of the calculus that can improve search performance without jeopardizing completeness. These restrictions forbid

the construction of tableaux that exhibit certain structural properties. Tableaux that exhibit these properties represent redundant lines of reasoning; they may lead to a proof, but there will always be a shorter path to proof elsewhere in T.

A number of structural refinements are applicable to model elimination [100]. Fortunately, many of these are, at least in part, provided as a side effect of DALI's intelligent backtracking mechanism. The only structural refinement that DALI deliberately enforces is the *identical-ancestor* refinement: Any tableau containing a branch with two identically labeled nodes can be discarded.

Checking each derived tableau against this condition would entail significant overhead. To maintain a high inference rate, DALI's uses a lazy approximation of this constraint that has similar search-reducing power. If β is the selected branch from tableau Δ, then $T(\Delta)$ is skipped if the label on $\omega(\beta)$ is identical to the label of some other node in β. This weaker condition pertains only to the selected branch at each tableau, and can be checked quickly as each node is expanded.

Since master and nagger use the same search procedure, any structural refinements applied to DALI affect the operation of both. It can be shown that, for both $\mathcal{P}$ and $\mathcal{A}$, the identical-ancestry refinement may cause the nagger to overlook solutions, but whenever $T(\Delta)$ contains a solution that is permitted under the refinement, $f(T(\Delta))$ will also contain one that is permitted.[7] By itself, this structural refinement simply rejects suboptimal tableaux in a uniform manner. As a result, myopia is preserved and the nagging properties of Section 5.2 are maintained.

5.5.2 Intelligent Backtracking

The basic WAM is built around a *chronological backtracking* mechanism. When a failure point is reached, the search returns to the most recent choice point with untried alternatives. In contrast, DALI uses a form of *intelligent backtracking*, where recent choice points that have no chance of repairing the failure can safely be skipped during backtracking.

DALI's intelligent backtracking component works by monitoring the variable binding structure in the tableau. It is a model-elimination analog of many similar schemes developed for Prolog [9, 23, 58]. Each choice point in the search may be either marked or unmarked. Informally, a mark on a choice point means that the decision made there might be a reason the proof could not be completed. Making that choice differently might change the tableau in a way that permits the proof to succeed. Marking is performed whenever the search is forced to backtrack. After the last applicable inference operation has been tried for branch β, a given choice point is marked if the inference operation applied there created β or changed its labeling. Upon backtracking, unmarked choice points are simply skipped.

This model of intelligent backtracking presents complications when combined with nagging. In Figure 5.8, for example, a serial search might mark the choice points at Δ_2 and Δ_3 while exploring $T(\Delta_4)$. If nagging prunes $T(\Delta_4)$, the master may not have occasion to mark both of these. As a result, the master may fail to consider other tableaux derived from Δ_3 and may miss some solutions. Fortunately, for functions in $\mathcal{P}$ and $\mathcal{A}$ there is a convenient means of updating these backtracking marks whenever nagging prunes the search. The nagging process must exhaust some portion of $f(T(\Delta))$ as a prerequisite to pruning the master's search. The nagger's marking procedure can be extended so that search in $f(T(\Delta))$ provides a marking for Δ and its ancestors. These marks don't affect the nagger's backtracking, but are maintained as a supplement for the master's marking whenever its search space is pruned. Whether pruning through `prune` messages or `infeasible-choice` messages, these nagger-generated marks are sufficient to retain completeness of the master's search.

[7] This is not true for arbitrary first-order transformations. For example, there is a less sophisticated notion of problem abstraction for which the identical-ancestor refinement compromises completeness.

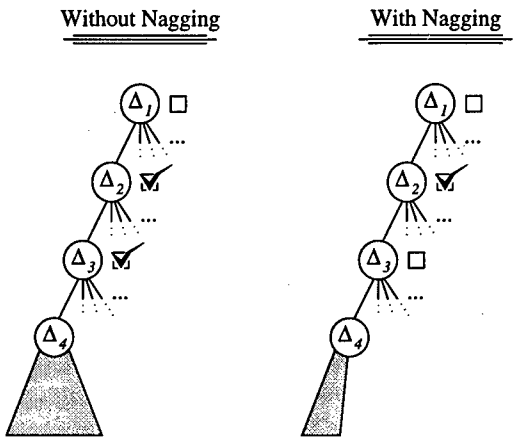

Figure 5.8: Interference of nagging with intelligent backtracking. Nagging-induced search pruning may interfere with the choice-point marking mechanism used in the master's intelligent backtracking procedure. Unchecked, this interference compromises search completeness.

Although DALI's intelligent backtracking scheme can help to significantly reduce search, it violates the definition of a Myopic Search Procedure. Without myopia, the Solution Ordering and Non-Increasing Search properties are no longer guaranteed. Thus, while the master's search remains complete, nagging in the presence of intelligent backtracking may actually increase search. Fortunately, empirical results suggest that nagging and intelligent backtracking usually cooperate quite well.

5.5.3 Subgoal Caching

Intelligent backtracking uses a failure in one part of T to avoid search elsewhere. It exploits the fact that the same tableau features often turn up in many places in T. Caching exploits the same phenomenon. It records features of the current tableau and then looks for these features in subsequent search nodes. If an appropriate match is found, the results of the former search may be used to avoid repeating the same work elsewhere in T.

DALI's caching mechanism is based on the bounded-overhead caching scheme described in Chapter 3. Using caching and nagging together requires some coordination but has the potential for significant speedup (see Section 7.3).

5.6 Empirical Evaluation

In this section we present empirical results that illustrate the effectiveness of nagging. We are particularly interested in determining, first, whether or not the basic nagging protocol is an effective means to perform distributed search, and, second, whether the extensions to nagging outlined in Section 5.4 result in improved performance. To this end, we compare the performance of the DALI system using a simple nagging protocol against an equivalent serial system. We then repeat the comparison using a more sophisticated nagging protocol.

In order for our results to be meaningful, they should be obtained on as wide a range of problems as possible. Our tests employ Version 1.1.1 of the *Thousands of Problems for Theorem Provers* (TPTP) problem set, a collection of 2652 first-order theorem proving problems given in clausal normal form [107]. TPTP problems are drawn from a broad range of domains and cover many

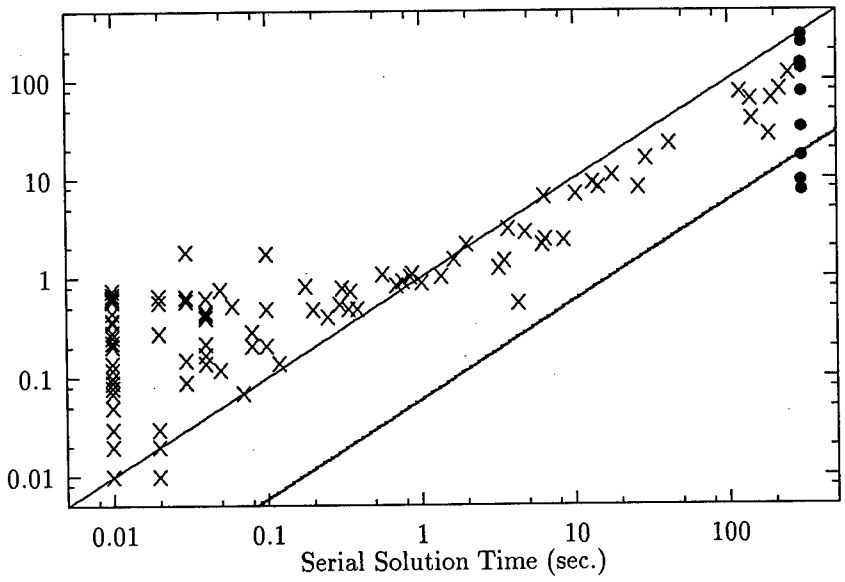

Figure 5.9: Comparison of a 16-nagger parallel system with an equivalent serial system on 400 randomly-selected problems. Datapoints falling below the upper line are faster on the parallel system, with the lower line being the threshold for linear performance improvement. The × datapoints correspond to the 127 problems solved by both systems, while the • datapoints represent the 9 problems solved only by the parallel system. The latter are "censored" in the sense that a lower bound on serial solution time is used as the x-coordinate value, in effect causing these points to appear some distance to the left of where they should be.

theories given in the literature. We used a randomly chosen 400-problem subset of the TPTP for both experiments.

Our first goal is to compare the performance of a simple nagging configuration against an equivalent serial system. Both the serial and nagging systems are configured to use the standard search order, intelligent backtracking, the identical-ancestor refinement, and a 200-element subgoal cache (*i.e.*, all of the serial search enhancements of Section 5.5). The nagging system consists of a master processor assisted by 16 naggers, half using transformations in $\mathcal{P}$ and half using transformations in $\mathcal{A}$. The master process was run on a Sun Sparc 670MP system with 128MB of real memory that was shared by a number of users. Nagging processes were run on equivalent or less powerful systems that were similarly shared. Each system was given a maximum of five minutes of elapsed real time to solve each of the 400 problems in the test set.

Summary statistics for this first experiment are as follows. The serial system solved 127 of the 400 problems within the allotted time. Using the same elapsed time constraint, the nagging system was able to solve the same 127 problems as well as 9 additional problems that were not solved by the serial system within the allotted time.

Of course, number of problems solved constitutes only a relatively coarse measure of performance. Figure 5.9 plots the solution times for each individual problem. Each datapoint corresponds to one of the 136 problems solved by at least one of the tested systems. Nagging system CPU time is plotted (vertical axis) against serial system CPU time (horizontal axis). Datapoints falling below the upper diagonal line represent problems solved more quickly by the parallel system, while points falling below the lower line represent problems solved more than 17 times faster on 17 processors. The 9 "censored" datapoints (appearing as "bullets" rather than "crosses") correspond to those

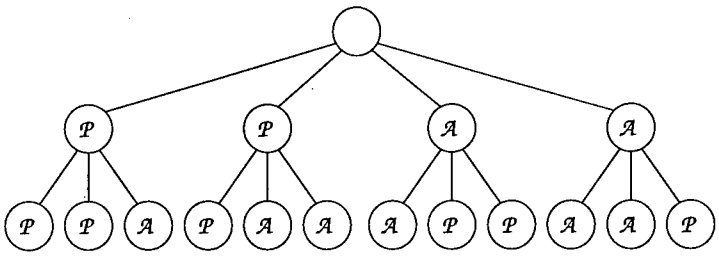

Figure 5.10: Nagging configuration used in second experiment.

problems solved only by the nagging system. Since these problems were not solved by the serial system, we use the CPU resources consumed within the time limit as an optimistic estimate of their actual serial solution time. Graphically, this artificially displaces each censored datapoint to the left of its true position by some unknown margin.

What is, perhaps, most surprising here is that some problems exhibit a performance improvement that exceeds the number of participating processors. Of course, linear speedup is not the theoretical limit for performance improvement under nagging; a nagging system is not simply a direct parallelization of its serial counterpart. As suggested by Figure 5.1, a transformation may give the nagger a substantial short-cut through the search. For this subset of the TPTP, $\mathcal{P}$ and $\mathcal{A}$ must sometimes give the nagger a search space that is more than 17 times smaller than the master's.

A visual inspection of Figure 5.9 reveals that the performance of the nagging system is often worse than that of the serial system on the "easier" problems (informally, those problems requiring less than 1 CPU second to solve on the serial system). This is not surprising since nagging induces some start-up overhead, establishing communication and transmitting the domain theory to all naggers on each problem. On most of the "harder" problems this overhead is outweighed by the search pruning nagging facilitates. Furthermore, the performance improvement on some individual "hard" problems dwarfs the loss in performance on all of the "easy" problems – an effect that is visually obscured by the logarithmic scale used for both axes of Figure 5.9.[8]

Our second experiment compares a more sophisticated nagging architecture with the same serial system using the same 400 randomly selected problems. As in the first experiment, 16 nagging processors were used, half taking transformations from $\mathcal{P}$ and half from $\mathcal{A}$. In this experiment, however, the naggers were hierarchically configured as shown in Figure 5.10. In addition, all nagging refinements described in Section 5.4 are enabled.

Figure 5.11 plots the results of the second experiment. Here, the more sophisticated nagging system is able to solve all problems completed by the naive nagging system in the first experiment, plus an additional 4 problems not solved by either the serial system or the naive nagging system for a total of 13 censored datapoints. When compared to Figure 5.9, the more sophisticated nagging system demonstrates not only a greater advantage on the "harder" problems, but also a smaller performance penalty on "easier" problems. In addition, a greater number of problems are pulled below the upper and the lower diagonal lines.

[8] For example, one censored datapoint shown here displays a speedup of *at least* 40 times with respect to the serial system. The time saved on this problem alone is more than an order of magnitude greater than the sum of the time penalty on the 85 problems where the serial system is faster.

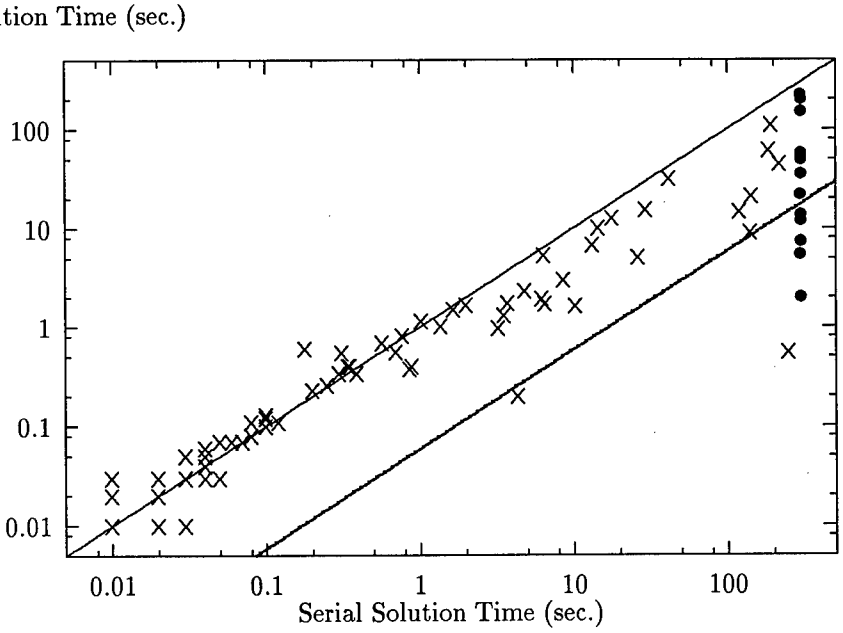

Figure 5.11: Comparison of a more sophisticated 16-nagger system and an equivalent serial system on the same 400 randomly-selected problems. As before, datapoints falling below the upper line are faster on the parallel system, and the lower line demarks linear performance improvement. The × datapoints correspond to the 127 problems solved by both systems, and the • datapoints represent to the 13 problems solved only by the parallel system.

5.7 Discussion

Our work on nagging for model-elimination theorem proving has obvious relevance to other work in parallel search and theorem proving. Since logical specifications are flexible with respect to their order of evaluation, many opportunities for parallelism have been identified [59]. Most work has concentrated on schemes that may be broadly classified as either AND-parallel or OR-parallel.

OR parallelism captures the natural parallelism in the search tree and has been popular among parallel theorem proving implementations [2, 4, 86]. In OR parallelism, the nodes of T are partitioned and each processor is given a subset to explore. It is typical to divide T at its subtree boundaries, but other, less obvious, divisions have been used in an effort to maintain a uniform distribution of work [26].

OR-parallel strategies are attractive primarily because of their potential for low overhead. On shared-memory architectures, there are many opportunities to share data among parallel processors, and highly efficient implementations have been developed [33]. When communication is more expensive (*e.g.*, on a network of workstations), processes can be assigned large portions of the search space and are permitted to explore them independently.

Among OR parallelism schemes, nagging is most closely related to work in competitive OR parallelism [39]. In this model, all processes attempt to solve the same problem, each using a different sound and complete search strategy. When any one of the processes finds a solution, the problem is solved. The hope is that one of these search strategies will lead to a solution quickly. This is similar to nagging, where master and nagger also compete to explore a portion of the search space. The major difference centers around nagging's use of problem transformation. This transformation may give the nagger a substantially reduced search space, but solutions it finds there don't necessarily have relevance to the original problem. A second important difference between

nagging and OR parallelism in general concerns the order in which multiple solutions are discovered. OR-parallel approaches do not usually preserve serial semantics; thus they may generate solutions in a different order than would a serial search procedure. In contrast, as shown in Theorem 1, nagging need not compromise solution ordering.

In the logic programming community, much of the work on parallel search has focused on AND parallelism. This technique emphasizes the parallelism inherent in closing more than one open branch in the tableau. Essentially, the set of open tableau branches is partitioned and each processor is charged with closing some of them. If all processes are successful, it may be possible to compose the subproofs they find into a single, consistent, proof. For these independently-generated subproofs to be composable, they they must agree in how they bind variables.

There are two common approaches to enforcing this inter-process constraint on variable bindings. Under one, AND-parallel processes exchange variable binding information as they perform search [34]; processes are not permitted to make bindings that might disagree with their peers. Other approaches, such as *restricted AND parallelism*, permit parallelism only when conflicting bindings cannot occur [31, 47].

AND-parallel strategies are attractive because of their potential for providing speedup in situations where OR-parallel strategies cannot. For example, many logic programming domain theories are designed to avoid OR choices, so the search tree exhibits little branching for OR parallelism to exploit. In addition, AND-parallel strategies do not in general compromise the order in which solutions are generated, an important factor in the logic programming community where "single solution" problems dominate "all solution" problems, and where a theory's procedural interpretation is typically more important than its declarative semantics alone.

Nagging and AND-parallelism exhibit interesting similarities. If one AND-parallel process determines that its assigned branches in Δ cannot be closed then it is clear that $T(\Delta)$ cannot contain a solution. In this case, it is safe to prune $T(\Delta)$ and free any sibling processes working on other branches of Δ. This is much like nagging under $\mathcal{P}$. The subtree $T(\Delta)$ can be pruned when any process identifies a set of branches in Δ that can't be closed. By composing naggers' weakly local subproofs, nagging also exhibits a component of AND parallelism. The difference between these two techniques stems from a difference in intent. In trying to reach a consistent solution in $T(\Delta)$, AND parallelism must insure that processes agree with respect to variable binding. This type of coordination may preclude the detection of unsatisfiable subsets of the branches in Δ. Since nagging processes do not have to agree or coordinate their activity, they are free to concentrate on showing the unsatisfiability of arbitrary subsets of tableau branches. Thus if solutions are sparse, then nagging is likely to make better use of available computational resources than AND parallelism. However, since naggers make no effort to guarantee that their choices agree with their neighbor's, it is only a product of good fortune when the solutions they find can be composed into a complete proof.

Nagging also bears some similarity to a number of serial search reduction techniques that direct search through some notion of problem transformation. Some of these techniques use the transformed problem as a template for solving the original, searching in the transformed space first and, if possible, coercing solutions found there into solutions of the original [81]. Other approaches focus on transformations that are guaranteed to yield efficiently solvable problems [8, 98, 54]. The transformed problems are used as computationally inexpensive approximations of the original. Typically, the approximation is consulted first and, if it is sufficient to solve the problems, search in T can be avoided.

5.8 Summary

This chapter has described nagging, a new parallel search-pruning technique, and its implementation in DALI, a distributed, adaptive model-elimination theorem prover. We have presented empirical results demonstrating that nagging is effective at reducing search in a variety of first-order domains. We have also shown that a number of refinements to the naive nagging model can enhance its effectiveness in these same domains. Furthermore, nagging combines neatly with a number of serial search reduction mechanisms, permitting us to bring multiple speedup techniques to bear in problem solving.

Nagging, like OR parallelism, requires only brief and infrequent communication, making it particularly suitable for high-latency low-bandwidth distributed systems. Under the right circumstances nagging, like AND parallelism, can even help to improve performance in theories where OR parallelism is ineffective, and needn't compromise the order in which solutions are generated. Another feature that distinguishes nagging from most OR-parallel and AND-parallel schemes is its intrinsic fault tolerance. This property has been particularly valuable when nagging in large, distributed computing environments, where a requirement for complete reliability is unrealistic [97].

The results in first-order inference have been so encouraging that we have begun developing nagging implementations in other domains such as alpha-beta minimax, the Traveling Salesman problem, and learning of Bayesian inference networks. Our continuing work on nagging includes instantiation of the basic protocol in these and other domains as well as further refinement to the first-order model.

Chapter 6

Iterative Strengthening and Anytime Optimization

In order to perform adequately in real-world situations, a planning system must be able to find the "best" solution while still supporting anytime behavior. We have developed a method for incrementally optimizing plans called iterative strengthening *that can be used in many situations where other optimization methods are not appropriate.*[1] *In particular, iterative strengthening supports optimized planning within an "anytime" environment using multiple simultaneous optimizing parameters, and it can be adapted to support inadmissible heuristics and undecidable domains.*

6.1 Introduction

In order to perform adequately in real-world situations, a planning system must do more than simply generate a plan that satisfies the user's goals. In many domains there are almost always multiple solutions to any given problem statement, and the user typically will want the *best* solution (although the criteria for "best" may change from one user to another or one problem to another). Additionally, many domains are time-critical and require support for "anytime" behavior [29]. In this context, an anytime algorithm is one in which a solution is incrementally refined over time; if the algorithm is run to completion it will find an optimal solution, but the user can interrupt it at any point and demand a useful (but not necessarily optimal) solution.

We have developed an algorithm called *iterative strengthening*, a flexible method of producing optimized plans where the user's criteria for optimization may change during the planning session. Iterative strengthening has the following properties:

- the underlying knowledge base is independent of any specific optimizing parameters;
- the method supports multiple simultaneous optimizing parameters;
- users can easily switch between sets of optimizing criteria;
- the method supports optimized planning within an "anytime" environment;
- the method is consistent with Prolog-style inference engines.

[1] This chapter was adapted from work presented in [12, 13, 14].

We have implemented this method within the ALPS system [2] and have tested it in a simplified transportation planning domain with optimality criteria such as total transport time, number of aircraft, and probability of success.

The remainder of this chapter is organized as follows. Section 6.2 presents the iterative strengthening algorithm itself. Section 6.3 describes how the algorithm supports flexible changes to optimality criteria. Section 6.4 discusses the interaction between optimal planning and theorem proving. Sections 6.5 and 6.6 describe how iterative strengthening can be used in situations where the optimality criteria are inadmissible or the domain theory is undecidable. Finally, Section 6.7 summarizes the way that different domain properties impact the efficiency of iterative strengthening.

6.2 The Concept of Iterative Strengthening

Iterative strengthening is an algorithm that can be used to search for an optimized solution in situations where there may be no control over the order of node expansion and in situations where the user may demand an answer before the optimal solution has been found.

Iterative strengthening is related to the concept of *iterative deepening* [57], in which the system searches to a given depth in the search tree for a solution, and if none is found, the system restarts the search from the beginning with a larger depth cutoff. Iterative deepening combines the small memory requirements of depth-first search with the guaranteed termination property of breadth-first search. Two other common variations on iterative deepening are *iterative broadening* [42], which forces backtracking when depth-first search exceeds the allowed number of alternative paths at a node, and *iterative weakening* [83], which is a more general procedure for iterating through alternative search strategies.

The iterative strengthening algorithm first performs an unconstrained search for any satisficing solution to the planning problem. When it finds that solution, it restarts the search, but now constrains the solution to be "better" than the first solution by some "increment", where "better" is measured by an optimization function specified by the user and "increment" is a function applied to the optimization parameters of the current plan. For example, if the goal is to find the plan that takes the minimum time to execute, and if the system has already found a plan that takes n minutes, it will restart the search constraining the new plan to $n - \delta$, where δ is a user-defined constant. The system continues strengthening the optimization parameters until no more solutions can be found; the last solution is the optimal answer.[3]

Figure 6.1 shows a pseudo-code description of the iterative strengthening procedure. It relies on an underlying planner (the `plan` function) whose behavior is minimally specified: the planner must accept parameters of the goal to solve, the current optimality constraints, and the most recent solution to the goal, and must return a solution to the goal that does not violate the constraints if such a solution exists.

Although iterative strengthening may take longer to find the final optimized plan than an algorithm such as A* [46, 79][4] (because of the overhead costs incurred by multiple passes over the same search space), iterative strengthening has the advantage that it can be interrupted at any time after the initial plan is found and will always have a valid plan available for the user. Since this initial plan is found using satisficing criteria instead of optimizing criteria, it is likely that

[2] Iterative Strengthening is fully supported in the Lisp Inference Engine and is partially supported in DALI.

[3] Technically, the last solution is optimal *modulo* δ. All plans with values in $[n - \delta, n)$ are considered equivalent, and the first such plan located is returned.

[4] A* is representative of a general class of heuristic search algorithms. The significant property of A* is that if the heuristic is chosen appropriately, A* is guaranteed to terminate with an optimal solution and is guaranteed not to backtrack. See Section 6.4.

```
begin procedure iterative-strengthening(goal, increments)
    constraints ← φ;
    answer ← plan(goal, constraints, φ);
    if (answer = φ)
        then return("No solution");
        else begin loop
            if (user-interrupt)
                then return("Best solution so far is", answer);
            constraints ← strengthen(constraints, increments, answer);
            new-plan ← plan(goal, constraints, answer);
            if (new-plan = φ)
                then return("Optimal solution is", answer);
                else answer ← new-plan;
        end loop;
end procedure.
```

Figure 6.1: The iterative strengthening algorithm.

iterative strengthening will generate a valid plan significantly faster than algorithms such as A*. In other words, iterative strengthening supports incremental improvements to existing valid plans; it can deliver an initial plan promptly and then spend any remaining time improving it until an optimal plan is discovered or until the available planning time is exhausted.

6.3 Flexibility of Optimality Criteria

One of our goals was to make iterative strengthening as flexible as possible regarding optimization criteria; in particular, we did *not* want to require domain knowledge engineers to write entirely separate sets of planning rules for each type of optimization. There should be a partition between the domain knowledge and the search expansion rules. We have accomplished this flexibility by using two runtime-configurable hooks:

- `opt-eval` is a pointer to a function that evaluates the objective function for a partial plan. It takes as parameters the current values of the parameters to optimize and the current partial plan.
- `strengthen` is a pointer to a function that calculates the new optimization parameters during the next iteration of the iterative strengthening function. It takes as parameters the current values of the optimization parameters, the increments to apply to those parameters, and the last successful plan. The reason for including the last successful plan is that for certain types of optimization we may be able to exploit any "lucky" improvements beyond the current parameters that were discovered in the last plan.

Using these hooks, it is possible to write planning rules for generic optimality functions. Typically, the underlying planner will call the `opt-eval` function every time the current plan has been extended; if the extended plan exceeds the optimization parameters, the planner can backtrack immediately. Similarly, each time a complete plan has been found, the iterative strengthening

module itself invokes the `strengthen` function to further constrain the search parameters for the next iteration.

To optimize on a different set of optimality criteria, it is necessary to change only these two hooks to point to different functions. The underlying knowledge base and set of planning rules need not change at all.

It is possible to optimize over multiple objective functions (for example, optimizing a transportation plan to minimize both the number of aircraft used *and* the flight time), but it is necessary to resolve ambiguities in defining both the `opt-eval` function and the `strengthen` function. For example, if plan A takes 10 hours and uses 5 aircraft, and plan B takes 12 hours and uses 4 aircraft, which plan is "better"? Likewise, once plan B is found, should the strengthening function decrease both the time and the number of aircraft, decrease just time, or perhaps decrease time and increase the number of aircraft (to look for a potentially large improvement in transportation time at the expense of a slightly larger number of aircraft)?

One possible resolution is to distinguish between *major* and *minor* objective functions. We first optimize on the basis of the major parameter, then we start restricting the minor parameters to choose among plans with the same value in the major parameter.[5] In the example above, if we use time as the major parameter, we first search for the fastest plan, and once we find that plan, we search for the plan that uses the minimum number of aircraft among those plans with the fastest time. This approach can be extended to cases of more than two parameters.

The particular implementation of iterative strengthening used by the ALPS Lisp Inference Engine can be described by specifying the definitions of the configuration hooks. The `opt-eval` function recalculates the parameters from the last successful plan (ignoring the current optimization parameters) and applies the increments to those updated parameters. The `plan` function invokes the ALPS inference engine on the top-level goal. ALPS does not directly use the previous answer to guide the search for the next answer; however, since the Lisp Inference Engine is able to preserve the state of the search space from one invocation to the next, ALPS will restrict its efforts to that part of the search space not yet explored. For the transportation domain, ALPS has `strengthen` functions for criteria such as total transport time, number of aircraft, and probability of success.

6.4 Node Expansion Requirements

Planners that are designed to produce optimal solutions typically implement some form of *best-first search*, often based on an algorithm such as A* [46, 79]. In these systems, each node in the implicit search tree of partial plans is associated with a function that measures the "goodness" of the plan so far, along with a heuristic estimate of how good the best complete plan extended from this position will be. At each choice point in the tree, the system compares the evaluation function of all nodes that have been generated but not expanded and selects the best one for expansion. This method has the property that if suitable heuristic functions are used, the first complete plan found is guaranteed to be the best.

In contrast, planners whose underlying inference engines are resolution theorem provers almost always focus on *satisficing* solutions rather than *optimizing* solutions. Rule selection, unification, and backtracking all occur in a fixed order, and only the first solution generated is of interest. Although some implementations allow permuting the order of choice points based on some fixed evaluation function, it is rare to allow suspending one search path, investigating another path, and returning to the first one (a necessary requirement for best-first search).

[5] Recall that the major parameter is optimized *modulo* δ and may produce a large equivalence class of plans with approximately the same value of the major parameter.

One desirable property of iterative strengthening is that it does not depend on the order of node expansion. The underlying planner is free to expand the search space any way it wants to. This makes optimal planning available to a wide variety of planning architectures, including our ALPS system, that otherwise would not be able to address optimization issues.

6.5 Admissibility Requirements

As with the A* search algorithm, the basic version of iterative strengthening requires an *admissible* search heuristic. There are two requirements for an admissible heuristic: it must be *monotonic* and *optimistic*. The monotonicity requirement states that during the course of a search, a maximizing heuristic must never increase and a minimizing heuristic must never decrease. The optimism requirement states that a maximizing heuristic must never underestimate the final value, and a minimizing heuristic must never overestimate the final value. The effect of these two requirements is that as soon as a partial plan violates the current optimization cutoff, the entire subtree rooted at that partial plan can be pruned because all possible extensions of the partial plan are guaranteed to violate the cutoff.

Note that while both the search heuristic used by A* and the optimality heuristic used by iterative strengthening measure the same thing (essentially a prediction of the "goodness" of any complete plan rooted at the current partial plan), the two algorithms use the heuristics in different ways. A* uses the heuristic to stop and restart various search threads, expanding one partial plan while keeping the rest on a priority queue. Iterative strengthening uses the heuristic to prune subtrees of the search space that are guaranteed not to contain the optimal answer.

In many circumstances, iterative strengthening's optimization criterion can be used directly as an admissible search heuristic. For example, if we want to minimize the number of aircraft used in a transportation plan, we can simply use the number of currently allocated aircraft as an admissible heuristic. This heuristic is guaranteed to be both optimistic (it will always underestimate the final number of assigned aircraft) and monotonic (it will only increase as we add more aircraft later in the plan).

Unfortunately, not all optimizing functions can be translated directly into an admissible search heuristic. For example, if we wanted to *maximize* the number of aircraft instead of minimizing them (possibly because using more aircraft may lead to a transportation plan that is more resistant to delays), we cannot use the number of aircraft currently allocated for the search heuristic because it is neither monotonic[6] nor optimistic.[7]

This example illustrates a fundamental difference in complexity between searching for minimizing and maximizing solutions. In the example above, if we are minimizing the number of aircraft and we have already found one solution that uses n aircraft, we can reject any partial plans as soon as they exceed n aircraft because we know it cannot possibly be an optimal plan. On the other hand, if we are trying to maximize the number of aircraft, we cannot abandon a partial plan just because it has less than n aircraft; it may be that the very last step in the plan will require several aircraft that will push the total over n and lead to an optimal plan. This possibility implies that we need to search each partial plan to completion to decide whether it is better than the current optimal plan. To state it another way, an admissible search function allows us to prune the search space as soon as the current optimal plan is exceeded, while with inadmissible functions we must continue to search until the search space is exhausted.

[6] It is monotonically *increasing* for a *maximizing* function, which is not allowed.

[7] It turns out that in a simplified domain where each cargo requires exactly one aircraft, it actually is possible to define an admissible heuristic by maximizing the number of aircraft currently assigned *plus* the number of cargo units currently *un*assigned. But in general, the inverse of an admissible heuristic is often not admissible.

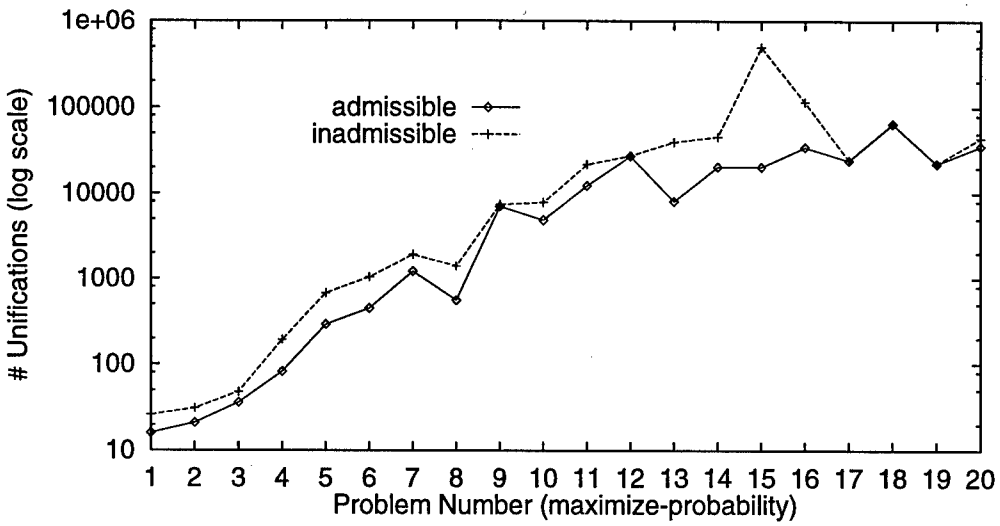

Figure 6.2: Admissible vs inadmissible heuristics for iterative strengthening. Problems are arranged in approximate order of increasing difficulty for naive non-optimized search.

Even though it may be much more expensive to search for an optimal plan using inadmissible optimization criteria, there may be situations where it is necessary. The iterative strengthening algorithm can be easily extended to support search for inadmissible heuristics, but at a substantial runtime penalty. The basic concept of iterative strengthening remains the same: the system finds the first solution with an unconstrained optimization parameter. It then strengthens the constraints on the optimization parameter based on the `strengthen` function. But when the underlying planner searches for subsequent solutions, it will test the constraint values only after it has found a candidate plan, rather than use the constraints as a threshold cutoff to force backtracking as soon as a search path exceeds the optimization parameter.

Figure 6.2 shows the effect of using an inadmissible heuristic on a suite of 20 transportation scheduling problems. In both instances the optimality criterion is to maximize the probability of success of the resulting plan; the only difference is that in one set of trials the heuristic was encoded as an inadmissible heuristic. In cases where the first plan found happens to be the optimal plan, the admissibility of the heuristic does not make any difference. However, in several cases the inadmissible version required 5–25 times as much effort to find the optimal plan.

These results indicate that end users should choose their heuristics carefully and must be prepared for significant performance penalties if they select inadmissible heuristics. On the more positive side, most other optimizing algorithms cannot use inadmissible heuristics *at all*, and those that can will necessarily be forced to pay the same performance penalty since it is inherent in the problem. And even more encouragingly, the admissibility of the optimality heuristic has absolutely no impact on finding the first satisficing solution, so iterative strengthening can still be used effectively as an anytime algorithm even with inadmissible heuristics.

6.6 Decidability Requirements

In order to guarantee an optimal solution, iterative strengthening requires that the domain theory and the underlying planner are *decidable*: given any query in the domain language, the planner

must either return a valid plan or report failure. Decidability is required because the final step of the iterative strengthening algorithm involves a search to determine that there are no superior plans. Unfortunately, many domains are undecidable with resolution theorem provers (for example, most encodings of recursive rules and frame axioms in situation calculus violate decidability).

There are two ways to get around the decidability requirement. One is to sacrifice completeness by enforcing a limit on each iteration of plan improvement based on how long it took to find the current plan; this limit could be expressed as search depth, number of nodes expanded, or CPU time. If the search exceeds the limit, the planner will report failure and return the last successful plan as the optimal answer (even though there may have been a better plan that was not found).

The second method is even simpler: since iterative strengthening is designed to be an anytime algorithm, termination conditions may not be terribly important. The user can interrupt the system at any time and demand the best answer so far; eventually, the user will get tired of waiting and decide that the current answer is good enough.

Neither of these methods is particularly satisfying, since the user will never know whether the answer is *really* the best. However, as with admissibility, the inherent difficulty of the problem means that no other algorithm can expect to do any better, and these methods allow iterative strengthening to perform in situations that most other optimizing algorithms cannot handle at all.

6.7 Discussion

The appropriateness of iterative strengthening depends on properties of the domain, the application, and the implementation.

Although iterative strengthening can be used in any domain (possibly using the extensions above to overcome admissibility and decidability requirements), it is more efficient in some domains than in others. Specifically, iterative strengthening will perform best in domains with the following properties (listed in decreasing order of importance):

1. The solution space is sparse with respect to unique optimizing function values, relative to the granularity of the `strengthen` increment size. A consequence of this property is that the iterative strengthening algorithm will need to loop only a small number of times to progress from the first satisficing solution to the final optimal solution.

2. The optimality function is admissible. A consequence of this property is that the theorem prover can backtrack and the search space can be pruned as soon as the current optimality parameters have been exceeded.

3. The domain theory is decidable. A consequence of this property is that the iterative strengthening algorithm will terminate with an optimal answer without sacrificing completeness or correctness.

4. Changes to the optimality evaluation become incrementally smaller as a plan is constructed. This means that backtracking and pruning can occur early in the search during each iteration (and hence a larger subtree can be pruned), which helps only if the optimality function is admissible.

5. The solution space is dense with respect to unique solutions. In this case, an initial satisficing plan can be found rapidly. However, note that a dense solution space will impede optimal search unless those solutions are clustered around sparse optimizing function values.

If all five of these properties hold, then iterative strengthening will perform almost as well as satisficing search. At the other extreme, in the worst scenario (an inadmissible optimality function and a large number of solutions all with unique optimizing values), iterative strengthening may perform even worse than exhaustive search because it will not prune and will search several areas of the search space multiple times.

Iterative strengthening is particularly appropriate in applications where anytime behavior is desired. As discussed above, if the implementation of the underlying inference engine is such that search control is fixed and cannot be altered, then iterative strengthening may be the only feasible method. Also, if the optimality constraints are inadmissible or the domain theory is undecidable, iterative strengthening may again be the only choice.

6.8 Summary

The basic ideas behind iterative strengthening are not new; they are closely related to the general technique of *branch and bound* [61, 79], which has been used for many years. Our contributions are to offer a particular formalization of this technique, to analyze the properties of this formalization under various situations, and to demonstrate the usefulness of this method in a specific implementation within the ALPS system. We have shown how iterative strengthening can be modified to deal with inadmissible optimality criteria and undecidable domain theories that are typically excluded by other methods, and we have discussed the tradeoffs involved in these modifications.

Chapter 7

Combining Multiple Speedup Techniques

This chapter describes the effects of applying multiple speedup techniques simultaneously in ALPS. Our experiments indicate that combining techniques can produce synergistic effects both by enhancing the speedup properties of single techniques and by decreasing the overhead cost associated with single techniques.[1]

7.1 Introduction

Speedup learning techniques are rarely studied in combination. When studied individually, it is difficult enough to tell whether a given speedup technique's advantages outweigh the problems it introduces. For example, while the use of EBL may provide some reduction of search, indiscriminate application may also entail some increase in search. As noted previously, it is also difficult to draw reliable conclusions about the performance effects of a single speedup learning technique from experimental data. These problems are only compounded by conflating effects of multiple techniques.

The message of this section is that speedup techniques show even greater strength in combination than their individual performance might imply. We base this observation on an extension of the empirical evaluations described in previous chapters that combine caching with the EBL*DI algorithm, the nagging algorithm, and the iterative strengthening algorithm.

7.2 Combining Caching and EBL

In this experiment, we performed four trials using four distinct configurations of the same theorem prover. For each trial, the theorem prover performed depth-first iterative-deepening with an increment of 1, and was therefore emulating the exploration order of breadth-first search. Each trial consisted of one or more passes through the 26 randomly ordered blocks world problems used previously (Experiment 1). Each problem was solved once by the control system in order to determine a difficulty parameter e_{bfs}. For each trial, we fixed a maximum resource limit of 600,000 nodes searched per problem.

In the first trial, we measured the performance of the non-caching, non-learning, iterative-deepening theorem prover. As before, we used the regression slope obtained from this trial as a base

[1] This chapter is adapted from work presented in [90, 12].

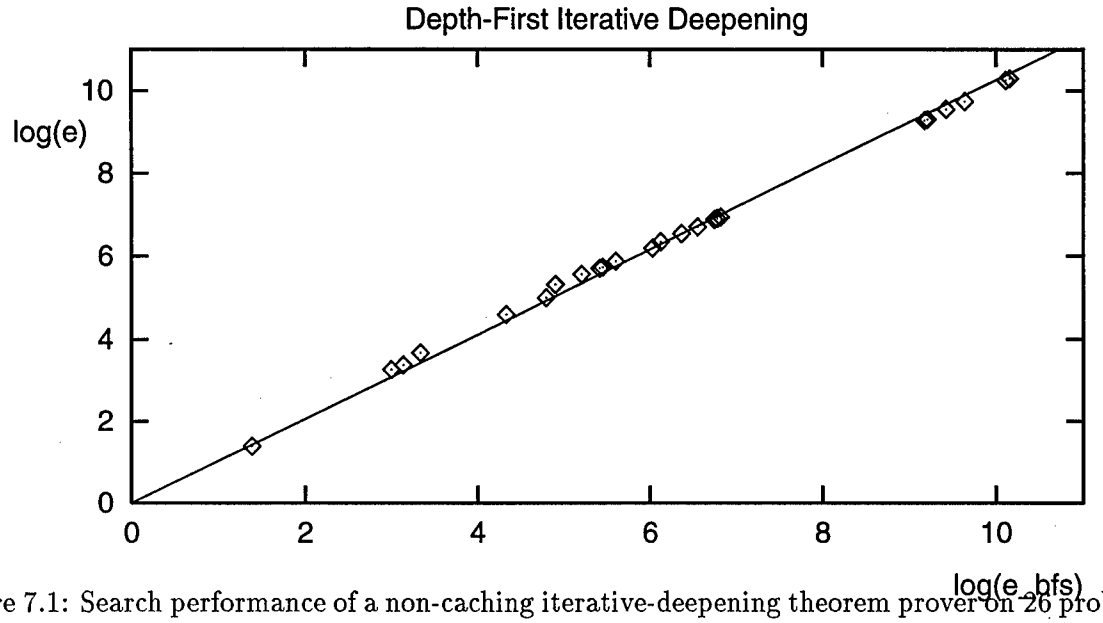

Figure 7.1: Search performance of a non-caching iterative-deepening theorem prover on 26 problems from a situation-calculus domain theory.

value for comparison with the other systems. In the second trial, we added an LRU success/failure cache of 45 entries to the same system used in the first trial. In the third and fourth trials, we measured the performance of the same theorem prover augmented with an EBL*DI learning element and then with both an EBL*DI learning element and an LRU bounded-overhead cache. For each trial, we analyzed the resulting data using the simple one-parameter linear regression model of Equation 3.3. The slopes obtained indicate the relative sizes of the search space explored for the different theorem prover/cache combinations. Slopes significantly smaller than the base value obtained in the first trial indicate an overall reduction of search space explored.

Figure 7.1 illustrates the search performance of the base system (compare with the time performance of Figure 3.1). All 26 problems were easily solved within the resource limit (in fact, all problems are solved searching less than 30,000 nodes). The computed regression slope and standard error, $\log(b) = 1.026 \pm .004$, serve as a basis of comparison for the other systems tested in subsequent trials.

Note that the computed regression slope implies that this system explores relatively more nodes than the control breadth-first search theorem prover, which would yield a slope of exactly $\log(b) = 1$ when measured against itself. While this comparison is invalid (the two systems' node expansion costs c are not even roughly equivalent), the increase in nodes explored is as expected, given that the system is performing iterative deepening with an increment of 1. Depending on the problem population, increasing the increment value may substantially reduce the computed regression slope.

Figure 7.2 shows the search performance of the second trial (bounded-overhead LRU caching system with a cache size of 45). The computed regression slope and standard error in this case is $\log(b) = .902 \pm .007$, indicating significantly fewer nodes are explored by the caching system than the base system of Figure 7.1. While the caching system's overhead will increase the node expansion cost c to some small degree, efficient indexing strategies combined with the relatively small cache size allow us to consider the respective c parameters to be roughly equivalent, enabling direct comparison with the base system's computed regression slope.

By comparison, an infinite-size (*i.e.*, unbounded overhead) caching system yields a computed

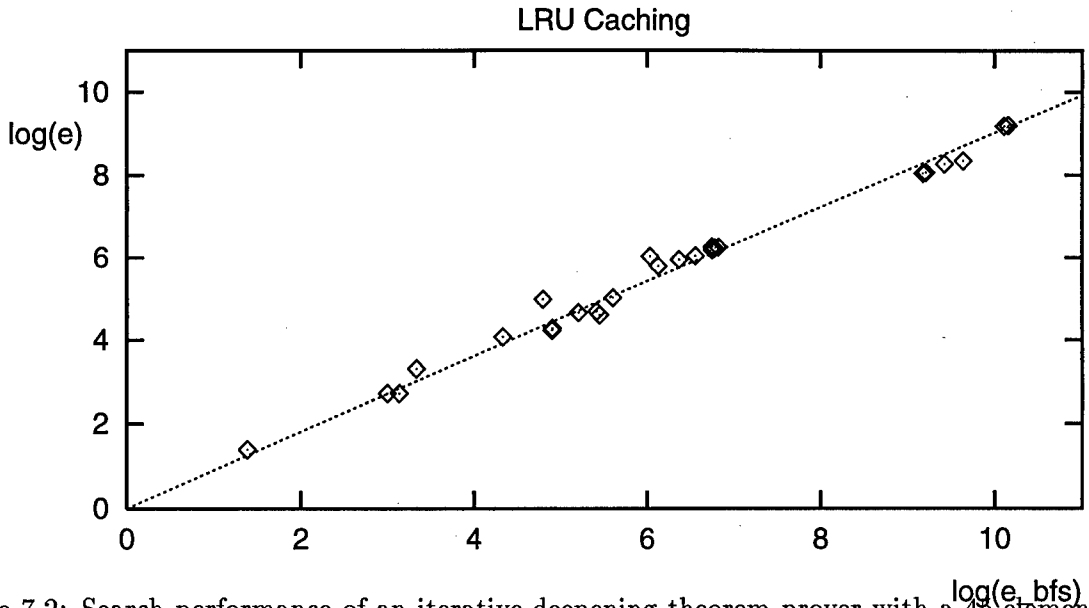

Figure 7.2: Search performance of an iterative-deepening theorem prover with a 45-element LRU cache on 26 situation-calculus problems.

regression slope and standard error of $\log(b) = .849 \pm .011$. However, after solving all 26 problems, the unbounded-overhead system contains a total of 15,447 entries (only 2,033 of which served to provide cache hits at some subsequent time); indexing into a cache this size each time a node is explored will have a large effect on c making direct comparisons with the base system untenable.

Intuitively, the effects of caching are clearly visible when comparing Figure 7.2 directly with Figure 7.1. Certain problems are helped by the presence of cache entries, and datapoints corresponding to such problems shift downwards in Figure 7.2 (recall that the cost of solving any given problem with the control system is invariant, thus datapoints can never shift left or right). By minimizing the sum of the squares of the errors, linear regression provides a good estimate of the slope over the entire problem distribution. As the datapoints spread downwards, the regression slope decreases, reflecting the need to search fewer nodes (on average) over all problems in the population.

In the third trial, we measured the performance impact of the EBL*DI algorithm. Since this is critically dependent on which problems are used in constructing new macro-operators, we altered the experimental procedure slightly to control for this parameter. We performed 20 passes over the 26 problems, each time selecting two problems as training examples and measuring performance of the original domain theory plus the two new macro-operators on the remaining 24 problems. On eleven passes, all 24 problems were solved within the resource limit, while on the nine remaining passes some of the problems were not solved within the resource bound. For the nine incomplete passes, we made (optimistic) estimates of search space explored by treating unsolved problems as if they were solved after exploring the entire resource limit.

When analyzed individually, the regression slopes for complete passes ranged from a low of $\log(b) = .745 \pm .061$ to a high of $\log(b) = 1.250 \pm .074$ (for incomplete passes, these ranged from $\log(b) = .774 \pm .071$ to $\log(b) = 1.334 \pm .096$). Ten of eleven complete passes searched significantly fewer nodes than the base system, while only two of nine incomplete passes did so (even though these are optimistic estimates of performance!). A somewhat more useful analysis is shown in Figure 7.3; all 480 datapoints obtained in 20 passes over 24 problems are plotted and analyzed

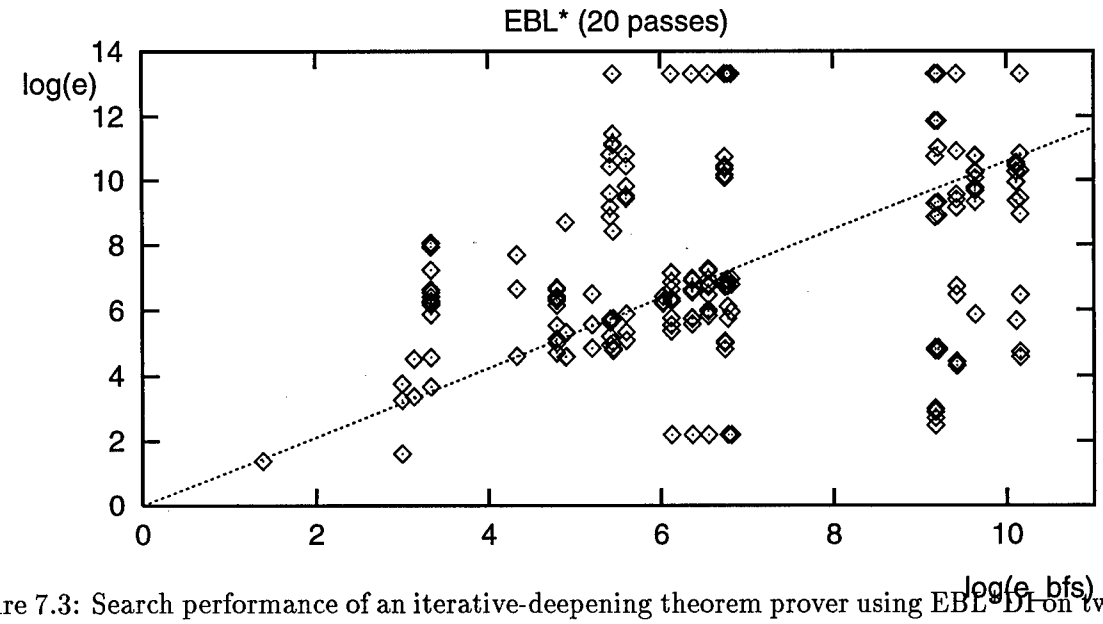

Figure 7.3: Search performance of an iterative-deepening theorem prover using EBL*DI on two randomly selected problems for the remaining 24 situation-calculus problems. 20 trials, 480 datapoints; multiple datapoints may coincide in the plot.

together (note that the computed regression slope obtained here is directly comparable to the computed regression slopes for single trials, while the standard error values are not).

As with caching, the effects of learning are clearly visible in the plot. Some problems are helped by the new macro-operators; their corresponding datapoints have shifted downwards. Other solutions are less efficient with the additional macro-operators; their corresponding datapoints have shifted upwards. The computed regression slope and standard error for the collected trials, which represents the average expected search performance over the entire problem distribution, is $\log(b) = 1.058 \pm .019$. This (optimistic) estimate of overall search performance factors out exactly *which* problems are selected for training, indicating that using this particular EBL algorithm and learning protocol is not a good idea unless one has some additional information to help select training problems.

A similar procedure is used to measure the performance of the combined EBL*DI and bounded-overhead caching system. Each pass in this trial used the same randomly selected training problems as in the last trial: all 24 problems were solved within the resource bound on each and every pass. Here, the individually analyzed regression slopes ranged from a low of $\log(b) = .666 \pm .050$ to a high of $\log(b) = 1.244 \pm .054$. Seventeen of twenty passes performed less search than the base system of Figure 7.1. The combined 480 datapoints are shown in Figure 7.4; the computed regression slope and standard error are $\log(b) = .896 \pm .014$. This result implies that, independent of which problems are selected for learning, the use of EBL*DI and a fixed-size LRU caching system will search significantly fewer nodes than the base system tested previously.

There are several observations we can make about the results reported here.

1. These results reflect reductions in search space explored and not *necessarily* improvements in end performance when measured by elapsed CPU time. Of course, savings in search space explored usually translate into lower elapsed times, but this is highly dependent on system implementation (*i.e.*, the actual value of c in our model).

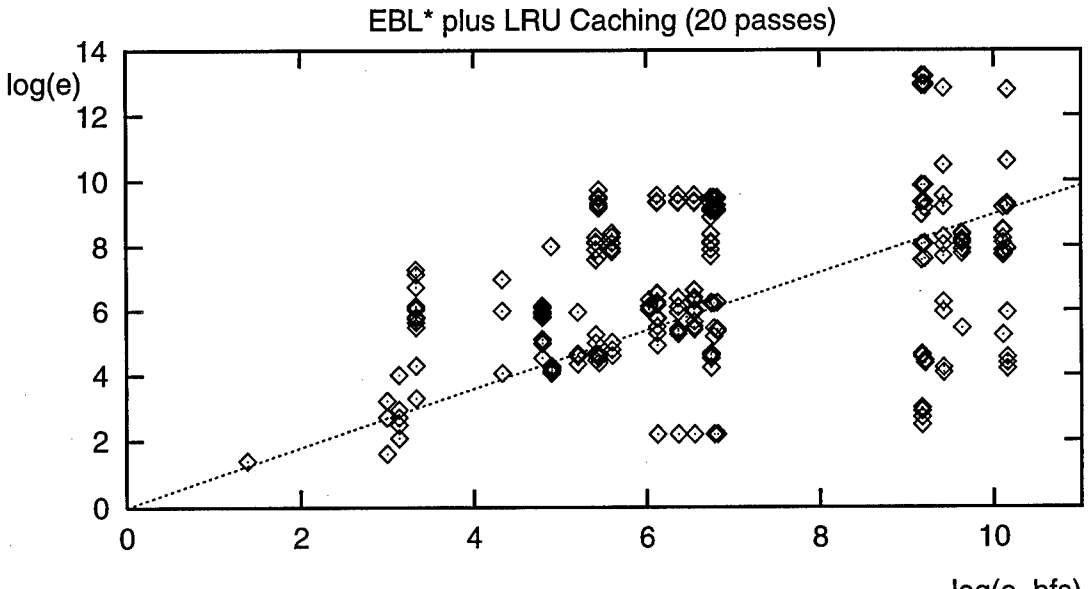

Figure 7.4: Search performance of an iterative-deepening theorem prover using a 45-element LRU cache and EBL*DI on two randomly selected problems on the remaining 24 situation-calculus problems. 20 trials, 480 datapoints; multiple datapoints may coincide in the plot.

2. The search reductions obtained by an unlimited size caching system ($\log(b) = .849 \pm .011$) reflect the theoretical upper bound on the search space reductions attainable via caching. These reductions are simply wishful thinking, since they can only be achieved by adding an unbounded overhead. Fixed-overhead caching is a practical compromise; it carries some limited performance penalty (cache overhead) and delivers some portion of the speedup attained by unbounded-overhead caching.

3. The use of EBL*DI alone under these experimental conditions runs afoul of the utility problem. While the results obtained on some individual passes are encouraging, returning better reductions in search than even the unbounded caching system, they represent a best-case scenario. The penalty imposed for badly-chosen training problems makes unguided use of EBL*DI unacceptable in the limit. We might well draw a different conclusion if we had some more informed way of deciding what to learn, managing what has been learned, or if we were to learn from a different number of problems.

4. Finally, the most striking result is that the combined EBL/caching system not only produces greater search reductions than the (optimistic) estimates for EBL alone, but on average achieves practically the same search reduction as the unbounded-overhead caching system. Given that the EBL/caching system displays bounded overhead (*i.e.*, its c parameter is dominated by the unbounded-overhead system's c parameter), we can conclude with confidence that it will outperform a similarly implemented unbounded-caching system.

Why do EBL*DI and subgoal caching work so well together? EBL*DI, like any EBL algorithm, introduces redundancy in the search space and therefore suffers from the utility problem, which, loosely stated, results from backtracking over these redundant paths. Success and failure caching both serve to prune redundant search, by recognizing the path as either valid or fruitless. Thus caching can work to reduce the utility problem, resulting in greater average search reductions.

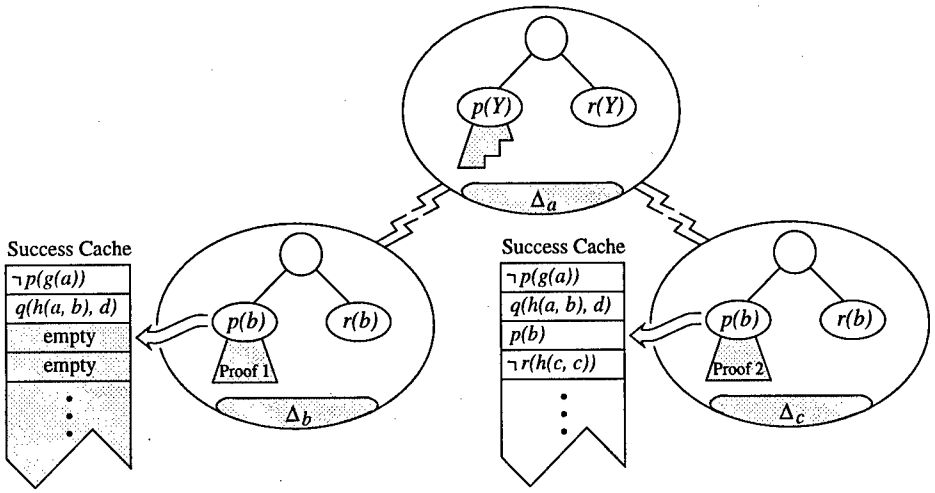

Figure 7.5: Potential for redundancy control through success cache. When a repeated subproof for the same branch is detected backtracking is performed immediately.

This effect is clearly visible when comparing Figures 7.3 and 7.4; problems below the regression line occupy roughly equivalent positions in both plots. Yet problems adversely affected by the presence of learned macro-operators in Figure 7.3 (datapoints above the regression line) are not affected nearly so much when caching is enabled (Figure 7.4). This is one example of a kind of *speedup synergy* that occurs when applying multiple speedup learning methods. Here, one technique (caching) mitigates a flaw in another technique (the EBL utility problem). Another example of speedup synergy arises when combining success and failure caching, as described in Section 3.3.5.

7.3 Combining Caching and Nagging

DALI is able to use the same general types of caching schemes that are available to the Lisp Inference Engine. DALI exploits cached search information in three ways. Through *success caching*, it records the labeling of branches that have been successfully closed. If a similarly labeled branch is encountered elsewhere in the search, the cached record of prior success may, under appropriate conditions, be used to avoid re-deriving a subproof. DALI's *failure cache* is used to record branches for which no subproofs were found. If a matching branch is encountered later, the search engine may backtrack immediately. Finally, DALI's *redundancy avoidance* cache capitalizes on a secondary use of success cache entries. For many theories, there is more than one way of proving the same thing. In Figure 7.5, for example, the $p(Y)$ branch has two proofs that both bind Y to b. When the leftmost branch is closed in node Δ_b, a success-cache entry is made for $p(b)$. The binding of Y to b is implicit in the cached label. In node Δ_c, the variable Y is again bound to b via a different subproof for the same branch. Insertion of this new success into the cache finds $p(b)$ already present. To exploit this, DALI retains information about which branches are responsible for each success-cache entry. Each time a success pattern for branch β is inserted, the caching mechanism checks to see if that pattern has already been inserted on behalf of β. If it has, the search immediately backtracks to find a different subproof for β. This policy is much like the *anti-lemmata* used in SETHEO [62].

In all of its caching schemes, DALI uses an approximate mechanism for remembering and matching tableau branches. Instead of storing the labeling of all nodes on the branch, it records only the label of the leaf. This relaxes the conditions necessary for matching a cache entry and is just as accurate for Horn-clause theories. For non-Horn theories, it is easy to enforce a set of

sufficient conditions for when this leaf label is a sufficient basis for a match. Unfortunately, these conditions forbid the use of failure caching when non-Horn clauses are present.

Using caching and nagging together requires some coordination. For example, when nagging prunes a subtree from the master's search, it interrupts the master's attempt to close some branches of the tableau. Even though the master may not yet have found subproofs for these branches, the failure caching mechanism must be forbidden from recording them as branches that could not be closed.

In general, caching schemes compromise myopia because they use the history of the search as a predictor of future success and failure. Naturally, their effects depend on the population of the cache and on what portions of T are explored first. Unfortunately, this means that adding nagging to a caching system may actually degrade performance. By pruning the search in one part of T, nagging may deprive the cache of some of the search results it would have otherwise learned. The absence of these cache entries may seriously impair the search in subsequent parts of T.

Mitigating this problem to some extent is the fact that each process may manage its cache independently, populating it with entries specific to its own experience.[2] In fact, differences between the caches of each parallel process may be desirable or even necessary. When nagging under $\mathcal{A}$, for example, the nagger's domain theory differs from that of its master. Consequently, its cache entries are not compatible with the tableaux generated by neighboring processes. More generally, the cache entries of one process may be useful in reducing its own search but may be significantly less useful in assisting the search in some different, transformed problem. By permitting each process to maintain its own cache, each is given the opportunity to populate it with entries that will be most useful against its particular transformed problems.

7.4 Combining Caching and Iterative Strengthening

One disadvantage of the iterative strengthening optimization technique presented in Chapter 6 is that, as with all iterative algorithms, it spends a significant percentage of its time searching areas that have already been covered during a previous iteration. This redundant effort can be significantly reduced through the use of failure caching. If a particular subgoal has been exhaustively shown to fail, that result can be stored in a cache; when that same subgoal is encountered on the next iteration, the planner can retrieve the failure from the cache in constant time and backtrack immediately.

We tested this method in the ALPS Lisp Inference Engine by running iterative strengthening on the same set of problems as in Figure 6.2, using a fixed-size cache with a least-recently-used replacement strategy. For this experiment, we disabled ALPS' ability to retain state space information between iterations in order to provide a fair comparison for other systems that do not have this feature.[3] We tested several cache sizes, ranging from 0 elements to 1000 elements. The results indicate that failure caching and iterative strengthening work very well together, and that caching has the most benefit on the largest problems (in certain cases there was a sixfold improvement). Figure 7.6 illustrate these results, plotting number of unifications needed for each problem using different cache sizes.

[2]This is in contrast to the typical use of caching in a memory system, where it is necessary to insure cache consistency between parallel processes. When performing caching in combination with nagging, master and nagger do not need to worry about the possibility that their cache contents differ.

[3]For this experiment, ALPS was run using an iterative deepening breadth-first search strategy with a depth increment of 1. Figure 7.6 does not distinguish between the benefits due to cache hits on iterative *strengthening* iterations and the benefits due to cache hits on iterative *deepening* iterations.

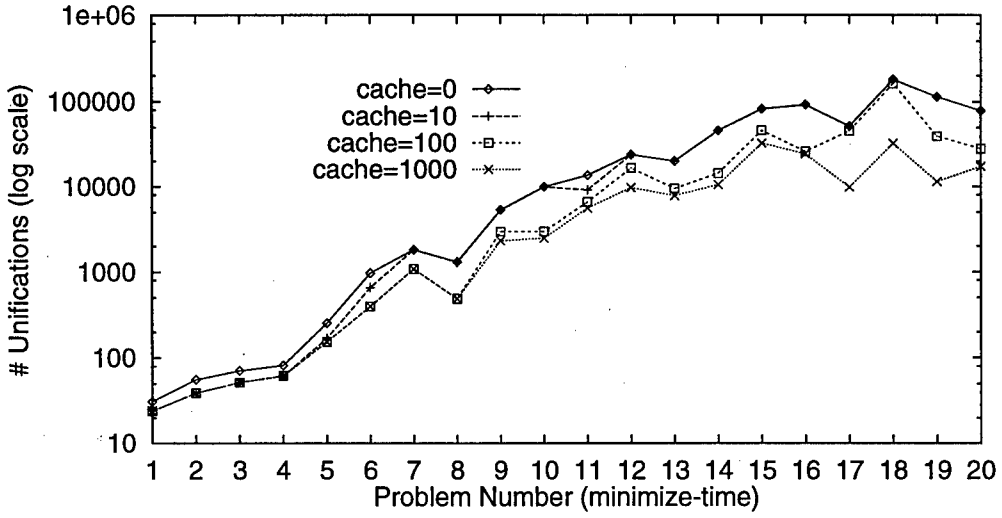

Figure 7.6: Benefits of caching for iterative strengthening. Problems are arranged in approximate order of increasing difficulty for naive non-optimized search.

However, while this type of analysis shows the potential benefit of caching, it does not necessarily demonstrate any practical benefit. When we measure how much processing time is actually saved by caching, Figure 7.7 shows a much different picture. For this particular implementation and domain theory, the overhead for caching erased almost all savings in unification: for many of the benchmark problems, the actual runtime was significantly *slower* with failure caching than without. This slowdown is because most of the cache entries for this domain are very large and expensive to search. In a subsequent experiment, we carefully analyzed the frequency of cache hits for each domain predicate and designed a customized caching strategy that cached only those predicates that are known to have low overhead and high hit probability in this particular domain. With this custom cache, ALPS achieved a modest runtime speedup of approximately 5%.

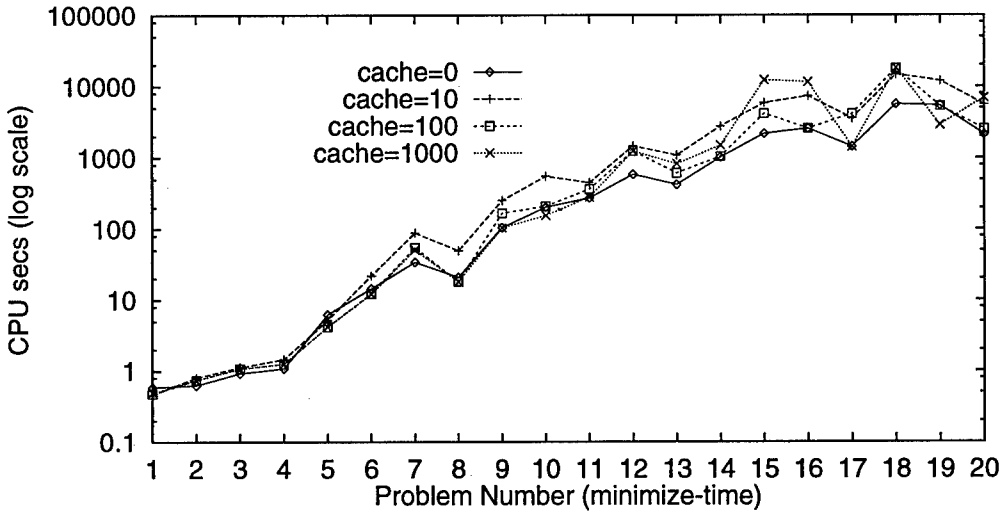

Figure 7.7: Overhead of caching for iterative strengthening.

Chapter 8

The ALPS Fast Scheduler and the Transportation Domain

This chapter describes the motivation, development, and performance of the ALPS "Fast Scheduler" for large-scale military transportation planning.[1]

Although the general goal of the ALPS project has been to develop general-purpose adaptive learning and planning techniques, the particular application we have addressed within the ARPA / Rome Laboratory Planning Initiative (ARPI) has been large-scale military transportation scheduling (see Section 1.1). When we first started developing our transportation domain theory, we ran into scaleup problems almost immediately: we could solve only about 100 cargos before the prototype ALPS Lisp Inference Engine exhausted all available memory, and it took up to an hour to get an answer for the larger problems. At first, we assumed that these problems were simply due to an inefficient first attempt at the domain theory, and that a combination of optimization and porting to the more efficient DALI inference engine would solve the problems. Doing those things did in fact give us one order of magnitude scaleup, but we were still stuck at between 500–1000 cargos. After trying several different approaches to improve the efficiency of the domain theory, we were able to solve certain problems with up to 2500 cargos; however, the performance was still unacceptably slow (on the order of several hours for 2500 cargos).

After further analysis, we concluded that one reason for the scaleup problem is that by treating scheduling as a logical domain, many useless intermediate results are maintained on the backtracking stack, even though we will never backtrack over them. For example, it is very expensive to recursively descend large lists because all sublists will be saved on the stack, even though we may know *a priori* that we will process the list only once. These results suggested that a strictly logical approach may never scale up fully in this class of scheduling problems because of the extreme memory requirements.

Faced with this conclusion, we looked for alternative approaches that still retained the spirit of the ALPS architecture. We took the basic domain theory developed for the Lisp Inference Engine and translated it directly into straight Lisp code. In most cases, this approach would not have been possible because by definition a logical domain theory does not contain any information on the order in which rules are executed (the execution order is controlled by the inference engine). However, in this case, we had already carefully ordered the rules while trying to optimize the theory, and we had constructed the theory in such a way that an answer could always be found with no backtracking over rules. Therefore, we could write a Lisp procedure that simply executed

[1] This chapter is adapted from [21].

each translated rule in the same order that the inference engine would have. By doing this, we could guarantee that our new Lisp program would get the same answer that the original domain theory would have produced, without the overhead inherent in the interpretation of logical rules. In particular, all recursive processing of lists could be done directly in Lisp and compiled into straight inline code; we could also use global hash tables to store much of the information more efficiently.

The net result is that we built a single domain-specific scheduler (referred to in this paper as the *"Fast Scheduler"*) that can solve TPFDD problems much more quickly than either the Lisp or DALI inference engines. We lose many of the adaptive properties of the core inference engines (caching, explanation-based learning, probabilistic theory revision, and distributed capabilities). We also lose the reusability of a core inference engine; to use a new domain theory, we would have to write a whole new program from scratch. However, we gain a dramatic increase in speed and scaleup: we have solved problems with 50 squadrons of aircraft and 10,000 cargos in about 3.5 minutes. This result more than justifies the loss of some speedup techniques that, while very effective in other domains, were not producing significant speedup in this domain anyway. See Section 11.2 for a further discussion of the relationship between the Fast Scheduler and the adaptive logical techniques developed during the ALPS project.

8.1 Evaluating the ALPS Fast Scheduler

To evaluate the ALPS Fast Scheduler and compare its performance against the other two inference engines, we needed a set of scalable transportation problems. The database files produced during military transportation scheduling exercises are called Time-Phased Force Deployment Data (TPFDD) files [50]. These files contain 86 fields for each cargo to be transported, providing information on such things as cargo size, type, origin, destination, intermediate points, preferred mode of transportation, and deadline restrictions for departure and arrival. A medium-sized TPFDD file may contain several thousand cargo records. Associated with TPFDD files are Geographic Location (GEOLOC) files describing the properties of all geographic locations mentioned in the TPFDD (location type, mapping from GEOLOC code to full name, and coordinates in latitude/longitude).

Because actual TPFDD problems are quite difficult to acquire due to their sensitive nature, we decided to write an automated random TPFDD generator (the TGEN module). TGEN can generate full TPFDD datafiles of arbitrary size and complexity. It can use both air and sea transport involving any location defined in a GEOLOC file. It is not restricted to a fixed set of pre-defined cargos or vehicles, but rather will generate appropriate cargos, airplanes, and ships on the fly. TGEN makes a fairly thorough attempt to ensure that the TPFDD it produces is both realistic and reasonable: it does appropriate clustering of ports of debarkation (PODs) and it verifies that each random cargo has at least one vehicle capable of transporting it from its origin to its destination within the allotted time. TGEN has turned out to be a very useful tool for producing scalable unclassified transportation scheduling problems.

To test the ALPS Fast Scheduler, we used TGEN to generate a scheduling problem with 50,000 cargos and 50 squadrons of aircraft and seacraft. Each squadron contains up to 32 identical vehicles selected from ten different vehicle types (five ship types and five aircraft types). The cargos and vehicles are based at random commercial US airports and seaports, and the destination is a cluster of airports and seaports around Puerto Rico. Delivery times are padded up to 30 days to simulate a one-month buildup of supplies at the destination. We used this base specification to construct a scaled set of increasingly difficult problems by selecting the first n cargos from the full specification (where n ranges from 10 to 50,000).

Figure 8.1 compares the performance of the three inference engines running on this set of

	elapsed runtime (hh:mm:ss)				memory used (MB)			
#Cargos	*Lisp*	*DALI*	*Fast*	*Fast(*)*	*Lisp*	*DALI*	*Fast*	*Fast(*)*
10	1:28	0:09	0:01	0:01	17	19	15	15
20	3:13	0:17	0:01	0:01	21	19	15	15
30	5:17	0:25	0:01	0:01	28	20	15	15
40	6:57	0:31	0:01	0:01	38	20	15	15
50	11:06	0:39	0:01	0:01	52	21	15	15
100		1:26	0:02	0:01		24	15	15
200		3:03	0:03	0:02		31	15	15
300		6:13	0:04	0:02		42	15	15
400		10:00	0:07	0:03		85	15	15
500			0:07	0:03			15	15
1000			0:17	0:08			15	15
2000			0:51	0:22			15	15
3000			1:34	0:41			15	15
4000			2:21	1:00			15	15
5000			3:14	1:20			15	15
10000			8:53	3:35			18	19
20000			27:09	11:03			25	25
30000			1:15:18	28:04			33	33
40000			2:28:17	54:18			39	40
50000			4:09:46	1:29:04			49	46

Figure 8.1: Comparison of three ALPS inference engines on transportation scheduling problems with 10 to 50,000 cargos. The "*Fast(*)*" test was run on a Sun Sparc5 with 64MB RAM and 190MB swap; all other tests were run on a Sun SparcIPC with 32MB RAM and 100MB swap. Tests were aborted if they exceeded the available memory.

problems. These results clearly indicate that the ALPS Fast Scheduler can successfully scale up to efficiently solve problems of the size found in real-life military transportation scenarios.

8.2 Graphical User Interface

ALPS has a graphical user interface (GUI) for use within the transportation domain.[2]

Figure 8.2 shows a screendump of the ALPS TPFDD Scheduler interface. Buttons across the top of the screen allow the user to invoke the TGEN problem generator, the ALPS Fast Scheduler, the TPFDD Simulator, the Plan Repair module, and a configuration window for adjusting display parameters. Under the control buttons are three text windows showing the original problem statement, the schedule created by the inference engine, and a trace of each simulation event (other textual information is available as well). The main window consists of timeline displays for the entire schedule. Time units are in days, subdivided into hours. The left portion of each timeline identifies the squadron and the particular trip (leg of the journey). A vertical red bar next to the

[2]The GUI was written using the Tcl/Tk software package. Tcl is an extensible general-purpose command language. Tk is a Tcl extension that provides an interface to the X Window System. Tcl and Tk are free, portable, and widely used.

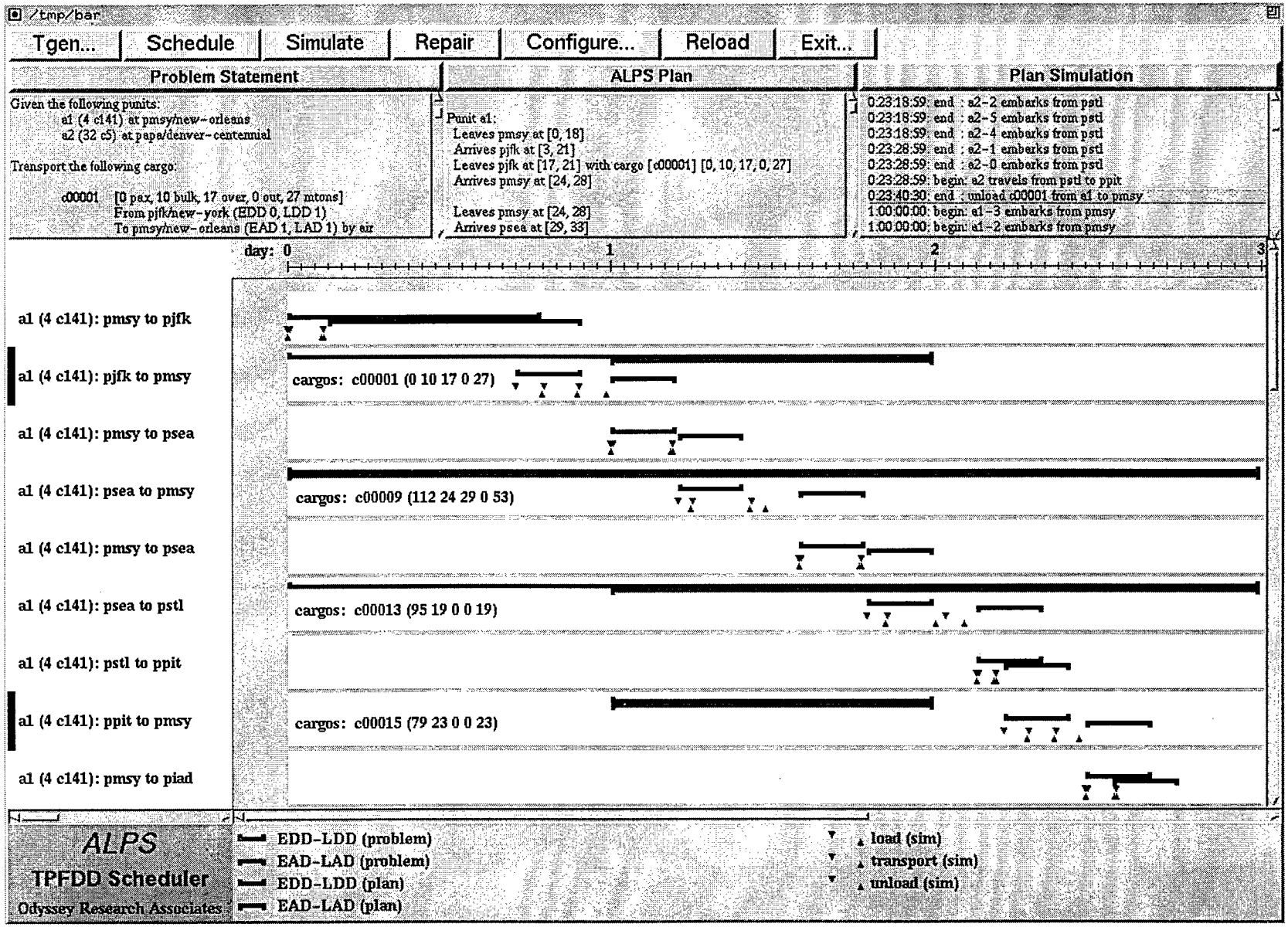

Figure 8.2: The ALPS TPFDD Simulator.

squadron is a warning flag that the cargos on this trip were not transported within the allotted time.

The right half of the timeline shows specific timing information for each trip. The top two bars (red and gray) illustrate the cargo departure/arrival time constraints from the original problem statement[3] (these bars are missing if there is no cargo for this trip). The middle two bars (blue and black) show the departure/arrival intervals proposed by the inference engine. The six triangles on the bottom show the actual times that the simulator started each process (black = begin/end loading, red = begin/end flying, blue = begin/end unloading). Clicking with the left mouse button anywhere in the white area will display the cargo manifest for this trip (cargo sizes are expressed as a 5-tuple of passengers, bulk tons, oversize tons, outsize tons, and measurement tons).

As an example, consider the second trip from Figure 8.2. This trip is identified as Squadron A1 flying from PJFK (New York) to PMSY (New Orleans) and is further identified as transporting Cargo C00001 (10 bulk tons and 17 oversize tons, for a total of 27 measurement tons). The original problem statement specified that the cargo must leave New York no earlier than Day 0 and no later than Day 1, and must arrive in New Orleans by Day 1. The schedule that ALPS generated specifies that Squadron A1 will leave between hours 17–21 and will arrive between hours 24–28. This schedule satisfies the original problem, but when the simulator attempts to execute this schedule, it discovers that loading and unloading the cargo takes less time than anticipated, and the cargo actually arrives about 20 minutes too early in New Orleans. Because this schedule does not satisfy the original problem constraints for Cargo C00001, this trip is flagged as a failure. When this schedule is sent to the plan repair module, the departure time will automatically be adjusted to prevent this failure.

[3]For trips involving multiple cargos, the departure time interval represents the intersection of the departure intervals for all cargos (similarly for arrivals).

Chapter 9

Iterative Plan Repair

This chapter surveys our work on iterative plan repair as part of the ALPS project. We propose a domain-independent plan repair algorithm. We view the plan repair problem as constructing a new plan based on updated information of action and state descriptions. Our approach addresses this problem by iteratively generating subproblems based on the failed plan, using the original planner to solve the subproblems, and fitting the new subplans back into the original plan. We have adapted this general plan repair algorithm to the transportation scheduling domain. Based on different completions of partially ordered subschedules, we have implemented two complementary repair strategies that can be combined very effectively.[1]

9.1 Introduction

Automated planning systems are becoming increasingly more powerful. Planning applications now include domains such as circuit design, transportation scheduling, and internet navigation. As planners are applied to these realistic domains, plan failures are inevitable. A plan can fail because of inadequate modeling of the domain (either deliberate to gain efficiency or accidental due to modeling errors). A plan can also fail because of changes in a dynamic environment. The task of plan repair is to adjust a faulty plan to eliminate failures.

There has been much active research in this area recently, and many techniques for plan repair have been proposed. Kambhampati [51] used validation structures to guide failure detection and repair. Hammond [45] used case-based reasoning techniques in his CHEF planner. Howe [49] presented several local repair techniques. There are also some domain dependent techniques in the literature; for example, Turner [110] suggested attaching "directives" to preconditions to handle plan failures.

In this paper, we use "plan repair" for what some other researchers referred to as "failure recovery" [49]. That is, we are not concerned with failure detection or failure analysis. We assume that when a failure occurs, a simulator or an execution monitor detects and reports the failure. The plan repair module takes failure information (and possibly an updated domain theory) as input, and it produces a new plan as output.

Once we separate failure detection and analysis from plan repair, the similarity between plan repair and plan generation becomes clear. Although a plan can fail for many reasons, repairing the plan involves the same inference process as generating the plan, and it should be possible to use the same planning system to perform both generation and repair. The main difference is that plan

[1]This chapter is adapted from [117].

repair can use the old plan as a starting point in the search. We use a basic heuristic assumption that most failures can be fixed with local modifications. If a failure involves global changes to the original plan, it is unlikely that any repair method will be better than simply replanning from scratch.

We propose an approach that exploits this locality of plan repair and maintains the completeness property by doing iterative replanning. We repair a plan by retracting actions that are "local" to the failure, formulating a new planning problem based on the goals of those retracted actions, and solving that problem to generate a replacement sequence of actions. We do the retraction and replacement of actions iteratively until the resulting plan is correct. In some sense, our approach is similar to Howe and Cohen's [49] planner; they iterate through different repair methods while our method iterates through different subplans. Our repair strategy is complete, domain independent, and suitable for a variety of plan representations. It has been incorporated into the ALPS transportation scheduling system, both as a general-purpose technique and as a domain-specific refinement for the transportation domain [20].

The rest of this report is organized as following. In Section 9.2, we survey some plan repair techniques in the literature. In Section 9.3, we present the algorithms for our plan repair technique and explain the details of various data structures. In Section 9.4, we give an example to demonstrate our algorithms. In Section 9.5, we show how our general repair techniques are adapted to a transportation scheduling domain and give some test results. Section 9.6 is a discussion section and conclusion.

9.2 Related Work

The simplest way of doing plan repair is to replan from scratch with updated information (new initial state, new action description, etc.). The obvious disadvantage of this method is inefficiency, but an equally important disadvantage in some domains is loss of continuity: replanning from scratch might make arbitrary changes to portions of the plan that could have been salvaged.

Local repair strategies attempt to circumvent these two problems by making isolated modifications, ranging from reinstantiating a single variable to replacing a faulty action [99, 116]. These methods can be highly efficient, but they can typically handle only a limited number of domain-specific failure types.

A more general approach to plan repair is to use ideas from plan modification and plan reuse. Kambhampati [51, 52, 53] provides a systematic way of performing plan modifications based on *validation structures*. A validation structure represents the internal dependency of a plan. This structure is used to identify the subplan to be modified, suggest modifications, select and control the refitting, and assist in plan mapping and retrieval. The method is both complete and consistent. Our plan repair problem can be viewed as a special case of this plan modification problem.

From the complexity point of view, Nebel [76] has shown that conservative plan modification is at least as hard as plan generation, and in some cases can be even harder. In this context, "conservative" means that plan modification causes minimal change in the old plan. However, this result may not be relevant in practice as long as we use conservatism as a desired heuristic rather than as a hard requirement.

Hammond [45] uses a case-based repair strategy in CHEF, a planner in the domain of Chinese cooking. Similar to [99], CHEF uses deep causal reasoning to explain the failure. It then uses this explanation to index plan repair strategies. Based on the reason for a failure, repair strategies are classified into several categories, each of which suggests a way of fixing the faulty plan. CHEF's repair strategy can handle many types of failure with carefully crafted solutions, but its main

drawback is that it requires highly knowledge-intensive effort for each new domain.

Turner [110] suggests a plan repair technique of attaching *directives* to preconditions. When a precondition is violated, either a "strategic" or "tactical" plan modification will be carried out to avoid the violation. Turner also makes a distinction between absolute and flexible preconditions. As with CHEF, designing the directives involves a large amount of domain-specific information. Also, directives do not address failures in which no preconditions are violated (for example, a dynamic simulator may simply report back a step failure and a new state without giving specific precondition violations that could be used to retrieve a directive).

It is often useful to make a distinction between failure *detection* and failure *recovery*. Our iterative replanning algorithm handles the latter part by reusing the same inference engine for plan generation. In this case, explaining a failure essentially means generating new action (state) descriptions based on the observed failure. Failure detection techniques from [11, 84] and causal reasoning techniques from [45, 99] might be useful to combine with our inference technique to produce an effective plan repair module.

9.3 General Plan Repair

In this section, we describe an algorithm for doing *general plan repair (GPR)*. We will provide pseudo-code for the algorithm, and show how the data structures used in the algorithm can be constructed efficiently.

The following notation is used:

- A: individual action.
- AS : an action sequence or a plan. We also use $\{A_i, \ldots, A_j\}$ to represent the action sequence from A_i to A_j.
- S: state. S_0 represents the initial state. A state is a set of properties and their truth values at a particular point in time. An action can be viewed as a transition between states $(A_i : S_{i-1} \rightarrow S_i)$.
- G: set of one or more goals or subgoals, which are properties that must be true in the final state.
- P: planning problem, which is a pair of starting state and final goals. $P = (S,G)$.

A complete plan can be viewed as a linear sequence of actions:

$$S_0 \stackrel{A_1}{\rightarrow} S_1 \stackrel{A_2}{\rightarrow} \ldots \stackrel{A_i}{\rightarrow} S_i \stackrel{A_{i+1}}{\rightarrow} \ldots \stackrel{A_n}{\rightarrow} S_n$$

It also corresponds to a sequence of state transitions. A failure is signaled when there is a discrepancy between a planned action/state and a current action/state. The current action/state can result from either simulation or execution.

In this framework, there are two reasons that a plan can fail:

1. Action A_i does not establish the expected transition between S_{i-1} and S_i.
2. State S_{i-1} (or S_i) may be changed to some other state S_σ by an external event.

```
state-based-plan-repair(A_i, AS)        [A_i = bad action, AS = plan]
{
    AS_before, AS_after ← partition-plan(AS, A_i)
    loop
    {
        S ← construct-state(S_0, AS_before)
        G ← construct-state(S_0, {AS_before, AS_after})
        AS_i' ← call-planner(S, G)
        if successful
            then return AS_f = {AS_before, AS_i', AS_after}
            else AS_before, AS_after ← retract-actions(AS_before, AS_after)
    }
}
```

Figure 9.1: The state-based plan repair algorithm.

For the purpose of plan repair, a type-2 failure can be reduced to type-1 by adding a dummy action A_d between S_i and S_σ, and reporting a type-1 failure on A_d.[2] In the following discussion, we assume that if an action A_i fails, a new action $A_{i'}$ is generated by failure analysis. The action description for $A_{i'}$ is then asserted to the domain theory and the description for A_i is retracted.

9.3.1 Algorithms for General Plan Repair

9.3.1.1 Overview

Our approach starts by trying to find an action sequence $AS_{i'}$ to replace a faulty action A_i; if that does not work, we then try to replace the action sequence from A_{i-a} to A_{i+b}, iteratively increasing the numbers a and b until a satisfactory replacement sequence $AS_{i'}$ is found. The final plan AS_f is

$$\{A_1, \ldots, A_{i-a-1}, \{AS_{i'}\}, A_{i+b+1}, \ldots, A_n\}$$

We generate the replacement sequence $AS_{i'}$ by analyzing the effects of the retracted sequence of actions $\{A_{i-a}, \ldots, A_{i+b}\}$, formulating a new subproblem P_i, and submitting it to the original planner for plan generation.

Notice that the actions we retract are always contiguous. This requirement greatly simplifies the generation of the new subproblem (see Section 9.6). Also notice that if the failure is reported by an execution monitor rather than a simulator, we must require that $a = 0$ because actions committed cannot be retracted.

The rest of our algorithm deals with generating the subproblem P_i and verifying the solution AS_f. Based on two different ways of generating the subproblem P_i, there are two approaches for plan repair: the *state-based* approach and the *goal-based* approach.

9.3.1.2 The State-Based Approach

The state-based approach to general plan repair is shown in Figure 9.1. Initially we use state S_i

[2] We report a type-2 failure only if the state change will affect future plan execution or the final goals and we need to capture the causality of an external event.

as the initial state, and we use the conjunction of all properties of state S_{i+1} as the goal of the subproblem. If the planner fails to solve the subproblem, we use state S_{i-a} as the initial state and use the conjunction of all properties of state S_{i+b} as the goal. The numbers a and b may be increased in any monotonic fashion after every iteration. The generation of states and their properties is discussed in Section 9.3.2.

The verification of the final plan AS_f is trivial. The replacement sequence $AS_{i'}$ starts from state S_{i-a} and reproduces state S_{i+b}. To the rest of the plan, $AS_{i'}$ is indistinguishable from the original sequence $\{A_{i-a}, \ldots, A_{i+b}\}$. So if the original plan is valid without failure A_i (i.e., the planner is sound), plan AS_f is valid with failure A_i corrected.

However, using the conjunction of all properties of state S_{i+b} as the goal for problem P_i is too restrictive. Very often it is impossible to reproduce exactly the same state with a different set of actions. Also, this approach greatly overconstrains the problem: state S_{i+b} can easily have hundreds of properties, many of which are irrelevant to the rest of the plan. We address these concerns with the goal-based approach below.

9.3.1.3 The Goal-Based Approach

This approach differs from the state-based approach in that the goal of P_i is formulated based on the relevant goals of action sequence $AS_i = \{A_{i-a}, \ldots, A_{i+b}\}$. Numbers a and b are also increased based on a goal structure through iterations. Using the domain theory and a complete plan, we can construct a list of state properties and a goal hierarchy. We say an action A "supports" a subgoal G if one of the effects of A achieves G. In the goal hierarchy, either A is listed under G or A has a pointer to G (see Section 9.3.2). The goal-based approach is shown in Figure 9.2.

1. Initially, when an action failure A_i is reported, we retract that action A_i and formulate a new subproblem based on the state S_i and the goals that A_i supports.

   An action is listed in the goal structure under exactly one subgoal; it has pointers to other subgoals it supports. The details of constructing the goal structure is explained in Section 9.3.2.

2. We send the subproblem to the planner. If the planner fails to produce a subplan, we backtrack to the parent G_p of action A_i in the goal hierarchy. The sequence of actions $\{A_{i-a}, \ldots, A_{i+b}\}$ under the goal G_p are retracted.

   We require that the sequence of actions are temporally continuous in the complete plan. Actions under one subgoal do not necessary satisfy this requirement, although our construction of the goal structure tries to achieve this continuity. In any case, we set A_{i-a} to be the first action under G_p, and A_{i+b} to be the last action under G_p.

3. The new subproblem P_p is formulated based on the state S_{i-a}, the goal G_p, and all other goals G_o that are supported by $AS_i = \{A_{i-a}, \ldots, A_{i+b}\}$.

   G_o should include only those subgoals that are not listed under G_p; any subgoals listed under G_p no longer need to be supported once G_p is retracted.

4. If the planner finds a new sequence of actions $AS_{i'}$ to solve the problem, we fit $AS_{i'}$ in the old plan and verify the preconditions of all actions following $AS_{i'}$.

   In the formulation of the new subproblem P_p, we account for the desired effects of the retracted actions by adding goals G_o. But we still have to verify the preconditions of later steps to ensure that the new action sequence $AS_{i'}$ does not introduce damaging side effects.

```
goal-based-plan-repair(A_i, AS)        [A_i = bad action, AS = plan]
{
    AS_before, AS_after ← partition-plan(AS, A_i)
    loop
    {
        S_before ← construct-state(S_0, AS_before)
        S_after ← construct-state(S_0, {AS_before, AS_after})
        AS_i' ← gen-plan-repair(S_before, subgoal ∪ side-effects, AS_after)
        if successful
            then return AS_f = {AS_before, AS_i', AS_after}
            else AS_before, AS_after, subgoal, side-effects ← retract-actions(AS_before, AS_after)
    }
}

gen-plan-repair(S_before, subgoals, AS_after)
{
    new-subgoals ← ∅
    loop
    {
        subplan ← call-planner(S_before, subgoals ∪ new-subgoals)
        new-subgoals ← verify-plan(subplan, AS_after)
    }
    until new-subgoals = ∅
    return subplan
}
```

Figure 9.2: The goal-based plan repair algorithm.

Given that state S_i has properties $\Pi_i = \{p_1, p_2, \ldots, p_n\}$,
Calculate Π_{i+1} (properties of S_{i+1} after applying action A_i in S_i) as follows:

STRIPS:
If action A_i has addlist α_i and delete list δ_i,
then $\Pi_{i+1} = \{\Pi_i - \delta_i\} \cup \alpha_i$.

Situation Calculus:
If action A_i is represented by the clauses
holds(x_1, do(A_i, S)) $\leftarrow$ holds(p_{1_1},S), holds(p_{1_2},S), ..., holds($p_{1_{n1}}$,S)
...
holds(x_m, do(A_i, S)) $\leftarrow$ holds(p_{m_1},S), holds(p_{m_2},S), ..., holds($p_{m_{nm}}$,S)
then $\Pi_{i+1} = \{x_j \bullet \theta \mid \forall k : p_{j_k} \bullet \theta$ unifies with some $p \in \Pi_i\}$
where θ is the most general unifier.

Figure 9.3: The algorithm for generating state properties.

5. If a precondition G_j of action A_j $(j > i + b)$ is not satisfied, we backtrack, add G_j to the goals of P_p, and continue until all preconditions are satisfied. If the repair at G_p level fails, we backtrack to a higher level subgoal in the goal hierarchy.

The iterative repair strategy is complete; in the worst case, it will retract all actions and replan the original problem from scratch.

9.3.2 Data Structures

There are two major data structures used by the plan repair algorithm. One is the list of states and their properties, and the other is the goal hierarchy. The state-based repair uses only the former, and the goal-based repair uses both.

9.3.2.1 List of State Properties

For both STRIPS [40] and situation calculus [68] notation, we can use forward chaining (progression) to generate the list of state properties, as shown in Figure 9.3.

9.3.2.2 Goal Structure

In a goal structure, there are two types of nodes: *action* nodes and *goal* nodes. An action node is connected to its parent node through a *subgoal* link and is connected to its children through *precond* links. We call the corresponding reversed links *action* and *precond_of*, respectively. We regard the goal hierarchy as a tree structure based on the subgoal/action and precond/precond_of links. We handle side effects by attaching *effect* and *supports* pointers to the tree structure. Figure 9.4 illustrates an example goal structure.

Goal structures can be generated based on the final goals and a complete plan using the algorithm shown in Figure 9.5. This algorithm works for both STRIPS and situation calculus notation, although the preconditions are represented differently for each.

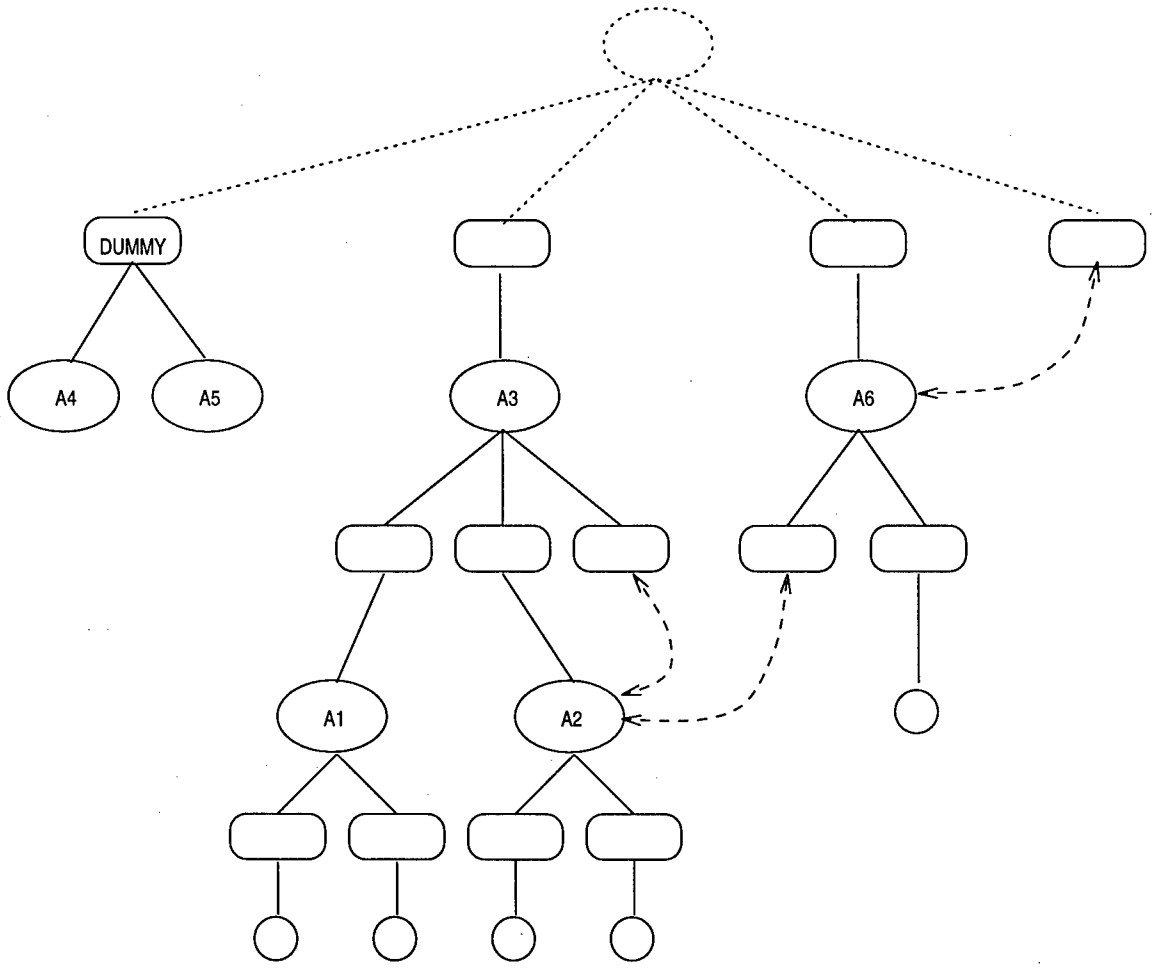

Figure 9.4: An example goal structure. Rectangles denote goal nodes; ovals denote action nodes; circles denote the initial state. Dashed arrows lines are effect/supports links. The dotted oval at the top is a dummy action node to guarantee a tree structure.

```
construct-goal-structure(G, AS)
        [G = {p_1, ..., p_m} = top-level goals, AS = {a_1, ..., a_n} = plan]
{
    for i from n downto 1 do
    {
        g ← the first goal in G that A_i supports
        if (g = ∅)
            then A_i.subgoal ← dummy
            else
            {
                g.action ← A_i
                A_i.subgoal ← g
                G ← G − g
                ∀p ∈ A_i.preconditions: G = p + G
            }
        ∀ other g ∈ G
        {
            if A_i supports g
                then
                {
                    A_i.effects ← g + A_i.effects
                    g.supported ← A_i
                }
            G ← G − g
        }
    }
}
```

Figure 9.5: The algorithm for constructing the goal structure.

Given a sound plan, the goal hierarchy constructed by the algorithm in Figure 9.5 has the following properties.

1. Each action is listed under exactly one subgoal through its subgoal link.
2. Each subgoal is listed under exactly one action through its precond link.
3. For each subgoal, one of the following is true:
   (a) the subgoal is supported directly by an action link;
   (b) the subgoal is supported indirectly by one or more effect/supports links;
   (c) the subgoal is supported indirectly as a property of the initial state.

   In other words, every subgoal in the goal structure is supported.

The first two properties can be proven trivially based on our algorithm. The final property follows from Chapman's truth criterion for complete plans [24].

Notice that we assign subgoal and effect links in a greedy fashion; that is, a subgoal is always supported by the last action that asserts it. In this way, the construction of the goal structure is greatly simplified because we do not have to worry about causal link violations. Our goal structure does not necessarily reproduce the same proof structure or causal structure in the plan generation. The correctness of the goal structure (property 3) is guaranteed by the truth criterion for complete plans.

An action A_d with a Dummy subgoal is a redundant action because A_d does not support any subgoals. We can safely remove actions listed under a dummy subgoal and reorder the plan. In the rest of this discussion, we assume that such actions do not exist.

9.4 An Example

In this section, we will walk through a simple example to demonstrate our plan repair algorithm. We use an extended version of the monkey and bananas problem. Since the complete domain theory is quite long and the preconditions and postconditions of each actions are intuitive, we omit the detailed listing.

The initial state is shown in Figure 9.6. The world includes six numbered rooms (`L0` through `L5`) and each room contains one or more objects. The goal is to achieve `destroyed(Box)`. The monkey (`M`) starts in room `L1`, and there is dynamite (`D`) in rooms `L4` and `L5`.

The original plan is shown in Figure 9.7. Now let us suppose that our description of the `GRAB` operator was incorrect: this operator actually has an additional precondition that dynamite must be unlocked in order to be grabbed. If the dynamite in room `L4` is locked, then the action `GRAB(M,D,L4,Floor)` will fail. We create a new operator `GRAB_N` that has an additional precondition requiring that the grabbed object is not locked, and we assert this new operator into the domain theory, retracting `GRAB`.

Based on the original plan and goal, we can construct a goal structure using the algorithm from Figure 9.5. The goal structure for this problem is shown in Figure 9.8.

We retract the failed action `GRAB(M,D,L4,Floor)` and formulate a new problem P_i. The starting state of P_i is S_1. The goal of P_i is `has(M,D)`, since that is the property supported by the retracted action. A solution for subproblem P_i is shown in Figure 9.9; basically, the monkey finds an unlocked dynamite in room `L5`. At the verification stage, we detect that the precondition `at(M,L4)` of a later action is violated. We add this precondition as an additional goal of P_i. A new solution of the

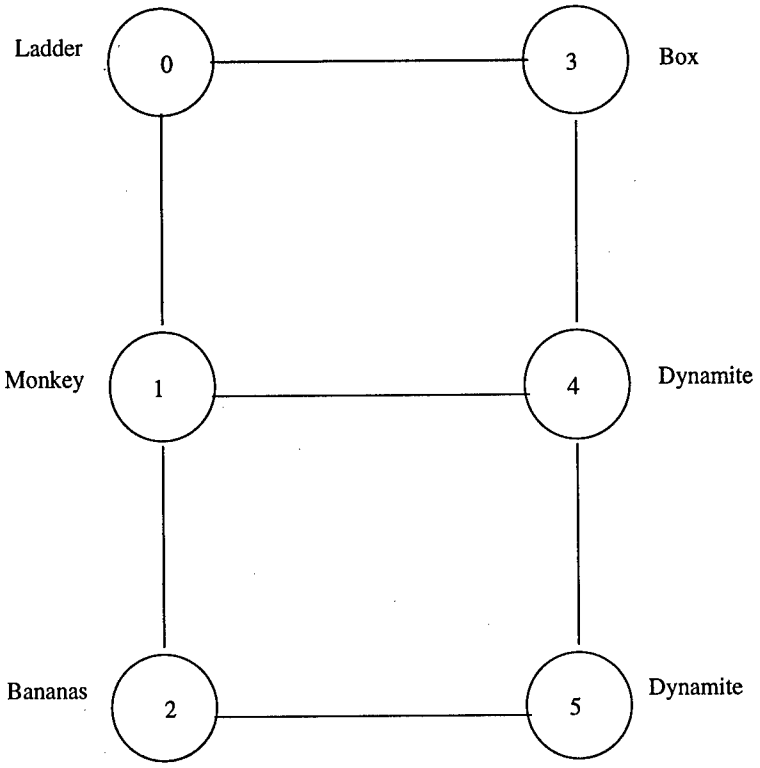

Figure 9.6: The initial state.

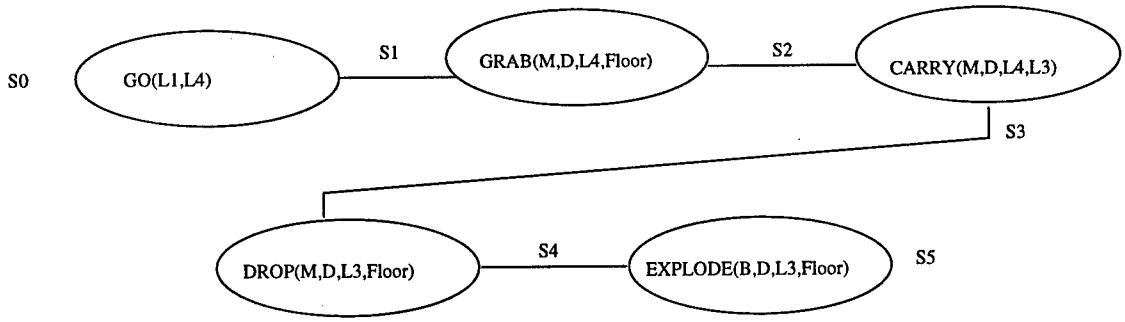

Figure 9.7: The original plan.

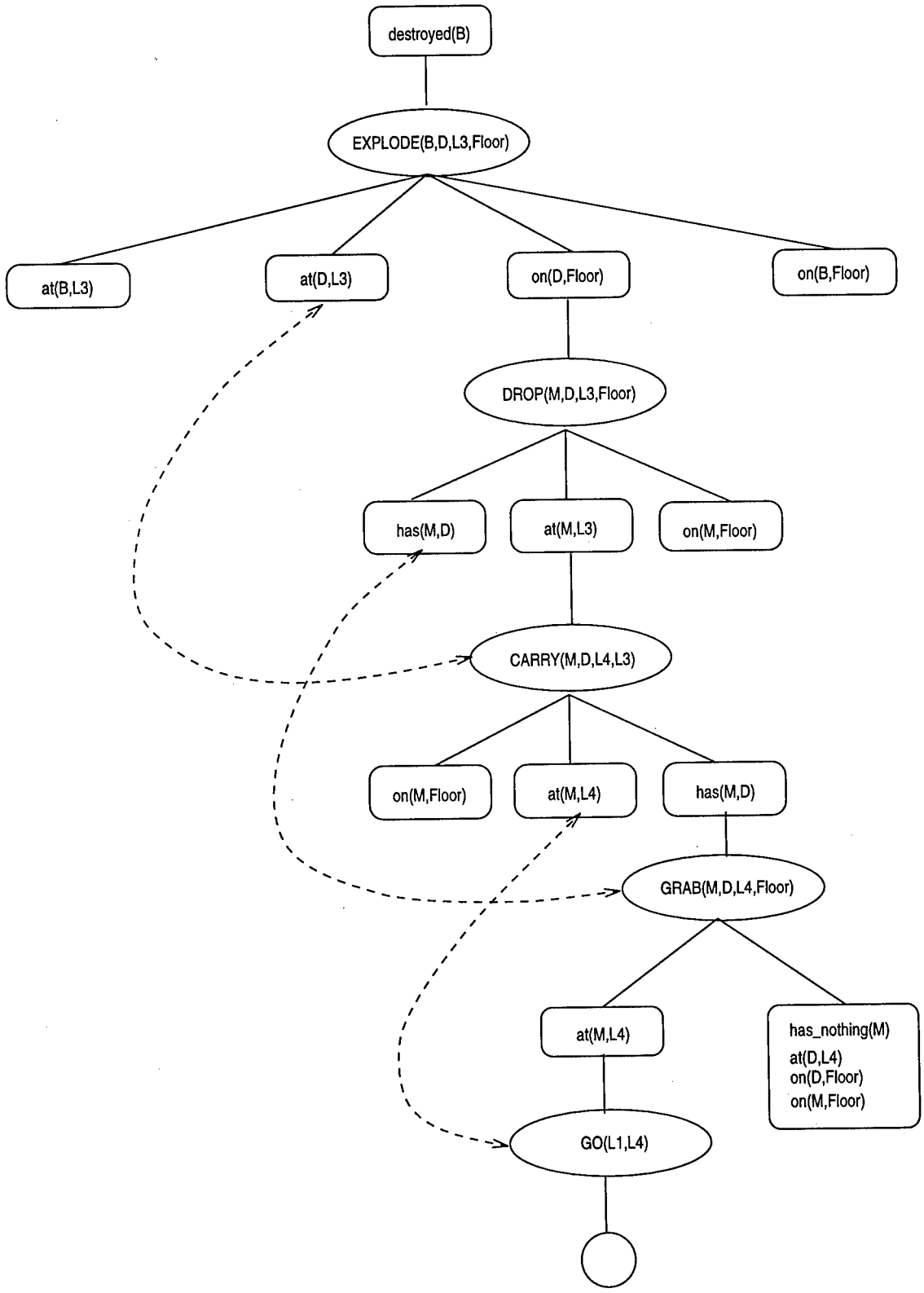

Figure 9.8: The goal structure.

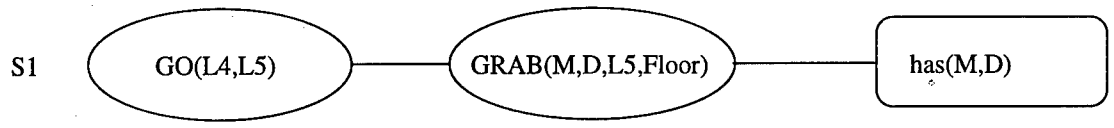

Figure 9.9: The first solution of subproblem P_i.

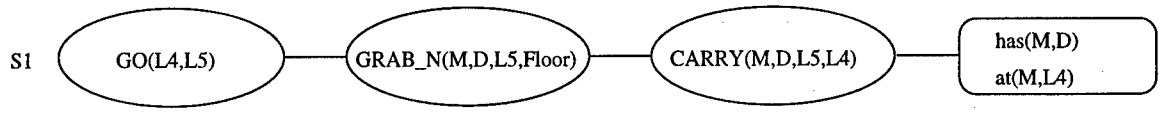

Figure 9.10: The final solution of subproblem P_i.

subproblem is show in Figure 9.10; this solution satisfies all preconditions of the later actions. The final plan is shown in Figure 9.11. If any future failure occurs, this final plan should be used to construct a new goal structure.

9.5 Plan Repair in the Transportation Domain

In this section, we show how the general plan repair strategy introduced in Section 9.3 can be adapted to the specific domain of transportation scheduling. We can use specialized domain information to optimize some of the procedures in the general plan repair approach. We begin by briefly explaining the transportation scheduling problem.[3]

A problem statement consists of a set of airports, a set of airplanes, and a set of cargos to deliver. Each airplane is based at some airport and has various constraints such as maximum speed, minimum runway length, and maximum cargo capacity. Each cargo has an origin at one airport, a destination at another airport, a size/weight description, and delivery constraints defined by earliest/latest departure time and earliest/latest arrival time. The goal is to deliver all cargos to their destinations without violating any constraints.

We call a single flight from one airport to another airport a *trip*. We assume that cargos are delivered using non-stop trips only and that multiple cargos can be transported during a single trip. A plan is represented as a set of trip schedules, one for each airplane. As discussed in Section 9.3, a plan can fail either because an action did not perform as expected (for example, a trip took too long) or because the world changed unexpectedly (for example, a cargo's weight changed). We continue to treat these two failure types uniformly from the perspective of repair.

Table 9.1 presents a mapping from the terminology of a general planning domain to the specialized terminology of transportation scheduling. This correspondence suggests how we can adapt general plan repair to the transportation domain. The transportation domain is an example where state-based plan repair is inadequate; since a state is represented by the locations of cargos and airplanes at a certain time, if a trip fails it is unlikely that an alternative schedule can reproduce the same state.

The method we use for plan repair in the transportation domain is very similar to the general plan repair algorithm, but we have taken advantage of the simple goal structure in this domain.

[3]For presentation purposes, we have simplified some irrelevant details in this description; see [15, 16, 17, 20] for a more complete description.

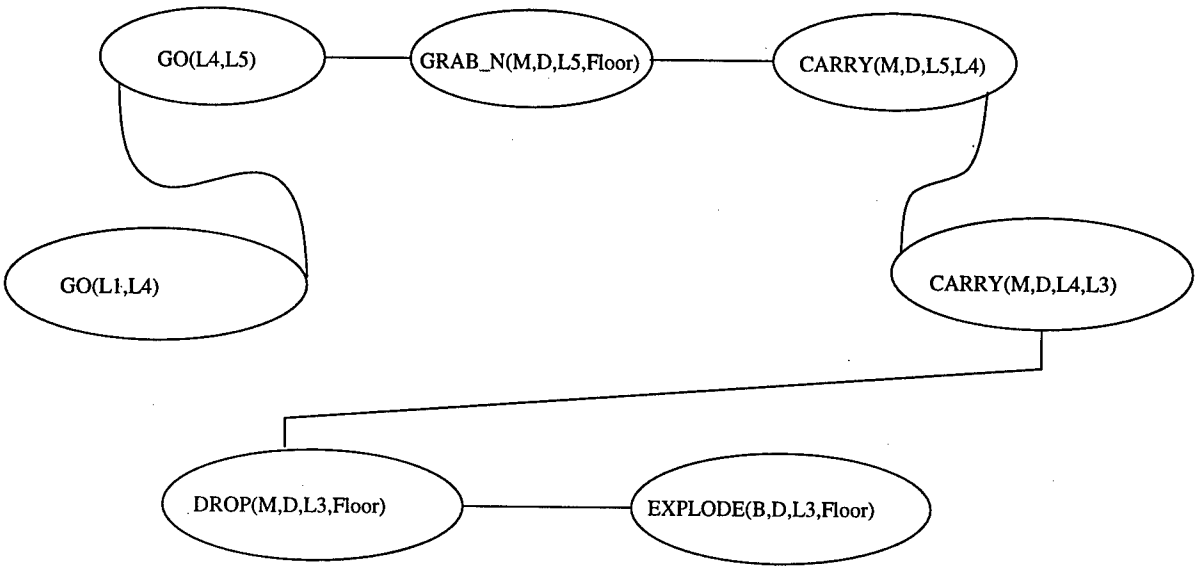

Figure 9.11: The final plan.

General Domain	*Transportation Domain*
plan	airplane schedule
state	airplane & cargo locations
action	trip
action sequence	sequence of trips
action failure	trip failure
world changes	new cargo assignment
new action	new scheduling algorithm
subgoal	individual cargo delivery
# actions for one subgoal	1
goal hierarchy	linear (degenerate) tree
depth of the goal structure	number of trips
plan verification	temporal/physical constraint propagation

Table 9.1: Terminology for generic and transportation planning.

Each action is a single airplane trip, and the only goal of that action is to deliver the cargos on that trip within some time constraints. Although an action does not have any other goals, it may still have undesirable side effects: if one trip of an airplane is late, all consequent ones may be affected. These side effects are handled by a temporal propagation procedure.

A plan is a list of schedules for a set of airplanes. There are no causal or temporal relations among these schedules, so the trip lists for two airplanes can be considered independently.[4] Viewing the plan as partially ordered based on delivery time intervals, we can change the behavior of the plan repair module by arrange trips in different ways. We have found two ways of viewing the plan that are particularly useful. One is to order the trips based on airplanes, and the other is to order the trips based on their departure times. We call the first one *single plane repair (SPR)* and the second one *multiple plane repair (MPR)*.

For SPR, we restrict the set of retracted trips to be within one single airplane schedule. Essentially, SPR repairs a flaw by rearranging the cargos and trips within a single airplane schedule. As in the general plan repair algorithm, we update the domain theory before performing any repair; for example, we might need to set the new traveling time of a trip based on simulator feedback. Initially, the repair module retracts the single identified failed trip, updates its temporal intervals, and tries to fit this updated trip back in the original schedule for this airplane. If the updated trip does not fit, the module will iteratively retract trips before and/or after the initial faulty trip, reschedule all cargos on these trips in isolation (using only this airplane), and try to fit the new subschedule back into the original schedule. The iteration stops when the rescheduled trip sequence fits in the airplane schedule. Note that during this process some cargos may have been "bumped off" because they are no longer possible to schedule.

MPR has the added ability of rearranging cargos among multiple airplanes. Initially, the repair module tries to insert an undelivered cargo directly into the existing global schedule. If this insertion is not successful, MPR will iteratively retract cargo trips within a certain time interval to create a "window" across all airplane schedules and will try to fit all cargos back into the global schedule (not necessarily on their original airplanes). The iteration succeeds if the undelivered cargo *and all retracted cargos* fit in the schedule; otherwise the undelivered cargo is marked as "too hard".[5]

Interestingly, SPR and MPR can be combined to handle different types of failures very efficiently. By ordering trip schedules differently, SPR and MPR can exploit two different views of locality to perform different types of "local repairs". SPR is more appropriate for handling delayed trips and deadline violations because these failures can most often be avoided by adjusting trips locally within the same airplanes. On the other hand, undelivered cargos or airplane failures often require the collaboration of multiple airplanes in MPR, and the most relevant trips are the ones clustered locally around similar departure times.

This synergy demonstrates the advantage of using specific domain heuristics to adapt a general plan repair strategy. In the transportation domain, the goal hierarchy is very simple: since each airplane schedule is independent and since each trip within a subschedule depends temporally on all trips before it, the hierarchy degenerates into a set of unconnected right-branching trees. So we can customize the method of iteratively retracting actions to take advantage of this structure: the partition step will treat each separate airplane subschedule as a linear sequence (split at the timepoint of the bad cargo's departure), and during each iteration we will retract one trip from each side of each subschedule.

We now provide a simple example from the domain of transportation scheduling. There are

[4] In this paper, we make the simplifying assumption that there is no resource contention among airplanes. But note that the actual ALPS system does address resource contention issues.

[5] Notice that MPR does not allow other cargos to be bumped off as SPR did; this is done primarily to avoid infinite loops (for example, cargo A bumps off cargo B, which then bumps off cargo A).

cargo	origin	destination	depart	arrive
C1	Miami	Los Angeles	48–143	80–100
C2	Washington DC	Los Angeles	72–143	72–143
C3	Honolulu	Los Angeles	0–171	48–171

airplane	home port
A1	Phoenix
A2	Chicago

Table 9.2: A sample transportation problem.

three cargos to be delivered on two available airplanes, as shown in Table 9.2. The departure (arrival) intervals represent the hours during which the cargo must depart (arrive).

The scheduler came up with the following plan.

```
Airplane A1:
  Leaves Phoenix at time 0--63
  Arrives Honolulu at time 6--69
  Leaves Honolulu at time 35--69 with cargo C3
  Arrives Los Angeles at time 48--82

  Leaves Los Angeles at time 48--82
  Arrives Miami at time 53--87
  Leaves Miami at time 67--87 with cargo C1
  Arrives Los Angeles at time 80--100

  Leaves Los Angeles at time 80--125
  Arrives Washington DC at time 85--130
  Leaves Washington DC at time 85--130 with cargo C2
  Arrives Los Angeles at time 98--143
```

During simulation of this schedule, we discover that

- the trip delivering cargo C3 is running late and actual flight time will be 98 hours.[6]
- Similarly, delivering cargo C2 will take 25 hours.
- A new cargo C4 must now be delivered from Pittsburgh to Los Angeles, departing and arriving during hours 0–191.

We first use SPR to adjust the trips within airplane A1 to fix the first two problems, with the following result:

```
Airplane A1:
  Leaves Phoenix at time 0--9
  Arrives Honolulu at time 6--15
  Leaves Honolulu at time 6--15 with cargo C3
  Arrives Los Angeles at time 104--113
```

[6]This example is deliberately extreme to illustrate the plan repair algorithm. A schedule with only three cargos does not usually need sophisticated repair.

```
  Leaves Los Angeles at time 104--113
  Arrives Washington DC at time 109--118
  Leaves Washington DC at time 109--118 with cargo C2
  Arrives Los Angeles at time 134--143

Cargo C1 could not be delivered.
```

Cargos C2 and C3 are rescheduled successfully, but cargo C1 is bumped off. MPR is called next to reschedule C1 as well as C4. The final schedule is

```
Airplane A1:
  Leaves Phoenix at time 0--9
  Arrives Honolulu at time 6--15
  Leaves Honolulu at time 6--15 with cargo C3
  Arrives Los Angeles at time 104--113

  Leaves Los Angeles at time 104--113
  Arrives Washington DC at time 109--118
  Leaves Washington DC at time 109--118 with cargo C2
  Arrives Los Angeles at time 134--143

  Leaves Los Angeles at time 134--173
  Arrives Pittsburgh at time 139--178
  Leaves Pittsburgh at time 168--178 with cargo C4
  Arrives Los Angeles at time 181--191

Airplane A2:
  Leaves Chicago at time 0--84
  Arrives Miami at time 3--87
  Leaves Miami at time 67--87 with cargo C1
  Arrives Los Angeles at time 80--100
```

9.6 Discussion and Conclusions

There are two basic ideas underlying our plan repair algorithm. One is that when an action fails, local actions are likely to be responsible for the failure (and thus the recovery). But what does "local" mean? In the transportation domain from Section 9.5, we used temporal locality based on the (temporal) execution order of actions in a plan. A more natural choice might seem to be "causal" locality; however, with most non-linear plans, temporal locality can be made to reflect causal locality when actions are ordered based on some particular traversal of a causal structure.

The other idea is that we can use the inference power of a generative planner to do plan repair. Once the failure is identified, we need to generate a plan based on a new set of requirements (in most cases similar to the old set). By reducing the plan repair problem to the formulation of a new planning problem (which is usually smaller and easier than the original planning problem), we can use the original planning system to perform repair as well.

Using temporal locality can also help to minimize the changes to the overall plan structure. Although optimally conservative modification is not computationally feasible [76], we do not necessarily want to truly minimize the number of changed actions. In the transportation domain, for

example, it may be less intrusive to reorder 50 trips within one single airplane than to replace 20 trips spread across many airplanes. Using a combination of SPR and MPR in this domain clusters the changes naturally, producing good locality of modification without requiring optimal conservation.

To demonstrate the effectiveness of this approach, we tested our algorithm by adding 100 random new cargos to an existing schedule already containing 100 cargos (we can treat this as one type of plan repair). Of these 100 new cargos, 92 of them could be added using SPR without having to modify the rest of the schedule at all. Seven of the remaining eight could be added after shuffling some trips with MPR. The final cargo could not be delivered at all. This example exhibits the locality nature of plan failure and recovery. Had we replanned from scratch for all these 100 cargos, the new schedule would have been significantly different from the original plan. In many domains (including transportation scheduling), such changes are very costly.

9.6.1 Comparison to Other Methods

Our plan repair methodology appears to fit within Howe and Cohen's [49] classification, which listed six different plan repair strategies. One of their strategies is to replan at the parent level. Our goal-based repair algorithm seems to be a domain-independent extension of that strategy.

The goal structure we use is similar to other causal link structures (e.g., [114]). The main difference is that our goal structure is treated as a tree, while other causal link structures are viewed as networks. In our goal structure, an action is listed under exactly one subgoal based on its temporal order in the complete plan. Links to other subgoals supported by an action are indicated by *effect/supports* pointers superimposed on top of the tree.

9.6.2 Limitations and Future Work

State-based repair is applicable to most planning representations; the only requirement is the ability to derive all properties of a state. For goal-based repair, we use the goal structure. The algorithm we provide works for simple STRIPS and situation calculus notations. It will be interesting to see how the algorithm can be extended to handle more powerful representations such as disjunctive goals and conditional actions. The plan and the domain theory alone may not be enough to construct the goal structure anymore, and we might have to rely on additional information from the planner such as a proof tree. So far we have deliberately avoided using any information from a proof tree because we are attempting to keep the method completely independent of any particular planner.

9.6.3 Conclusion

This report summarizes our investigation in plan repair techniques. We have designed a general plan repair algorithm that is independent of any particular planner or planning domain. Our plan repair strategies can be either state-based or goal-based. Both of these strategies handle STRIPS and situation calculus notations. We have adapted the general algorithm to the transportation domain. The domain-specific method takes advantage of the special goal structure in transportation schedules. We use a combination of two separate modules (single plane repair and multiple plane repair) to form a plan repair unit that is complete and correct relative to the underlying planner. This plan repair unit adds to the ALPS fast scheduler the capability of locally adjusting individual trips and adding new cargos while maintaining continuity in the overall plan structure.

Chapter 10

The ALPS TPFDD Simulator

Within the transportation domain, ALPS uses a domain-specific simulator to identify potential flaws in transportation schedules. The ALPS TPFDD Simulator is designed to augment particular capabilities and deficiencies of the ALPS transportation domain theory, and is also able to test schedules for robustness in the presence of unanticipated external events.[1]

Our original design for the ALPS architecture had provisions for a plan critic, whose purpose was to project a proposed plan or action into the future, identify potential flaws, and assist a plan repair module in correcting those flaws. In the process of refocusing our past year's effort on transportation domain issues, the original general-purpose plan critic has evolved into a domain-specific TPFDD simulator. Unlike other feasibility analyzers, however, this simulator is designed to augment particular capabilities and deficiencies of the ALPS transportation domain theory.

Once the ALPS inference engine generates a schedule, the schedule is passed along to the simulator. The simulator performs two primary services. First, it analyzes the schedule at a finer level of detail than the inference engine did. This analysis allows the simulator to identify resource contentions and bottlenecks that the inference engine would have missed. Second, the simulator can test the schedule for robustness in the presence of unanticipated difficulties by simulating nondeterministic external events that may affect the outcome of the schedule (such as storms, mechanical failures, or terrorist activity). Information gained from the simulation is sent to the ALPS plan repair module, which attempts to correct any flaws that have been discovered.

The ALPS simulator is based on an object-oriented design implemented in C++. The simulator takes as input the initial world state (locations of cargos, allocation of transportation assets, etc.) that was given to the inference engine, along with the schedule that the inference engine generated. It constructs a stream of events and executes these events in a simulated world, reporting the results of this simulation.

Our design considers the asynchronous nature of transportation problems. In the real world, transport events may be delayed due to bottlenecks such as high demand for runways at airports. From a simulation perspective, we see the most important features of a transport event as the dependencies between it and other transport events, rather than the exact time at which the event is scheduled to occur. Consider this example: We plan for aircraft $A1$ to deliver a cargo to airport $P1$ at time T. We also plan for aircraft $A2$ to pick up the same cargo at $P1$ at time $T + 2$ hours, then carry it to airport $P2$. If it happens that the arrival of $A1$ is delayed for 4 hours, we need to know that the departure of $A2$ should be delayed until the arrival of the cargo carried by $A1$.

[1]This chapter was adapted from work presented in [20].

Our general structure of plans embraces the relations between transport events, so we gain access to automatic and computer-aided reasoning for future plan repair.

10.1 Simulator Design

The ALPS simulator uses powerful object-oriented techniques implemented in C++. The main data structures are

- a **state tree** of C++ objects representing the state of the physical objects such as aircraft, ships, and cargos (their locations and arrival times);
- a **process graph** representing the dependencies between scheduling steps as a directed acyclic graph (DAG), with markers representing which steps have been executed;
- a **process queue** containing the events (or **processes**) currently under execution, sorted according to their completion times.

Processes are treated as transformations on the state space. They simulate the movements of objects from one place to another and are represented by actual movements of nodes in the state tree. For instance, loading a cargo from an airport onto a transport plane is represented by removing it from the list of cargos held by the airport and placing it on the list of cargos held by the plane. Processes are represented by nodes in the DAG. An edge from p_1 to p_2 means that p_1 must complete before p_2 can start (i.e., that p_2 *depends on* p_1).

The process graph depicting the dependencies between transportation events is constructed directly from the schedule produced by the inference engine. The graph consists of a number of "chains" (one for each punit)[2] as illustrated in Figure 10.1. The basic order is that a punit will repeatedly load cargo, take off, fly to its destination, land, and unload. In some cases, where the punit must first go to another airport to load the cargo, the missing loading/unloading events will be represented by a dummy node. Since a punit is composed of multiple aircraft or ships, there are periodic synchronization events where each aircraft will take off individually and rendezvous before flying to the next location (similarly for landing).

Before a process is placed on the process queue, the simulator determines how much time the process should require. This determination is based primarily on the domain knowledge (airspeeds and distances) supplied by the inference engine, but we have deliberately added more fine-grained knowledge to the simulator; for example, the scheduler assumes a constant time duration for unloading an aircraft, while the simulator calculates a proportional duration based on the amount of cargo. The resulting process duration is added to the current time to give the expected completion time. The *actual* completion time may differ from this estimate because of delays, resource contention, and external events. The current time is the completion time of the most recently completed process.

The process queue is initially loaded with the first event on each punit's process chain. The simulator then goes through the following loop until the process queue is empty:

1. Get from the queue the process p that is the next to complete.
2. Set the current time equal to the completion time of p.
3. Transform the state as determined by p.

[2] A punit (pronounced "pee-unit") is a collection of aircraft or ships of the same type, based at the same location, that travel together — roughly equivalent to a squadron.

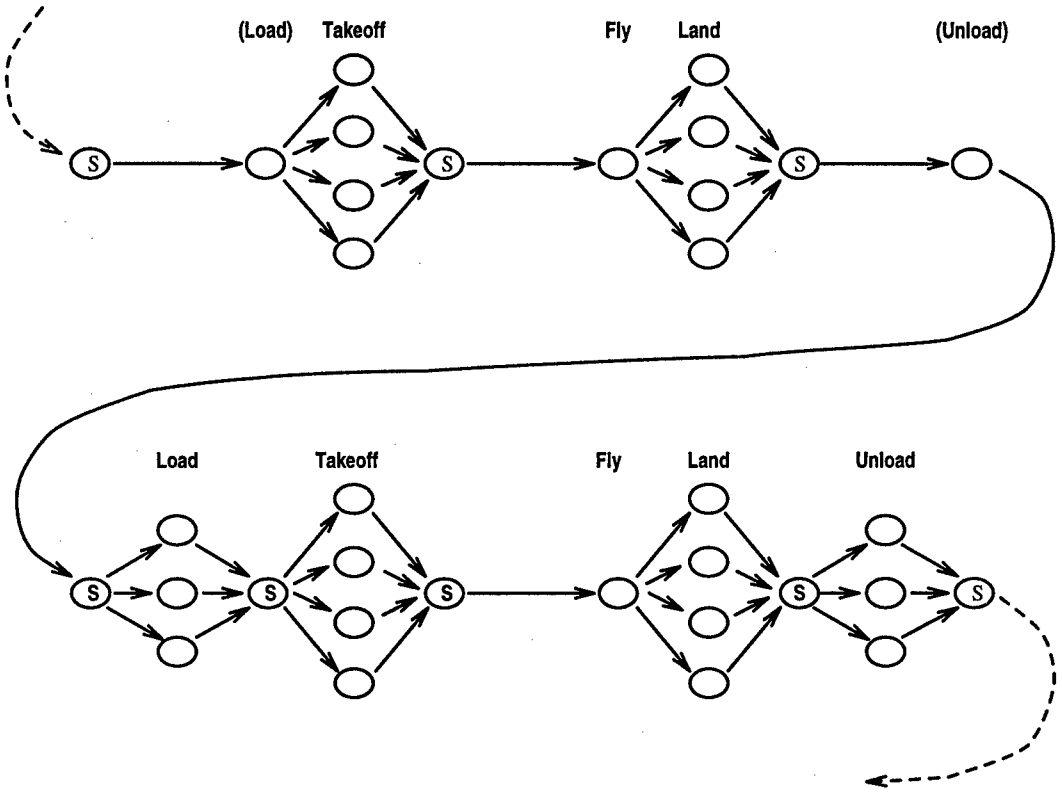

Figure 10.1: The ALPS simulator process queue.

4. For each process p_i that depends on p, signal p_i that p_i is no longer waiting for p. If p was the last process for which p_i was waiting, determine the completion time for p_i, then put p_i on the queue.

10.2 Process Queue Details

To help explain how the process queue works, we will run through a typical example starting from when the parser first reads the input file.

1. When the simulator reads a complete vehicle schedule from the scheduler output, it forms a chain in the process graph (as illustrated in Figure 10.1) and invokes the appropriate start function to begin simulation of that chain.

2. Each process type (for example loading or unloading cargo) has a different start function. The first thing that a start function does is check whether the minimum start time for that action has arrived yet; if it has not, the simulator starts a delay process, which will wake up at the appropriate time and restart the original process.

3. Once the minimum start time has arrived, the start function makes requests to various resource managers for any resources it needs. When each resource manager awards its resource, it will re-invoke the process's start function, and eventually all resources will be available. At that time, the process calculates its finish time, actually enters the queue, and starts executing (causing a print manager to issue a message to the simulation log).

4. Once the process is in the queue, it is considered to be actively executing. When the finish time for the process is reached, the print manager issues a message to the simulation log, the process is removed from the queue, and all of the process' children in the process tree are notified that one of their dependencies has completed. If any of these children now have all of their parents satisfied, those children's start functions are invoked and the procedure repeats.

10.3 Simulation of Bottlenecks

We simulate bottlenecks using **monitors**. Monitors are implemented as C++ objects, each of which manages a set of resources. Consider the simple example of a runway monitor. Each airport has a monitor that manages the use of its runways. When a plane-landing process is ready to begin, it sends to the airport's runway monitor a request for a runway. If a runway is available, the monitor assigns the runway to the process, which can then begin executing; otherwise the process is placed on a queue maintained by the monitor. When a process gives up a runway and the monitor's queue is not empty, the monitor assigns the runway to one of the waiting processes, which can then begin. Otherwise the runway is made available for new requests.

This simple model implies an approximate simulation of delays due to load on an airport. It also supports the collection of resource utilization statistics that can be fed back to the planner for plan refinement and repair. By making use of this information, a planner could avoid the over-scheduling of resources without generating fragile, excessively detailed plans. The monitor abstraction is general enough to simulate models of resource management more complex than the above example.

10.4 Simulation of External Events

In addition to simulating the dynamic but predictable behavior of resource bottlenecks, the ALPS TPFDD Scheduler is able to test a schedule for the effects of *unexpected* external events. For example, between the time that a schedule is generated by ALPS and the time that the schedule is actually executed, an unexpected storm might force travel speeds to change or airports to close.

Since there may be many different types of external events the user may wish to simulate (storms, earthquakes, equipment failure, terrorist activity, etc.), the ALPS Simulator models the *effects* of external events rather than the events themselves.[3] As a proof of concept, the simulator currently handles one specific effect (more may be added in the future):

> Any vehicle of type t traveling between port $p1$ and port $p2$ during time interval i must use speed s instead of its default speed.

This single effect can be used to model more complex behaviors such as weather patterns and mechanical failure by accelerating and decelerating vehicles appropriately.

Once the simulation is running, the user will want to know whether failures are caused by a particular external event or whether they are inherent in the underlying plan. Unfortunately, assigning responsibility for failures is a fairly complex problem. Even in the simplest case, where a cargo arrives late after its trip speed was modified by an external event, the simulator would have to compare the current analysis to an analysis of what would have happened if that event did not occur (since the cargo may have arrived late anyway). But each external event may also adjust the behavior of resource bottlenecks, potentially causing cascading effects throughout the schedule. For now, the only way to classify scheduling failures is to run the simulator twice (once with the external events and once without) and compare the output; any differences could be tagged with a warning that the failure *may* be a consequence of the external event modeling.[4]

[3] It should be straightforward to add some preprocessor to translate conceptual events into physical effects, but ALPS does not currently have this capability. Another natural extension would be to allow random external events, such as "create random delays using the following probability distribution".

[4] But it could also be possible that the failure is a result of nondeterministic behavior within the scheduler.

Chapter 11

Conclusions

As with any extended project, ALPS has had its share of successes and difficulties. We have reported some significant and impressive results, both from a theoretical perspective and within the specific transportation domain. We have also encountered some unexpected complications in applying the ALPS methodologies to the transportation domain. This chapter summarizes those results and discusses the lessons learned.[1]

As with any extended project, ALPS has had its share of successes and difficulties. We have reported some significant and impressive results, both from a theoretical perspective and within the specific transportation domain. We have also encountered some unexpected complications in applying the ALPS methodologies to the transportation domain. This chapter summarizes those results and discusses the lessons learned.

11.1 Speedup Techniques

Our results have shown that not only are our speedup techniques effective across a wide range of problems, but these techniques show even greater strength in combination than their individual performance might imply.

11.1.1 Caching

We have performed several experiments to measure the effectiveness of different caching configurations [90]. These experiments used 26 randomly ordered blocks-world problems.[2] While an unlimited-size cache provides the maximum reduction in the number of nodes explored in search of a solution, it causes an overall *decrease* in performance due to increased overhead. Similarly, although removing redundant cache entries reduces the number of expanded nodes even further, the additional overhead involved overwhelms any runtime benefit. For the particular domains tested, the best runtime performance came from a single fixed-size cache using a modified LRU policy with both success and failure entries; this configuration was more than 35% faster than an identical non-caching system.

[1] This chapter is adapted from [21].

[2] Note that these results are dependent on the particular domain theory used in the experiments; these results should be taken as indicative of what *might* be achieved in other domains rather than a promise of what *will* be achieved.

11.1.2 EBL

We have performed a series of experiments comparing our EBL*DI algorithm with the traditional EBL algorithm to determine whether macro-operators produced by EBL*DI reliably outperform macro-operators acquired by traditional EBL across a spectrum of application domains [88]. The results of the experiments are as follows:

- For the blocks world domain with carefully selected training sets, EBL*DI is significantly faster on average than EBL and solved the test problems in as little as 32% of the time while searching as few as 33% of the nodes searched by an otherwise equivalent non-learning system. This encouraging result is offset somewhat by the fact that selection of the training set was critical; training sets consisting of randomly chosen problems typically do not give speedup in this domain.
- For a propositional calculus domain (the "Logic Theorist" domain), again with carefully ordered training sets, EBL*DI solved significantly more problems within a fixed time limit than equivalent non-learning and traditional EBL systems. It searched far fewer nodes (17% of those searched by the non-learning system) and was also faster (CPU time ratio of about 13%) than the other systems.
- For a synthetic domain theory with random uniformly distributed problems, both traditional EBL and EBL*DI solved fewer problems within a fixed time limit than an equivalent non-learning system.[3] However, EBL*DI learned intrinsically more useful macro-operators than EBL, so was able to better mitigate the adverse effects of the utility problem. Furthermore, of the problems that could be solved, EBL*DI took only 20–30% as much time as the non-learning system to solve each problem.

11.1.3 Nagging

Nagging has proven itself to be an exceptionally powerful speedup technique. We have conducted experiments [97] involving over 1000 first-order logic problems [108] in a variety of domains including planning, logic, graph theory, algebra, program verification, circuit design, and classical AI benchmarks. Comparing the base-level DALI inference engine on a single workstation to a DALI configuration with caching, intelligent backtracking, and a network of 99 nagging subprocessors, the network was able to solve 34% more problems within a fixed resource constraint. On the problems that were solved by both systems, the nagging network outperformed the base system 80% of the time. An unexpected result is that several problems demonstrated superlinear speedup (i.e., the problems were solved more than 100 times faster with 100 processors). More recent experiments [102] confirm and extend these results, allowing us to evaluate several extensions and refinements to the nagging protocol that yield even greater performance improvements.

The development of nagging is perhaps the single most important technical advance obtained in the course of our research on inferential systems. Our tests on 100 processors constituted, at the time, the single largest theorem proving experiment on record. Since that time, our continued work has led to a number of refinements to the naive nagging model that effectively enhance its performance in this broad cross-section of domains. Nagging is a truly novel development that permits us to exploit existing loosely-coupled computational resources to solve large, practical problems that are beyond our reach without nagging.

[3] This is expected since the utility problem is most severe when the problem set is uniformly distributed.

Our results in first-order inference have been so encouraging that we have begun developing nagging implementations in other domains such as alpha-beta minimax, the Traveling Salesman problem, and learning of Bayesian inference networks[60]. Our continuing work on nagging includes instantiation of the basic protocol in these and other domains as well as further refinement to the first-order model.

11.1.4 Multiple Techniques

We have performed a set of experiments to study the effects of combining multiple speedup techniques [90, 12]. Specifically, we studied fixed-overhead success/failure caching with the EBL*DI algorithm, the nagging algorithm, and the iterative strengthening algorithm.

While the caching-only system provided a more or less uniform speedup across all problems, the results for the learning-only system were highly dependent on the ordering of the problems. If the problems were ordered appropriately, then the macro-operators learned during training are very useful and significantly fewer nodes are expanded. Otherwise, the learning system ran into the utility problem and the new macro-operators served only to increase redundant search on some of the problems.

When both caching and learning were used together, the benefits from caching reduced the effects of the utility problem to such an extent that, independent of which problems were selected for learning, the combined caching and learning system searched significantly fewer nodes than the base system. In fact, the combined system performed almost as well as the theoretical limit for an *unbounded*-overhead caching system, even if we disregard the extreme storage and searching overhead of unbounded caching.

EBL*DI and subgoal caching work so well together because caching serves to prune the redundant search that EBL*DI may introduce through the utility problem. Caching does the same thing with the iterative strengthening optimization algorithm, speeding up the repeated passes through the search space that are introduced by the iteration.

Using caching and nagging together requires additional coordination, and the rapid backtracking from nagging may decrease the efficiency of global caching, but the added ability of each processor to set its own local customized caching policy will often make up for this.

11.1.5 A Paradox

While the results we have reported for nagging, EBL*DI, bounded-overhead subgoal caching, and combinations of these speedup techniques are quite impressive from a research perspective, it is clear that these techniques were not as effective as we had hoped in the transportation scheduling domain (see Section 8). What can we learn from this paradox?

All of these speedup techniques expect that the domain theory given is inclusive, that is, that it can be used to solve *any* transportation scheduling problem, using *any* initial partial problem specification to derive legal solutions. In other words, the theory is assumed to be logically complete, yet devoid of control information about how the theory should be used (in classic Prolog terminology, such theories are described as "logic without control"). Speedup learning and nagging are meant to provide a means to execute such logically complete but control-poor theories efficiently and effectively. The domains we used for our experimental evaluations of nagging and speedup learning, which are taken from the machine learning and theorem proving literature, are good examples of "logic without control." Based on our evaluations, our speedup techniques clearly meet their original promise, allowing us to use the theories effectively to solve very large scale problems using large numbers of workstations.

In practice, however, arbitrary partial specifications are *not* usually given as input; indeed, we generally know a great deal about how the theory is to be used to derive a solution. Thus, in almost every case, it is possible to hand-tune solutions that take advantage of this domain-dependent control information (which is, unfortunately, not stated explicitly) to derive highly efficient solutions at the expense of generality. In particular, the transportation planning domain theories we developed are definitely not "control free." By hand-tuning our domain theory and adding extra-theoretical control knowledge, we made the theory more effective than any speedup technique from the start.

It is interesting to note that we did not initially intended to develop a transportation domain theory during this project. Instead, we hoped to obtain an existing logical specifications of the domain from a domain expert. If that had been possible, we suspect that such a declarative specification would have been much closer to logic without control, and our speedup techniques would have been more effective (but see Section 8).

11.2 Transportation Planning and the Fast Scheduler

When we first started using ALPS in the full TPFDD transportation scheduling domain, the logical domain theory worked quite well on small problems. But neither the Lisp Inference Engine nor DALI was able to handle full-sized transportation problems with tens of thousands of cargos, and the adaptive speedup techniques were not enough to overcome the huge scaleup issues (see Section 11.1.5). To overcome this difficulty, we created the ALPS Fast Scheduler by hand-translating the logical domain theory directly into straight Lisp code. Although the Fast Scheduler lost many of the adaptive properties and the reusability of the core inference engines, it gained a dramatic increase in speed and speedup: we have solved problems with 50 squadrons of aircraft and 10,000 cargos in about 3.5 minutes. This result more than justifies the loss of some speedup techniques that, while very effective in other domains, were not producing significant speedup in this domain anyway.

Our conclusions from this experiment are that in some performance-critical domains such as the transportation domain, while logical specifications work fine in principle, in practice it may be necessary to perform the additional step of compiling to an executable program. The important point is that from a *logical* perspective, all three inference engines do exactly the same work and produce exactly the same results; the performance differences can be thought of as implementation details.

Our intuition is that trying to write the fast scheduler from scratch would have been more difficult and time-consuming than adapting an existing logical domain theory; when we had this theory, it took only about a day to do the basic translation. In fact, rather than viewing our TPFDD scheduler as a separate program, it may be more appropriate to think of it as just another step in the compilation of a system, where in this case much of the compilation was done by hand. In the future, we hope to be able to automate some of the work required in generating a "hand-crafted" theory, but we suspect that automated techniques may be effective only on certain classes of domains.

Appendix A

Acronyms

This appendix defines the acronyms used in this report.

AI	Artificial Intelligence
ALPS	Adaptive Learning and Planning System
ARPA	Advanced Research Projects Agency
ARPI	ARPA / Rome Laboratory Planning Initiative
CLRU	Cheapest Least Recently Used
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DALI	Distributed Adaptive Logical Inference
DLRU	Dearest Least Recently Used
EAD	Earliest Arrival Date
EBL	Explanation-Based Learning
EBL*DI	Explanation-Based Learning / Domain Independent
FIFO	First In First Out
GEOLOC	GEOgraphic LOCation
GPR	General Plan Repair
GUI	Graphical User Interface
LAD	Latest Arrival Date
LFU	Least Frequently Used
LRU	Least Recently Used
LT	Logic Theorist
MB	MegaByte
MPR	Multiple Plane Repair
NUMA	Non-Uniform Memory Access
ORA	Odyssey Research Associates, Inc.
POD	Port of Debarkation
RAM	Random Access Memory
RDD	Required Delivery Date
SPR	Single Plane Repair
TGEN	TPFDD Generator
TPFDD	Time Phased Force Deployment Data
TPTP	Thousands of Problems for Theorem Provers
WAM	Warren Abstract Machine

Appendix B

Bibliography

[1] H. Aït-Kaci. *Warren's Abstract Machine.* MIT Press, 1991.

[2] P.E. Allen, S. Bose, Edmund M. Clarke, and S. Michaylov. Parthenon: A parallel theorem prover for non-horn clauses. In Ewing L. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 764–765. Springer-Verlag, May 1988.

[3] Armed Forces Staff College. The joint staff officer's guide 1991. Technical Report AFSC-PUB1, Armed Forces Staff College, 1991.

[4] O.L. Astrachan and D.W. Loveland. Meteors: High performance theorem provers using model elimination. In R. S. Boyer, editor, *Automated Reasoning, Essays in Honor of Woody Bledsoe*, pages 31–59. Kluwer, 1991.

[5] Neeraj Bhatnagar and Jack Mostow. On-line learning from search failures. *Machine Learning*, 15(1):69–117, 1994.

[6] M.S. Braverman and S.J. Russell. Boundaries of Operationality. In *Proceedings of the Fifth International Machine Learning Conference*, pages 221–234, Ann Arbor, MI, June 1988.

[7] M.S. Braverman and S.J. Russell. IMEX: Overcoming Intractability in Explanation Based Learning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 575–579, St. Paul, MN, July 1988.

[8] Stefan Brüning. Detecting non-provable goals. In Alan Bundy, editor, *12th International Conference on Automated Deduction*, pages 222–236. Springer-Verlag, June 1994.

[9] M. Bruynooghe. Intelligent backtracking for an interpreter of horn clause logic programs. In B. Dömölki and T. Gergely, editors, *Mathematical Logic in Computer Science*, pages 215–258. North-Holland, 1978.

[10] Alan Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, 1983.

[11] Randall J. Calistri. *Classifying and Detecting Plan-Based Misconceptions for Robust Plan Recognition.* Ph.D. dissertation, Department of Computer Science, Brown University, Providence, Rhode Island, May 1990.

[12] Randall J. Calistri-Yeh. Generating optimal plans in ALPS: A preliminary report. Technical Report TM-93-0054, ORA, Ithaca, New York, November 1993.

[13] Randall J. Calistri-Yeh. Iterative strengthening: An algorithm for generating anytime optimal plans. In *Proceedings of the ARPA / Rome Laboratory Knowledge Based Planning and Scheduling Initiative Workshop (ARPI-94)*, pages 3–13, Tucson, Arizona, February 1994.

[14] Randall J. Calistri-Yeh. Iterative strengthening: An algorithm for generating anytime optimal plans. In *Proceedings of the Sixth International Conference on Tools with Artificial Intelligence*, pages 728–731, New Orleans, Louisiana, November 1994. IEEE Computer Society Press.

[15] Randall J. Calistri-Yeh and Alberto M. Segre. The design of ALPS: An adaptive architecture for transportation planning. Technical Report TM-93-0010, ORA, Ithaca, New York, April 1993.

[16] Randall J. Calistri-Yeh and Alberto M. Segre. ALPS: An adaptive learning and planning system. In *Proceedings of the TECOM Artificial Intelligence Technology Symposium*, pages 309–326, September 1994.

[17] Randall J. Calistri-Yeh and Alberto M. Segre. ALPS year-1 annual report. Technical Report TM-94-0011, ORA, Ithaca, New York, February 1994.

[18] Randall J. Calistri-Yeh and Alberto M. Segre. The design of ALPS: An adaptive learning and planning system. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pages 207–212, Chicago, Illinois, June 1994.

[19] Randall J. Calistri-Yeh and Alberto M. Segre. The design of ALPS: An adaptive learning and planning system. In *Proceedings of the ARPA / Rome Laboratory Knowledge Based Planning and Scheduling Initiative Workshop (ARPI-94)*, pages 251–262, Tucson, Arizona, February 1994.

[20] Randall J. Calistri-Yeh and Alberto M. Segre. ALPS year-2 annual report. Technical Report TM-95-0013, ORA, Ithaca, New York, February 1995.

[21] Randall J. Calistri-Yeh, Alberto Maria Segre, and David Sturgill. The peaks and valleys of ALPS: an adaptive learning and planning system for transportation scheduling. In A. Tate, editor, *Advanced Planning Technology — Technological Achievements of the ARPA/Rome Laboratory Planning Initiative (AIPS-96 Advanced Technology Supplement)*. AAAI Press, May 1996. To appear.

[22] C-L. Chang and R.C-T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[23] J-H. Chang and A.M. Despain. Semi-intelligent backtracking of Prolog based on static data dependency analysis. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 10–21. IEEE Computer Society, July 1985.

[24] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[25] K. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, NY, 1978.

[26] W.F. Clocksin. Principles of the DelPhi parallel inference machine. *Computer Journal*, pages 386–392, 1987.

[27] Steven Cross. A proposed initiative in crisis action planning. Technical report, ISTO/DARPA, Arlington, Virginia, May 1990.

[28] T.R. Davies and S.J. Russell. A Logical Approach to Reasoning by Analogy. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 264–270, Milan, Italy, August 1987.

[29] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, St. Paul, Minnesota, August 1988.

[30] Tom Dean. Technical description of the transportation problem. Created for the DARPA/RL Knowledge-Based Planning and Scheduling Initiative, March 1992.

[31] D. DeGroot. Restricted and-parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 471–478. North-Holland, November 1984.

[32] G. DeJong and R. Mooney. Explanation-Based Learning: An Alternative View. *Machine Learning*, 1(2):145–176, 1986.

[33] S.A. Delgado-Rannauro. Or-parallel logic computational models. In P. Kacsuk and M. J. Wise, editors, *Implementations of Distributed Prolog*, pages 3–26. John Wiley & Sons, 1992.

[34] S.A. Delgado-Rannauro. Stream and-parallel logic computational models. In P. Kacsuk and M. J. Wise, editors, *Implementations of Distributed Prolog*, pages 239–257. John Wiley & Sons, 1992.

[35] T. Dietterich. Machine Learning. *Annual Review of Computer Science*, 4:255–306, 1990.

[36] T.G. Dietterich. Learning at the Knowledge Level. *Machine Learning*, 1(3):287–316, 1986.

[37] C. Elkan. Conspiracy Numbers and Caching for Searching And/Or Trees and Theorem-Proving. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 341–348, Detroit, MI, August 1989.

[38] C. Elkan and D. McAllester. Automated Inductive Reasoning about Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 876–892, Cambridge, MA, August 1988. MIT Press.

[39] W. Ertel. Random competition: A simple but efficient method for parallelizing inference systems. In *Parallelization in Inference Systems*, pages 195–209. Springer-Verlag, December 1990.

[40] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[41] A. Ginsberg. Theory Reduction, Theory Revision, and Retranslation. In *Proceedings of the National Conference on Artificial Intelligence*, pages 777–782, Boston, MA, July 1990.

[42] Matthew L. Ginsberg and William D. Harvey. Iterative broadening. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 216–220, Boston, Massachusetts, August 1990.

[43] G.J. Gordon and A.M. Segre. Nonparametric statistical methods for experimental evaluations of speedup learning. Bari, Italy, January 1996. Submitted to The Thirteenth International Machine Learning Conference.

[44] C. Green. Application of Theorem Proving to Problem Solving. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, pages 67–87. Morgan Kaufmann, San Mateo, CA, 1990.

[45] Kristian J. Hammond. Explaining and repairing plans that fail. *Artificial Intelligence*, 45:173–228, 1990.

[46] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, 1968.

[47] M.V. Hermenegildo. An abstract machine for restricted and-parallel execution of logic programs. In E. Shapiro, editor, *Third International Conference on Logic Programming*, pages 25–39. Springer-Verla, July 1986.

[48] H. Hirsh. Reasoning About Operationality for Explanation-Based Learning. In *Proceedings of the Fifth International Machine Learning Conference*, pages 214–220, Ann Arbor, MI, June 1988.

[49] Adele E. Howe and Paul R. Cohen. Failure recovery: A model and experiments. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 801–808, Anaheim, California, August 1991.

[50] Joint Data Systems Support Center. *Joint Operation Planning System (JOPS) Time Phased Force Deployment Data (TPFDD) and Related Files: Data Base Specification*, SPM DS 143-87 edition, April 1987.

[51] Subbarao Kambhampati. Mapping and retrieval during plan reuse: A validation structure based approach. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 170–175, Boston, Massachusetts, August 1990.

[52] Subbarao Kambhampati. A theory of plan modification. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 176–182, Boston, Massachusetts, August 1990.

[53] Subbarao Kambhampati and James A. Hendler. Control of refitting during plan reuse. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 943–948, Detroit, Michigan, 1989.

[54] Henry Kautz and Bart Selman. An empirical evaluation of knowledge compilation by theory approximation. In *Proceedings of the AAAI-94*, pages 155–161. MIT Press, August 1994.

[55] R.M. Keller. Defining Operationality for Explanation-based Learning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 482–487, Seattle, WA, July 1987.

[56] M. Koppel, R. Feldman, and A.M. Segre. Bias-driven revision of logical domain theories. *Journal of Artificial Intelligence Research*, 1:159–208, February 1994.

[57] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[58] V. Kumar and Y-J. Lin. An intelligent backtracking scheme for Prolog. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 406–414. IEEE Computer Society, August 1995.

[59] F. Kurfeß. Potentiality of parallelism in logic. In *Parallelization in Inference Systems*, pages 3–25. Springer-Verlag, December 1990.

[60] W. Lam and A.M. Segre. Distributed learning of bayesian inference networks. Bari, Italy, January 1996. Submitted to The Thirteenth International Machine Learning Conference.

[61] E.L. Lawler and D.E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[62] R. Letz, K. Mayr, and C. Goller. Controlled integration of the cut rule into connection tableau calculi. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[63] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. A high-performance theorem prover. *Journal of Automated Reasoning*, 13(3):183–212, 1994.

[64] J.W. Lloyd. *Foundations of Logic Programming, Second Edition.* Springer Verlag, 1987.

[65] D.W. Loveland. A unifying view of some linear Herbrand procedures. *Journal of the Association for Computing Machinery*, 19:366–384, 1972.

[66] D.W. Loveland. *Automated Theorem Proving: a Logical Basis.* North-Holland, 1978.

[67] S. Mahadevan. Using Determinations in EBL: A Solution to the Incomplete Theory Problem. In *Proceedings of the Sixth International Machine Learning Workshop*, pages 320–325, Ithaca, NY, June 1989.

[68] John McCarthy. Programs with common sense. In M.L. Minsky, editor, *Semantic Information Processing*, pages 403–409. MIT Press, Cambridge, Massachusetts, 1968.

[69] S. Minton. Quantitative Results Concerning the Utility of Explanation-Based Learning. In J. Shavlik and T. Dietterich, editors, *Readings in Machine Learning*, pages 573–587. Morgan Kaufmann, San Mateo, CA, 1990.

[70] S. Minton. Quantitative Results Concerning the Utility of Explanation-Based Learning. *Artificial Intelligence*, 42(2-3):363–392, March 1990.

[71] T.M. Mitchell, R.M. Keller, and S.T. Kedar-Cabelli. Explanation-Based Generalization: A Unifying View. *Machine Learning*, 1(1):47–80, 1986.

[72] R. Mooney. The Effect of Rule Use on the Utility of Explanation-Based Learning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 725–730, Detroit, MI, August 1989.

[73] R. Mooney and S. Bennett. A Domain Independent Explanation-Based Generalizer. In *Proceedings of the National Conference on Artificial Intelligence*, pages 551–555, Philadelphia, PA, August 1986.

[74] J. Mostow. A Problem-Solver for Making Advice Operational. In *Proceedings of the National Conference on Artificial Intelligence*, pages 279–283, Washington, DC, August 1983.

[75] J. Mostow. Searching for Operational Concept Descriptions in BAR, MetaLEX, EBG. In *Proceedings of the Fourth International Machine Learning Workshop*, pages 376–382, Irvine, CA, June 1987.

[76] Bernhard Nebel and Jana Koehler. Plan modification versus plan generation: A complexity-theoretic perspective. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1436–1441, Washington, DC, August 1993.

[77] A. Newell, J.C. Shaw, and H. Simon. Empirical Explorations with the Logic Theory Machine: A Case Study in Heuristics. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*. McGraw Hill, New York, NY, 1963.

[78] P. O'Rorke. LT Revisited: Explanation-Based Learning and the Logic of Principia Mathematica. *Machine Learning*, 4(2):17–160, November 1989.

[79] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[80] D. Plaisted. Non-Horn Clause Logic Programming Without Contrapositives. *Journal of Automated Reasoning*, 4:287–325, 1988.

[81] David A. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16:47–108, 1981.

[82] A.E. Prieditis and J. Mostow. PROLEARN: Towards a Prolog Interpreter that Learns. In *Proceedings of the National Conference on Artificial Intelligence*, pages 494–498, Seattle, WA, July 1987.

[83] Foster John Provost. Iterative weakening: Optimal and near-optimal policies for the selection of search bias. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 749–755, Washington, DC, August 1993.

[84] Glen A. Reece and Austin Tate. Synthesizing protection monitors from casual structure. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pages 146–151, Chicago, Illinois, June 1994.

[85] R. Reiter. On Closed World Data Bases. In B.L. Webber and N. Nilsson, editors, *Readings in Artificial Intelligence*, pages 119–140. Morgan Kaufmann, San Mateo, CA, 1981.

[86] Johann M. Schumann and R. Letz. Partheo: A high-performance parallel theorem prover. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 40–56. Springer-Verlag, July 1990.

[87] Johann M. Schumann, R. Letz, and F. Kurfess. Tutorial on high-performance theorem provers: Efficient implementation and parallelism. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, page 683. Springer-Verlag, July 1990.

[88] Alberto Segre and Charles Elkan. A high performance explanation-based learning algorithm. *Artificial Intelligence*, 69(11):1–50, September 1994.

[89] Alberto Segre and Daniel Scharstein. Bounded-overhead caching for definite-clause theorem proving. *Journal of Automated Reasoning*, pages 83–113, August 1993.

[90] Alberto M. Segre. Evaluating adaptive inference. Technical Report TM-93-0044, ORA, Ithaca, New York, September 1993.

[91] A.M. Segre. On The Operationality/Generality Trade-off in Explanation-Based Learning. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 242–248, Milan, Italy, August 1987.

[92] A.M. Segre. *Machine Learning of Robot Assembly Plans.* Kluwer Academic, Boston, MA, March 1988.

[93] A.M. Segre. Learning How to Plan. *Robotics and Autonomous Systems*, 8(1-2):93–111, November 1991.

[94] A.M. Segre, C. Elkan, and A. Russell. Technical Note: A Critical Look at Experimental Evaluations of EBL. *Machine Learning*, 6(2):183–196, March 1991.

[95] A.M. Segre, C.P. Elkan, G.J. Gordon, and A. Russell. A Robust Methodology for Experimental Evaluations of Speedup Learning. Technical report, Department of Computer Science, Cornell University, Ithaca, NY, October 1991. Working Paper.

[96] A.M. Segre, G.J. Gordon, and C.P. Elkan. Exploratory analysis of speedup learning data using expectation maximization. *Artificial Intelligence*, 1996.

[97] A.M. Segre and D.B. Sturgill. Using hundreds of workstations to solve first-order logic problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 187–192, Seattle, Washington, August 1994.

[98] Bart Selman and Henry Kautz. Knowledge compilation using horn approximations. In *Proceedings of the AAAI-91*, pages 904–909, 1991.

[99] 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, Minnesota, August 1988.

[100] Mark E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4:353–380, 1988.

[101] M.E. Stickel and W.M. Tyson. An analysis of consecutively bounded depth-first search with applications in automated deduction. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 1073–1075, August 1985.

[102] David Sturgill and Alberto Segre. Nagging: a distributed, adversarial search-pruning technique applied to first-order inference. Technical Report TM-96-0003, ORA, Ithaca, New York, February 1996. Also submitted to Journal of Approximate Reasoning, December 1995.

[103] David B. Sturgill. *Nagging: a New Approach to Parallel Search Pruning.* PhD thesis, Cornell University, 1996.

[104] D.B. Sturgill and A.M. Segre. A novel asynchronous parallelization scheme for first-order logic. In *Proceedings of the Twelfth Conference on Automated Deduction*, pages 484–498, Nancy, France, June 1994.

[105] D. Subramanian and R. Feldman. The Utility of EBL in Recursive Domain Theories. In *Proceedings of the National Conference on Artificial Intelligence*, pages 942–951, Boston, MA, July 1990.

[106] G.J. Sussman. A Computational Model of Skill Acquisition. Technical Report 297, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1973.

[107] Geoff Sutcliffe, Christian Suttner, and Theodor Yemenis. The TPTP problem library. In Alan Bundy, editor, *12th International Conference on Automated Deduction*, pages 252–266. Springer-Verlag, June 1994.

[108] C.B. Suttner, G. Sutcliffe, and T. Yemenis. The TPTP problem library (TPTP v1.0.0). Technical Report FKI-184-93, Institut für Informatik, Technische Universität München, Munich, Germany, 1993.

[109] M. Tambe and A. Newell. Some Chunks Are Expensive. In *Proceedings of the Fifth International Machine Learning Conference*, pages 451–458, Ann Arbor, MI, June 1988.

[110] Roy M. Turner. Modifying previously-used plans to fit new situations. In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, pages 1–10, Seattle, Washington, July 1987.

[111] L.G. Valiant. A Theory of the Learnable. *Communications of the Association for Computing Machinery*, 27(11):1134–1142, 1984.

[112] F. van Harmelen and A. Bundy. Explanation-Based Generalisation = Partial Evaluation (Research Note). *Artificial Intelligence*, 36(3):401–412, October 1988.

[113] D.H.D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, October 1993.

[114] Dan Weld. Mixed-initiative planning. In *Proceedings of the ARPA / Rome Laboratory Knowledge Based Planning and Scheduling Initiative Workshop (ARPI-94)*, page 518, Tucson, Arizona, February 1994.

[115] A.N. Whitehead and B. Russell. *Principia Mathematica.* Cambridge University Press, Cambridge, England, 1913.

[116] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann Publishers, Inc., Los Altos, California, 1988.

[117] Yunshan Zhu and Randall J. Calistri-Yeh. Iterative plan repair in ALPS. Technical Report TM-95-0067, ORA, Ithaca, New York, September 1995.

☆U.S. GOVERNMENT PRINTING OFFICE: 1997–509–127–47120

MISSION OF ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.