NAGGING: A GENERAL, FAULT-TOLERANT APPROACH TO PARALLEL SEARCH PRUNING

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
David Brian Sturgill
January 1997

NAGGING: A GENERAL, FAULT-TOLERANT APPROACH TO PARALLEL SEARCH PRUNING

David Brian Sturgill, Ph.D.

Cornell University 1997

For some interesting problems, all known algorithms rely, to some degree, on exhaustive search. Since combinatorial search cannot scale to large problem instances, no general-case solutions to these problems are available. However, because solutions to many of these problems have practical value, various software techniques have been developed to avoid or reduce search in a number of useful, special cases. Unfortunately, different software techniques exhibit varying performance advantages from one problem instance to the next; given a particular problem instance, it is not always clear which approach would be most effective.

This paper introduces a parallel search-pruning technique called nagging which is means of coordinating the activity of a number of different search procedures. Under this technique, search-based problem solvers compete in parallel to solve parts of a particular problem instance. Each problem solver contributes to advancing the search wherever it is the most effective.

Nagging's intrinsic fault tolerance and scalability make it particularly suitable for commonly available, low-bandwidth, high-latency distributed computing environments. It is sufficiently general to be effective in a number of domains. A prototype implementation has been developed for first-order theorem proving, a domain both responsive to a very simple nagging model and amenable to many refinements of this model. Nagging is evaluated by testing this implementation on a suite of well-known theorem proving problems.

Biographical Sketch

David Sturgill graduated from the University of South Carolina in 1989, Magna Cum Laude in Computer Science. After two years working in user interface with NCR, he returned to school pursuing the MS and Ph.D. in Computer Science at Cornell.

Extracurricular pursuits included wood carving, hiking in the Ithaca gorges and playing the trombone as well as work at Ithaca Baptist Church.

He maintains an interest in artificial intelligence/theorem proving and in the application of intelligent techniques to user interface and verification problems. In the fall of 1996, he began working as at assistant professor at Baylor University.

Acknowledgements

Thanks must go to the members of my special committee. I appreciate my chair Dr. Alberto Segre's support and advice throughout this process as well as his guidance through the Ph.D. program and into the field.

Drs. Keshav Pingali and Sam Toueg were very helpful as local committee members and Dr. Beena Khurana provided an excellent introduction to cognitive psychology in the classroom as well as advisement on the special committee.

Special thanks to Dr. Don Loveland, who suggested work in the area that has become abstraction-based nagging.

I appreciate the support of the Air Force Office for Scientific Research through a Graduate Student Fellowship.

Finally thanks to my wife for proofreading and to my family on both sides for emotional support during graduate school.

Table of Contents

List of Figures

Chapter 1

Search

Problems and problem solving are central to the definition of computer science. The field has identified many powerful techniques for classifying both problems and approaches to solving them. Naturally, there is an interest in solving as many problems as possible and, consequently, in identifying the most effective means possible for solving various types of problems. Many useful problems have been shown quite amenable to computational solution. Unfortunately, other problems of similar practical import seem disappointingly less responsive to conventional computation.

1.1 Search Problems

Search problems are those for which every known means of solution is tantamount to simply trying potential solutions until an actual solution is found. For such a problem, we may have a reliable means of recognizing valid solutions, but exhibiting such a solution may require repeated generation and testing of many non-solutions.

Consider, for example, the N-queens problem. Given a chess board of N by N squares, this problem requires that N queens be placed on the board such that no queen threatens to capture any other. No two queens can occupy the same row, column or diagonal. An attempt to solve this problem might involve simply placing queens on the board, one after another, until N of them had been placed. Figure 1.1 demonstrates such an attempt. This attempt fails to reach an actual solution since there are threats among the queens in the final board.

When placing each queen in Figure 1.1, there were many open spaces from which to choose. The particular choices made here lead to failure, but the 6-queens problem does actually have solutions. This attempt to solve it fails because some of the choices made in Figure 1.1 are incorrect.

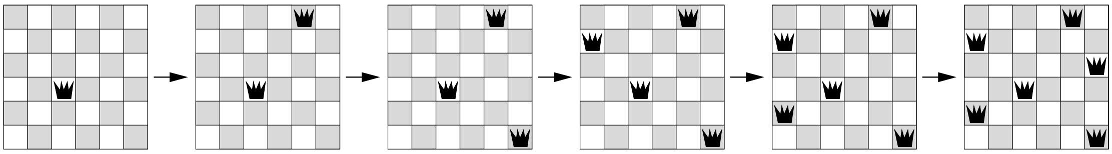
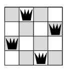


Figure 1.1: Unsuccessful attempt to solve 6-queens

Features of this N-queens example are representative of search problems in general. Constructing a solution to the problem requires that choices be made. For legitimate search problems, there is no completely reliable means of making these choices correctly. To discover a solution, it may be necessary to consider a large number of potential solutions, each generated by making a different decision at some choice point. The set of all such potential solutions is called the *search space*, and a mechanism for enumerating elements of this set and extracting actual solutions is called a *search procedure*.

1.2 Search Trees

The *search tree* is a framework for organizing the search space, identifying the choices necessary for constructing a potential solution and describing the order in which potential solutions will be examined. A search tree contains a possibly infinite number of *nodes* connected by *edges*. The nodes correspond to elements of the search space and the edges connecting them describe the possible choices from each node. The nodes satisfying the conditions of the problem statement are called *solution nodes*.

In the case of the N-queens problem, the nodes correspond to possible board configurations and the edges between them correspond to options for the placement of the next queen. Figure 1.2 presents part of a search tree for the 4-queens problem. The node at the top of the figure represents an empty 4×4 board. The nodes connected below it represent alternatives for placing the first queen. If the tree were written out in its entirety, it would contain all possible configurations that include from zero to four queens. The node  would be a solution node in this tree.

1.2.1 Notation

The search tree is a powerful paradigm for describing search problems and is used throughout this text. Relevant definitions and notation are given here, but a more thorough introduction is available in a number of texts [Joh90,CLR90].

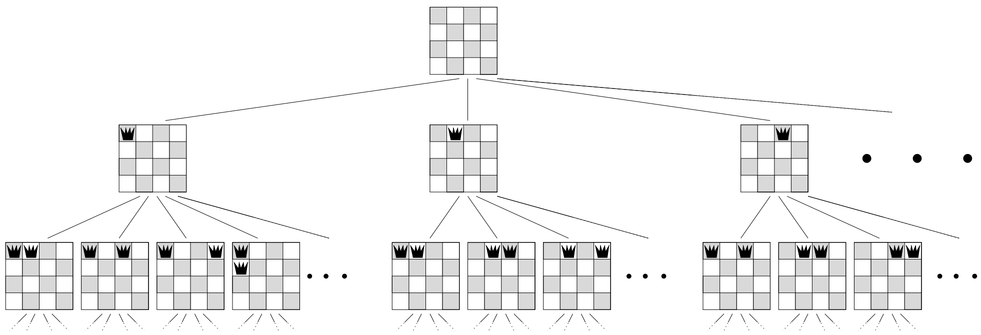
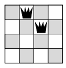
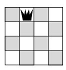
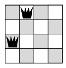
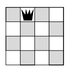
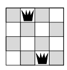
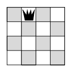


Figure 1.2: Partial, naive search tree for 4-queens.

The symbol T is used to represent a search tree, while n and its variants are used to stand for nodes in the search tree. Taking some liberty with notation, $n \in T$ is used to indicate that n is a node in the tree T. The structure of any tree is captured in its edge relation, most naturally expressed in terms of parents and children. In Figure 1.2, a node's children are those resulting from the placement of a new queen. The notation $c(n)$ is used to stand for the set of children of node n. In Figure 1.2, for example,  $\in c\Big($  $\Big)$ and  $\in c\Big($  $\Big)$. Within this text, it is assumed that search trees are *finitely branching*. This is to say that each node has some finite number (possibly zero) of children. The parent of n, written $p(n)$, is the node satisfying $n \in c(p(n))$. In Figure 1.2, $p\Big($  $\Big) =$ . Within this text, notation such as $p^2(n)$ is used as shorthand for $p(p(n))$, with $p^0(n) = n$.

With one exception, all nodes in a tree have a single parent. The node with no parent is called the *root* of the tree, and in the 4-queens example, the root is the empty board. Nodes with no children are called *leaves*. Two nodes, n_a and n_b are *siblings* if the have the same parent. The *ancestors* of n are the nodes $p^1(n), \ldots p^k(n)$ where $p^k(n)$ is the root. The *depth* of a node is the number of ancestors it has, and the n^{th} *level* of the search tree is the set of all nodes having depth n. The *height* of a tree is the number of levels it contains. For some $n \in T$ the *subtree rooted at* n, written $T(n)$, is the tree containing only n and the nodes for which n is an ancestor. A subtree is, itself, a tree, and any definitions that apply to trees will also apply to subtrees. Tree T' is called a subtree of T if $T' = T(n)$ for some $n \in T$.

1.2.2 Search Procedures

The search tree is a structured representation of the search space. The operation of many search procedures may be interpreted as an exploration of the search tree in an effort to locate solution nodes. It is important to appreciate that, when attempting to solve a problem, a search procedure starts with a description of the problem and not a copy of the search tree itself. While the search tree represents the search space explicitly, a problem description and search procedure represent it implicitly. This is a necessary distinction since it is possible to describe problems that have infinite search trees. This distinction is also reflected in the treatment of N-queens. The informal problem description given in Section 1.1 contains none of the search space elements shown in Figure 1.2. Instead, the problem description may be seen as a basis for the construction of any N-queens search tree.

The operation of a tree-search procedure is most commonly seen as a process of explicitly constructing the search tree T. A node of T is *generated* whenever it is explicitly constructed. Search begins by generating the root. The root is then *expanded* by generating its children. Nodes that have been generated but not yet expanded are said to be on the search *fringe*. In general, search is a process of repeatedly expanding nodes on the fringe until a solution is found or the fringe is emptied.

The rule used to select nodes from the fringe is the most significant feature classifying individual search procedures. The poles in this classification are represented by *depth-first search* and *breadth-first search*. Breadth-first search always selects the node that has been on the fringe the longest, while depth-first search selects the node that entered the fringe most recently. This contrast in policy has profound effect on the order in which nodes in T are visited. As Figure 1.3 indicates, depth-first order explores T a subtree at a time, while breadth-first explores it a level at a time [Win92].

Different search procedures also make different demands for storage. While exploring T it is only necessary to store nodes that are on the fringe; once a node has been expanded and checked against the solution criteria it is no longer needed. As illustrated in Figure 1.3, breadth-first search may maintain a larger fringe than depth-first. In general, the storage required by a breadth-first search may be exponentially greater than that of a depth-first search.

This economical storage requirement is balanced by the fact that depth-first search is not always capable of finding all solutions in T. As illustrated in Figure 1.4, this occurs only when T is infinite. Once depth-first search begins exploring a subtree, it does not leave the subtree until its exploration is complete. In Figure 1.4, because $T(n_b)$ is infinite, a depth-first search would never exit from $T(n_b)$ to examine the shaded nodes. If one of the shaded nodes is a solution, this solution would not be found. In general, the ability of a search procedure to find all existing solutions is called *completeness*. A dual property of *soundness* stipulates that any solutions

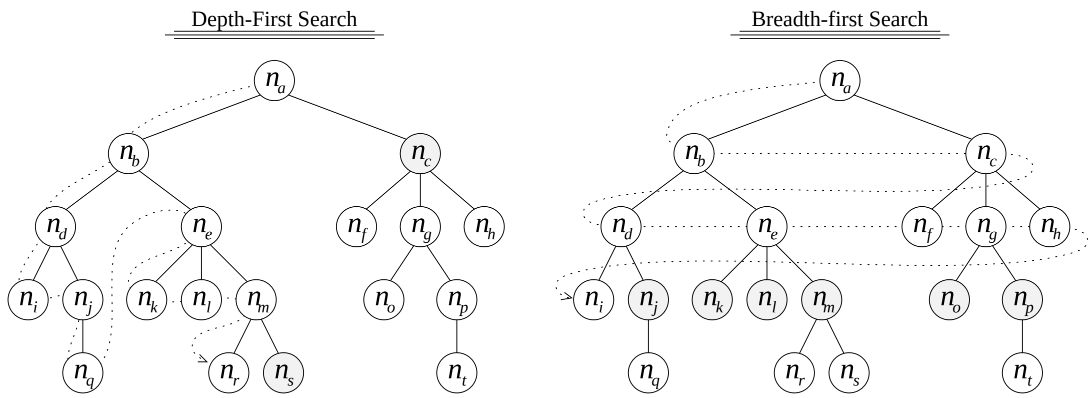


Figure 1.3: Depth-first and breadth-first search procedures. The dotted lines indicate the order in which nodes are expanded, and the shaded nodes indicate the search fringe.

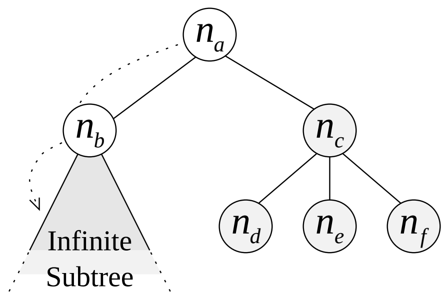


Figure 1.4: Incompleteness of depth-first search in infinite trees. Shading indicates nodes that depth-first search cannot reach.

reported by a search procedure actually satisfy the problem statement.

1.3 Resource-Bounded Search

In principle, any sound, complete search procedure is suitable for solving any search problems in a given domain. In practice, however, the problem-solving power of a search-based system is limited by constraints on the resources it can realistically be permitted to expend. Most often, time is the operative constraint as one is willing to wait only so long for a solution. Thus, problems solvable within a limited amount of time are a more representative measure of the usefulness of a search-based problem solver than are the soundness and completeness of its search procedure alone. Of course, this is the case with any problem solving system, but the restriction of the resource bound is even more acute on search-based problem solvers.

Figure 1.2 suggests why computational expense may be a particularly uncooperative feature of search problems. The abbreviation of the 4-queens search tree in Figure 1.2 indicates that this particular tree would be quite large. The first three levels would contain 257 nodes if they were filled in completely. The two omitted levels contain 3360 and 43680 nodes. For N-queens, larger boards yield still larger search spaces, while for many problems the search trees are infinite. Alone, the size of the search space alone is not as meaningful as is its relationship to the size of the problem description. Search space size may change greatly even when the problem description changes only a little. The full search tree for the 4-queens problem as outlined in Figure 1.2 contains $47,297$ nodes. A similar tree for the 6-queens problem would have more than 10^9 nodes, and one for 8-queens would have more than 10^{14}. In contrast, describing the 8-queens instance is little different from describing the 4-queens instance. Since the time required to solve search problems is dependent on the size of T, small changes in a problem's description can have radical impact on the amount of computation required to solve it.

This disparity in the problem description size and the amount of computation it demands is the reason search is not a particularly attractive way to solve problems. A problem is said to be *tractable* or efficiently solvable if the amount of computation required to solve any problem instance of size N is at most proportional to N^k for some constant k. For efficient solutions, the computation required is polynomially related to the size of the problem description; multiplicative increases in the resource bound yield multiplicative increases in the size of problem instances that can be solved. For search problems, this is not the case. Search spaces may grow exponentially or worse as the size of the problem description grows. Doubling the resource bound permits only an incremental gain in the size of solvable problems. Thus, approaches that resort to combinatorial search simply do not scale well to larger problem instances.

Figure 1.5 illustrates the impact of greater computational resources on a search-based problem solver. The search tree given in Figure 1.2 was used as the basis for an implementation of an N-queens problem solver. The top line in Figure 1.5 represents the solution times for N-queens problems of various sizes under this implementation. The next lower line represents the expected solution times on a computer that is twice as fast. The two remaining lines represent computers 4 and 8 times as fast. Here, an increase in computational power speeds the completion of solvable problems but does little to extend the threshold of problems solvable within a fixed time limit.

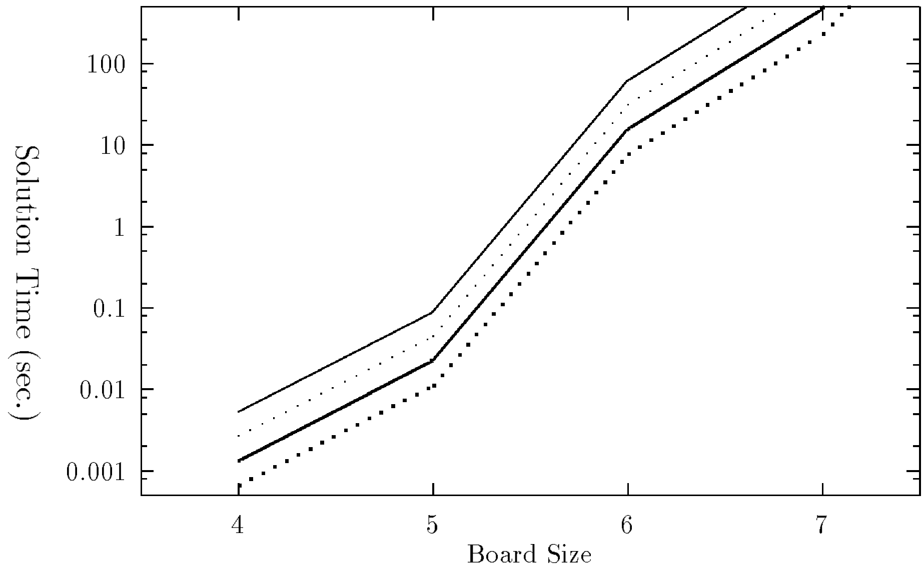


Figure 1.5: Influence of computation resources on N-queens solution time. Each line plots the expected solution time of N-queens problems on computers of varying speed.

1.4 Search Reduction

Unfortunately, theoretical results suggest that, for many problems of practical importance, there are no efficient solutions [HU79, GJ79]. It seems quite likely that all techniques for solving these problems must either fail to solve some problem instances or resort to exploring an exponentially large search space.

Of course, just because a system may have to resort to enumeration of candidate solutions in some cases does not mean that nothing can be done to avoid unnecessary search in other cases. In general, knowledge about the domain of interest can be used to refine the search procedure so that it does not have to rely on a completely naive generate-and-test approach to finding a solution.

Even the casual reader will notice abundant opportunities for avoiding search in Figure 1.2. Most obviously, some identical boards occur more than once even in this abbreviated search tree. These duplicates result from placing queens in the same board positions in different orders. Of course, the order in which queens are placed has no bearing on whether or not a board configuration is an N-queens solution. A search procedure that detected identical board configurations and generated each configuration only once would encounter only $1,820$ nodes in the 4-queens search space. Another obvious search refinement specific to N-queens is that, once two

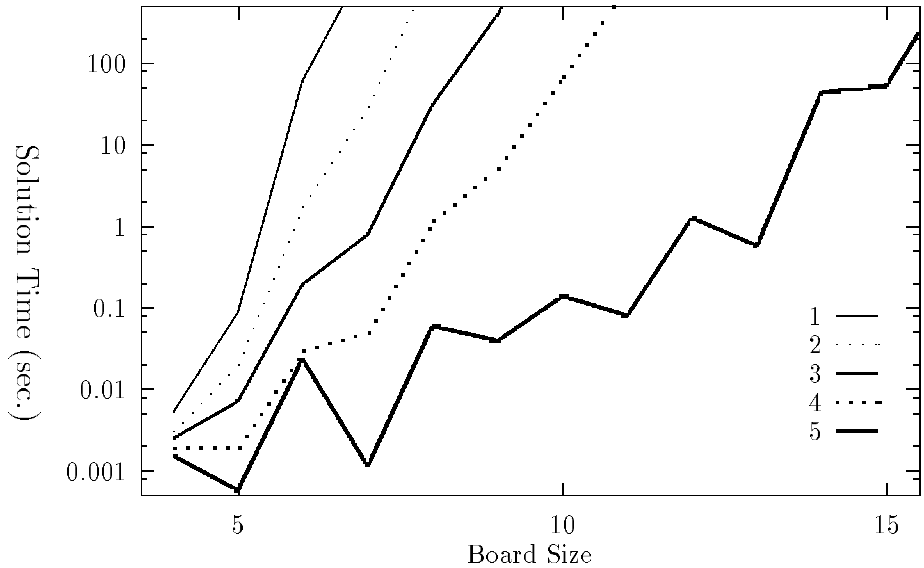


Figure 1.6: Influence of search procedure on N-queens solution time. Lines 1 through 5 plot the performance of search procedures of varying sophistication on N-queens problems of different sizes.

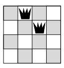
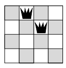
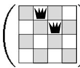
queens threaten each other, no placing of additional queens can eliminate this threat. Node  for example already violates the conditions of the N-queens problem and would not constitute a solution no matter how the remaining queens were placed. Thus, not only may the node  be skipped during the search, but the entire subtree T () may be omitted. If the search procedure is refined so that it never generates configurations in which inter-queen threats already exist, then the 4-queens search space is reduced to just 17 nodes.

Figure 1.6 demonstrates the reduction in search that results from these simple refinements. This figure charts the performance of five systems, each adding a search refinement to its predecessor:

1 As in Figure 1.2, the search procedure checks for threats only after all queens are placed and may generate duplicate board configurations.

2 Each board configuration is generated only once in the search tree.

3 Boards with two queens on the same row are never generated.

4 Boards with two queens in the same column are never generated.

5 Boards with two queens along the same diagonal are never generated.

Here, even the most sophisticated system has not done away with search completely, but clearly there are cases in which the search space can be dramatically curtailed. The search spaces for all systems appear to grow at least exponentially, but N-queens problems are solved much more quickly with search refinements in place than without them. In fact, search reduction schemes such as these can facilitate a multiplicative increase in the size of problems solvable with fixed computational resources.

The performance improvements achieved through changes to the search procedure differ qualitatively from those achieved through changes to the resource bound. This is evident in the difference between Figures 1.5 and 1.6. Increases in computation resources are attractive because they have a somewhat uniform effect across all problem instances. For example, Figure 1.5 shows that doubling computing power can cut solution time in half for all problems. Unfortunately, it is not as effective at actually increasing the set of solvable problems. Changing the search procedure may have potential to appreciably increase the set of solvable problems, but its influence across all problem instances is less uniform. In Figure 1.6, the effects of search refinement are most pronounced on the larger problems.

1.5 Search Parallelism

Alternatively, effective solution time may be reduced by permitting a number of processors to work together to solve the problem. All that is required is a means of dividing the search among available processors. Again, Figure 1.2 demonstrates abundant opportunities for such a division. In what is perhaps the most obvious scheme, each search subtree could be assigned to a separate process. Less obvious divisions of the problem might involve policies like making each processor responsible for the placement of a single queen.

Although parallelism is not typically intended to reduce the total amount of computation required, it can facilitate faster solution by bringing to bear greater computational power. Unlike increased speed in a single processor, computational power gained through parallelism may have nonuniform utility across a set of problem instances. This can be particularly apparent in the context of a refined serial search. In the most naive N-queens search, for example, subtrees rooted at the same level are guaranteed to have the same size. However, this is no longer true when the N-queens search refinements listed above are in place. The relative sizes of these subtrees would have nontrivial influence on the effectiveness of some of the more obvious mechanisms for parallelizing search.

1.6 Nagging

My research focuses on this nonuniform influence changes to the search procedure can have on the size and shape of the search space. This nonuniformity is vaguely apparent in Figure 1.6, but a more carefully designed experiment will make it even more evident.

I introduce a variant of the N-queens problem called the M:N-queens problem. The solution criteria for $M:N$-queens is the same as N-queens, but the starting board configuration features some number $M < N$ of queens already placed. These problems are not too different from standard N-queens problems. In fact, they resemble the roots of subtrees of an N-queens search tree. To illustrate the nonuniform influence of search-reduction mechanisms, I compare the solution of an M:N-queens problem to the solution of its *transpose*. If B is a starting board configuration for M:N-queens then the transpose of B has a queen it row i and column j if and only if there is a queen at row j and column i of B.

Figure 1.7 plots the results of solving 100 randomly generated $M:N$-queens problems[1] on system 3 of Figure 1.6. The system was required to report a solution if one existed and to report that the problem was unsolvable otherwise. Each point represents an individual problem. The point's X coordinate is the solution time for the randomly generated problem and the Y coordinate is the solution time for its transpose. The diagonal line in Figure 1.7 identifies problems that are solved exactly as quickly as their transpose. Observe that, while points tend to congregate around the diagonal, most deviate from it by some small margin.

Figure 1.8 plots the results of solving 100 randomly generated $M:N$-queens problems[2] on system 5 of Figure 1.6. As before, each point compares the solution time for an individual problem with its transpose. Here, the deviation from the diagonal is much more pronounced. In fact, some problems are solved more two orders of magnitude faster than their transpose. A five-minute time limit was imposed on search for each problem. The points indicated by the symbol $\circ$ represent problems that were solved in only one of their forms within this bound.

[1]This suite of M:N-queens problems features a fixed N value of 10 and M values ranging from 25 to 50 percent of N. Each of these problems contained no threats but did not necessarily permit threat-free placements for the remaining queens.

[2]Because this system is generally more effective at solving N-queens problems than the system used in Figure 1.7, larger M:N-queens problems were generated here. This suite features a fixed N value of 40 and M values ranging from 25 to 50 percent of N.

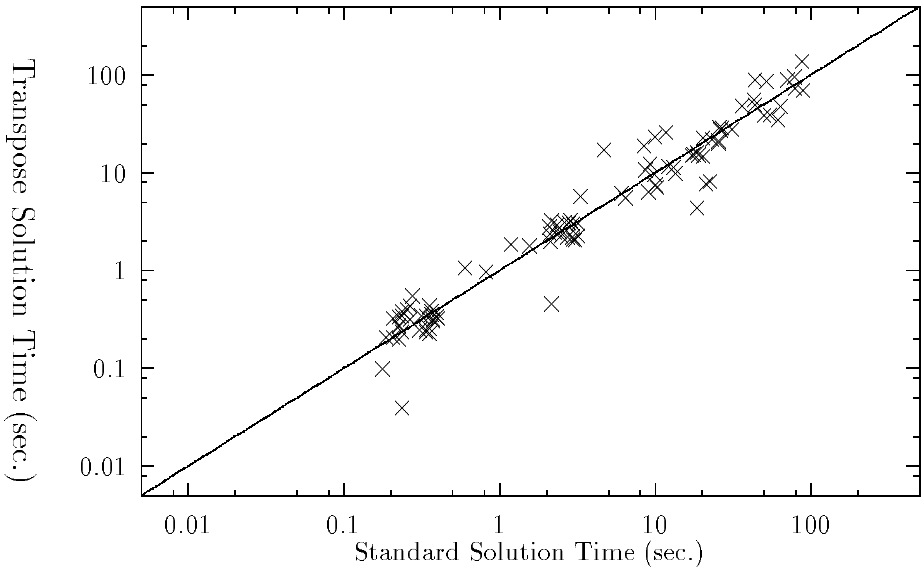


Figure 1.7: Naive search performance on randomly generated M:N-queens problems and their transpose instances.

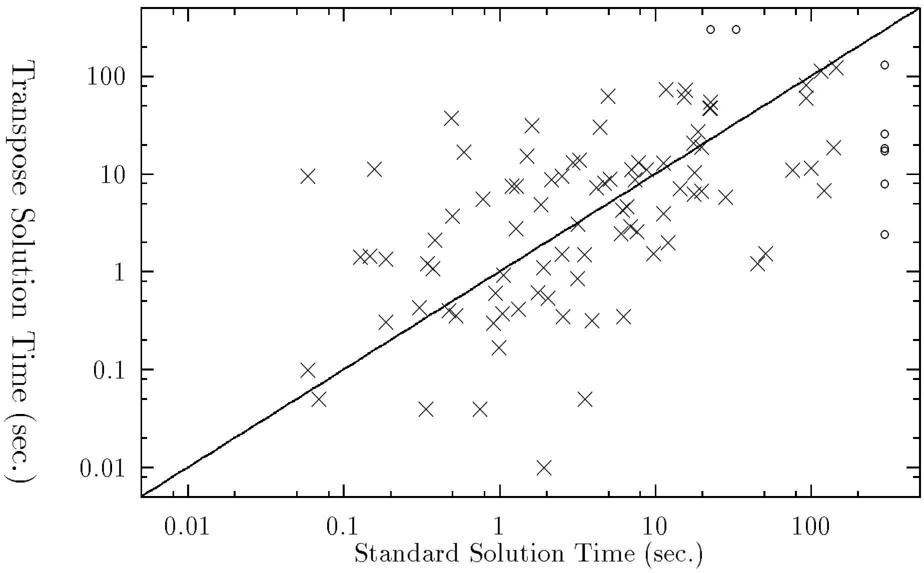


Figure 1.8: Sophisticated search performance on randomly generated M:N-queens problems and their transpose instances.

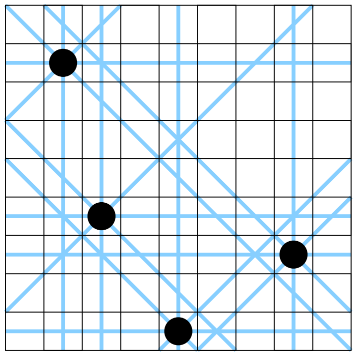


Figure 1.9: In this pathological $M:N$-queens example there is no solution. The placement of only four queens threatens every space in the leftmost column but leaves four spaces in the top row open. Working from top to bottom, even the most refined search procedure expands 8 nodes before determining that there is no solution. However, the unsatisfiability of the transpose is immediately evident.

This deviation from the diagonal is a bit surprising as there is no fundamental difference in difficulty between a M:N-queens problem and its transpose. Both have exactly the same number of solutions and transposition even facilitates a mapping between these solutions. Figure 1.9 illustrates a simple M:N-queens case for which the transpose is much easier to solve. The differences in solution time evident in Figure 1.8 are a reflection of bias in the search procedure. What is perhaps more interesting is that this bias does not appear to be a result of naivete in the search procedure. To the contrary, Figures 1.7 and 1.8 suggest that refinements to the search procedure may introduce a greater bias.

Figure 1.8 illustrates the degree to which a particular search procedure can be sensitive to details of a problem's encoding. Here a fairly benign change in representation can have a great impact on how difficult it is to solve. Dually, a small change in the search procedure can have a very nonuniform influence on the ease with which a particular problem instance is solved.

Nagging is an adversarial, distributed, search reduction technique that is designed to exploit this sensitivity to details of representation or search procedure. Under nagging, parallel processes compete to explore regions of the same search tree using different problem representations or different search procedures. If one such process can determine that a region of the search space contains no solutions, other processes can avoid exploring it.

The simple search reduction techniques given for N-queens in Section 1.4 are effective because they use some fairly inexpensive checks of the board configuration to avoid exploring exponentially large regions of the search space. Essentially, they trade efficient for inefficient computation when it is possible. Nagging tries to avoid exploring an exponentially large region of the search space by completing an alternative search problem instead. Although this alternative problem is still an exponentially large search tree, Figure 1.8 suggests that it may be significantly smaller than the original. If this is the case, the total search may be reduced.

Chapter 2

Nagging

Nagging is a search reduction technique by which multiple processes may cooperate to solve large problems. In addition to its potential for search reduction, nagging is well suited to a distributed computing environment. This chapter describes nagging in the context of tree search and formalizes some of its interesting properties.

2.1 Nagging Framework

Nagging relies on a function f that maps from a subtree of the search space to an alternative search tree:

Definition 2.1 *For search tree T and node $n \in T$, $f(n, T) = T'$ where T' is an alternative search tree such that T' contains solutions whenever $T(n)$ contains solutions. The class of all such functions f is denoted $\mathcal{F}$.*

Functions in $\mathcal{F}$ are called *problem transformation functions*. The intuition behind them is that, given a problem in some domain of interest, they produce a new, "simpler" problem. The new problem is simpler in that it has a solution whenever the original does.

In the N-queens domain, for example, the board transposition of Chapter 1 might serve as a problem transformation. Although this operation was introduced as a means of generating alternative $M{:}N$-queens problems, the analogy between $M{:}N$-queens and states of the N-queens search make obvious its applicability to subtrees in N-queens. A more interesting form of N-queens transformation might actually relax the solution conditions. Given a board B with some queens already placed, the transformed problem could require that queens be placed only on the unoccupied, odd-numbered rows. This operation is a member of $\mathcal{F}$ since completing an N-queens solutions requires that queens be placed on all unoccupied rows. If

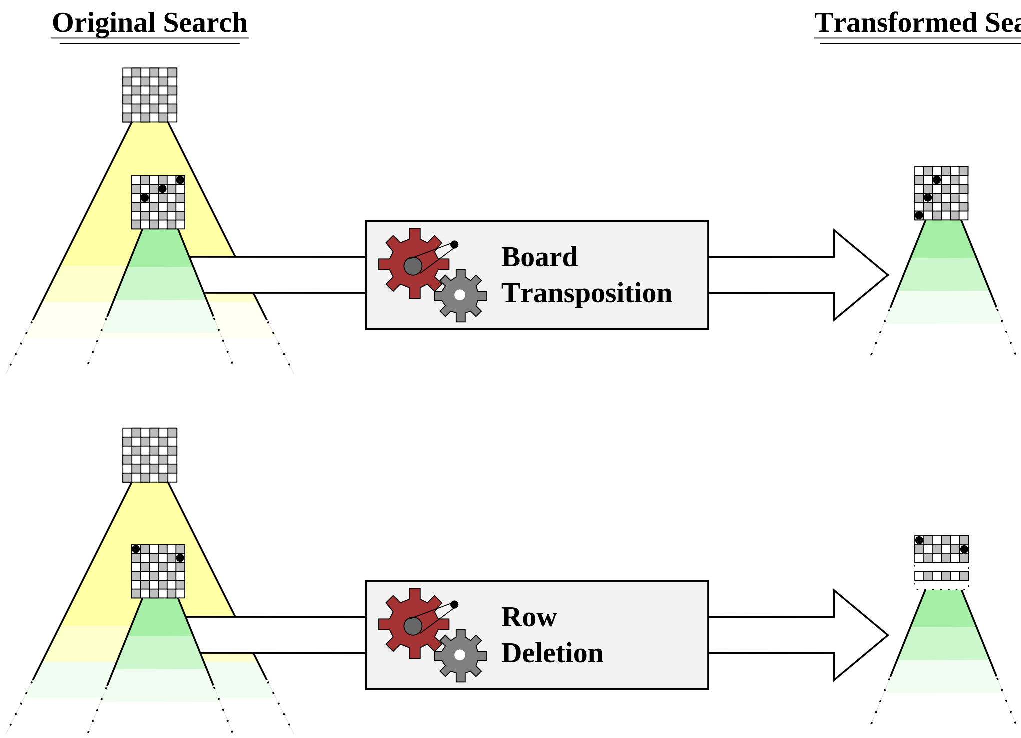


Figure 2.1: Examples of problem transformation in the N-queens domain. The *row deletion* transformation introduces solutions that are not present in the original search tree, while *board transposition* does not.

queens can be safely placed on all rows, they can certainly be safely placed on only the odd-numbered ones. However, this transformation differs from board transposition in that it may introduce solutions in the transformed problem that do not exist in the original.

Problem transformation functions are useful because they can provide information about the location of solutions in T without requiring that T be exhaustively explored. This is captured in Theorem 2.1.

Theorem 2.1 *For search tree T, node $n \in T$ and $f \in \mathcal{F}$, if $f(n,T)$ contains no solutions, then $T(n)$ also contains no solutions.*

Proof: *This is simply the contrapositive of the definition of $\mathcal{F}$. Since the presence of a solution in $T(n)$ implies that there is a solution in $f(n,T)$, the absence of a solution in $f(n,T)$ guarantees that $T(n)$ also contains no solutions.* □

Theorem 2.1 indicates that it may be possible to avoid exploring some $T(n)$ by searching in $f(n,T)$ instead. If $f(n,T)$ is explored under a complete search procedure and no solutions are found, then $T(n)$ must also contain no solutions, and exploring $T(n)$ would be futile. This is clearly true of the transformations for N-queens given above.

Nagging exploits precisely this property of the problem transformation function. 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. When a nagging process is idle, it consults the master process. The master selects some node $n \in T$ such that n has been expanded but $T(n)$ has not yet been completely explored. The nagger then explores $f(n,T)$ for some $f \in \mathcal{F}$. If the nagger exhaustively explores its $f(n,T)$ without finding a solution, Theorem 2.1 guarantees that $T(n)$ contains no solutions and continuing to explore it would be futile. Thus $T(n)$ can be pruned from the master's search without risk of missing a solution. If, instead, the nagging process finds a solution in $f(n,T)$, no pruning of the master's search is warranted, but as soon as the nagger finds a solution it again becomes idle and is free to explore a new transformed subproblem.

Nagging is named colloquially to reflect the annoyance the nagging processes represent to the master. While the master attempts to solve its search problem directly, under its given formulation, naggers examine simpler problems of partial relevance. However, a nagger must always be ready to intervene when its efforts indicate that the master's current approach is futile.

Figure 2.2 gives an idealized view how nagging can reduce search. Assume that $T(n_b)$ contains no solutions while n_c is a solution. A depth-first search procedure

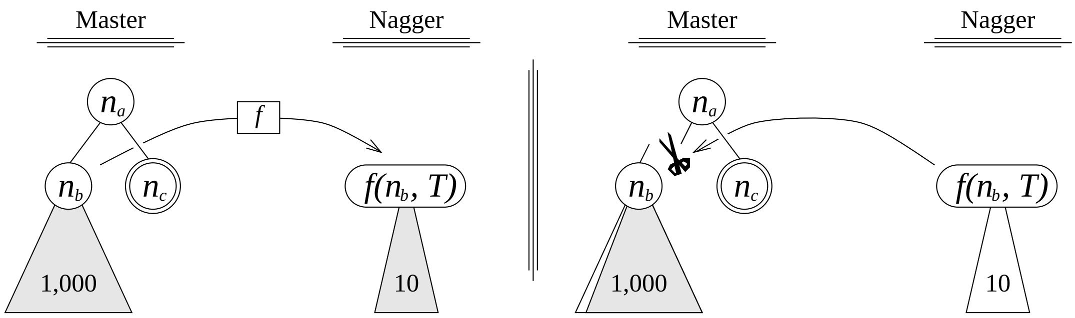


Figure 2.2: Model of search pruning under nagging. In this case, exploring a 10-node, transformed subtree permits the pruning of a 1000-node subtree from the original search space.

would have to completely explore the large subtree $T(n_b)$ before noticing the solution n_c. In Figure 2.2, the transformed subtree $f(n_b, T)$, like the original, contains no solutions. Unlike $T(n_b)$, however, $f(n_b, T)$ contains only a few nodes. Consequently, exploring $f(n_b, T)$ may require a small amount of computation compared to $T(n_b)$. If a nagger elects to explore $f(n_b, T)$ it may be able to quickly determine that it contains no solutions and prune the master's search before much computational effort has been wasted on $T(n_b)$.

2.2 Nagging Protocol

In its simplest form, nagging requires only three types of messages. Each message is exchanged between the master and a single nagging process.

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

problem The *problem* message is the mechanism by which the master distributes work to available nagging processes. When the master receives an idle message from a nagger, it may assign the nagger a new transformed search problem. The master selects some $n \in T$ such that n has been expanded but $T(n)$ has not yet been completely explored. The master then transmits a problem message to a nagging process specifying some $f(n, T)$. When a nagger receives a problem message, it begins exploring the prescribed search tree under some complete search procedure. The master process continues exploring T according to its serial search procedure.

prune If a nagging process exhausts its assigned search problem $f(n, T)$ without finding any solutions it reports this to the master in the form of a *prune* message. When the master receives a prune message, it knows that further exploration of $T(n)$ would be futile. After sending a prune message, the nagger becomes idle, and, after receiving a prune message, the master is licensed to assign the nagger a new subproblem.

The prune message is the means by which the master's search is actually reduced. The treatment of this message is a bit more subtle than the other two. The master transmits a problem message containing $f(n, T)$ only after exploration of $T(n)$ has begun. The master may also continue to explore $T(n)$ while a nagger explores $f(n, T)$. As a result, when a prune message for $T(n)$ arrives, some portion of $T(n)$ will have already been explored. In the extreme case, the master may even finish its search of $T(n)$ before a relevant prune message arrives.

In general, a message to prune $T(n)$ does not prevent the subtree from being explored; it simply permits the master to discard the unexplored portions of $T(n)$. When the master receives a message to prune $T(n)$, it deletes any nodes in $T(n)$ from the fringe and continues with its usual search procedure. Prior to the receipt of the prune message, the master's search is unaffected. As demonstrated by Theorem 2.2, this policy captures the intuition that, once the master process receives a prune message, it no longer needs to waste time exploring the pruned subtree. The truncation of the fringe is the only direct effect the prune message has on the master's search.

Theorem 2.2 *During a search of T, if $n \in T$ has already been expanded, then deleting all nodes in $T(n)$ from the fringe prevents further exploration of $T(n)$.*

Proof Sketch: *Proof is in two parts. For any node $n' \in T$ there can never be more than one representative $p^j(n')$ on the fringe at any point in the search. Removing the nodes in $T(n)$ from the fringe eliminates this single representative for each $n' \in T(n)$. If neither n' nor any of its ancestors are on the fringe, then n' cannot subsequently be expanded during the search*[1].

2.3 Transformation Function Properties

Membership of a transformation functions in $\mathcal{F}$ is sufficient to justify pruning of the master's search, but it is not generally adequate to realize this pruning in practice. Thus, although they are not required for membership in $\mathcal{F}$, transformation

[1]Complete proofs may be found in the Appendix A.

functions will normally be expected to satisfy some additional properties that foster opportunities to actually prune the master's search. Consider function f_1.

$$f_1(n, T) = T_1 \text{ where the root of } T_1 \text{ is a solution node}$$

Although $f_1 \in \mathcal{F}$, nagging with f_1 is incapable of reducing the master's search. Theorem 2.1 permits pruning only when $f(n, T)$ contains no solutions. Since $f_1(n, T)$ always contains a trivial solution, it can never be used to justify the transmission of a prune message. To exclude functions like f_1, transformation functions will normally be expected to be *informative*. Viable transformation functions must have the potential to generate subtrees void of solutions.

Definition 2.2 (Informative) *A problem transformation function $f \in \mathcal{F}$ is* informative *if there exist search tree T and $n \in T$ such that $f(n, T)$ contains no solutions.*

Similar lacunae in the definition of $\mathcal{F}$ are revealed by f_2.

$$f_2(n, T) = T(n)$$

As demonstrated in Figure 2.2, nagging is built around the assumption that it is sometimes more effective to explore a transformed version of the search tree than it is to explore the original tree. Function f_2 is the identity problem transformation, so exploring the search trees it generates is never any easier than exploring the original. Those functions that are capable of reducing the size of a search tree are called *reductive*.

Definition 2.3 (Reductive) *A problem transformation function $f \in \mathcal{F}$ is* reductive *if there exist search tree T and $n \in T$ such that $|f(n, T)| < |T(n)|$.*

This property is not strictly necessary for search reduction. If the master employs something other than a depth-first search policy, then even the identity transformation function can facilitate search pruning. Figure 2.3 illustrates how this may occur. Under breadth-first search, the master explores both $T(n_b)$ and $T(n_c)$ concurrently. The nagger, however, concentrates all of its effort on exploring $T(n_b)$ and can be expected to finish its search problem before the master. If $T(n_b)$ contains no solutions, any portion of $T(n_b)$ that the master does not explore before the nagger finishes will be discarded by the ensuing prune message. This type of search reduction is achievable under the nagging protocol but is inconsistent with the intent of nagging[2]. Thus, although both properties are not necessary for search reduction,

[2]In general, this type of search pruning is more appropriately classified as OR parallelism, which is described in Chapter 7.

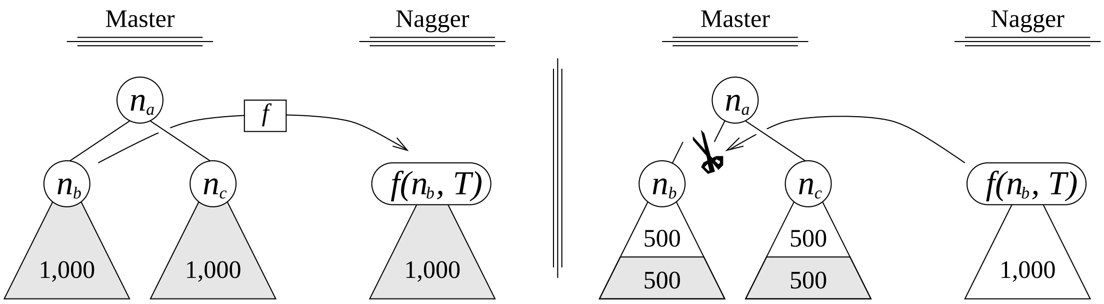


Figure 2.3: Search reduction with trivial transformation function. If the master's search procedure is not depth-first, then nagging can reduce search even when the nagger's search space is larger.

transformation functions are typically expected to be informative and reductive. In fact, a viable transformation function should share a witness to the informative and reductive properties. A reduction in search space size must accompany the absence of solutions to be useful.

A special case of the reductive property is of particular interest and is necessary for search reduction even when the general property is not. For any $f \in \mathcal{F}$, successful nagging requires that some $f(n, T)$ be exhaustively explored. This presupposes that $f(n, T)$ is finite. Thus, even for infinite search trees, a viable transformation function must, at least occasionally, generate alternative search spaces that are finite. Chapter 3 demonstrates that, in practice, this condition may not be problematic.

The final property prohibits functions that work too hard. The function f_3 gives perfect information about the location of solutions, and it gives it in a very concise form.

$f_3(n, T) = T(n')$ where n' is a solution if and only if $T(n)$ contains solutions and where $c(n') = \emptyset$.

Although f_3 would be ideal for search pruning, all known means of computing $f_3(n, T)$ would be analogous to exploring $T(n)$. Thus, it might be just as easy to explore $T(n)$ as it would to perform the transformation. For this reason, transformation functions are usually expected to be efficiently computable; $f(n, T)$ should be computable in time that is polynomial in the size of the problem description or the depth of n.

These properties capture much of the intuition behind nagging, and the need for them is reflected in Figure 2.2. They identify situations in which a readily available, alternative problem is better than the original. Nagging does not require that every computed transformation exhibit both the informative and reductive properties. If the $M:N$-queens results of Chapter 1 are representative, nagging processes will encounter search spaces that are both smaller and larger than the original by orders of magnitude. In effect, the price paid for the chance to reduce search is that transformation will sometimes increase it or introduce illegitimate solutions. As long as some naggers occasionally receive transformed problems that are better than the original, the master's search can be reduced.

2.4 Nagging Properties

Although nagging is conceptually quite simple, it exhibits properties that make it attractive as both a general-purpose search-pruning technique and a parallelism scheme. These properties may be broadly interpreted as promoting two design objectives: similarity to the underlying serial search procedure and suitability for a distributed computing environment.

2.5 Influence on Search Procedure

In the development of nagging, a principal goal was to minimize negative impact on the operation of the master process. Since the master is the only process charged with actually exploring T and finding solutions, it is desirable that this process be permitted to perform its search with minimal interference. Also, since all nagging processes must communicate directly with the master, an interest in scalability makes it advantageous to keep the overhead on the master low. Nagging enjoys several properties that are consistent with these goals. As illustrated by the nagging protocol given in Section 2.2, all communication is nagger initiated. The master only needs to respond to idle messages and truncate its fringe when prune messages arrive. The master does not need to constantly attend to opportunities to distribute the search; it only needs to dispatch appropriate subproblems when naggers request them. Likewise, naggers are free to concentrate on performing search rather than on coordinating with other processes. Once a nagger receives a transformed subtree of the search, it is permitted to explore that tree without interruption. Thus, a nagging implementation may entail few changes to the underlying serial search procedure and may preserve many of its favorable performance characteristics.

Nagging also preserves some qualities of the serial search order. The following condition on search procedures enables a precise specification of how nagging may influence the order and speed with which the master finds solutions.

Definition 2.4 (Myopic Search Procedure) *Let T be a search tree and F the*

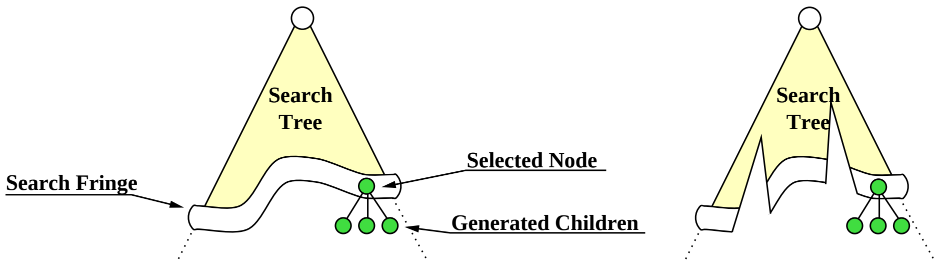


Figure 2.4: When choosing a node from the search fringe for expansion, a myopic search procedure will always select the same node even if other nodes are missing from the search tree. Expanding this node produce the same set of children even if nodes elsewhere have been pruned.

set of nodes on the fringe at some point during a search of T. Let node n be the member of F that is selected from F as next for expansion and let $F' \subseteq F$ be a search fringe reached during an exploration of a different tree T' such that $n \in F'$. A search procedure is said to be myopic *if the following three conditions are met for any T, F, F' and n.*

- *Node n must also be selected as the next node to be expanded from F'.*
- *If C is the set of children generated when n is selected from F and expanded, then C must also be generated when n is selected from F' and expanded.*

The myopic property is so named because it excludes a variety of techniques by which information gained while searching in one part of T can be used to prune or reorder search elsewhere in T. These techniques include various search reduction mechanisms like intelligent backtracking [Bru78,KL87]. Some learning schemes such as caching and lemmaizing [AS92,SS93] also compromise myopia although "offline" techniques like Explanation Based Learning [MKKC86,DM86,SE94] typically do not. When myopia is preserved, nagging exerts a well understood influence on the search order.

Theorem 2.3 (Solution Ordering) *If the master's search procedure is myopic, the master will find all solutions with nagging that it would without nagging, and it will find them in the same order.*

Theorem 2.3 is easily proven given the following lemma:

Lemma 2.4 *For any search tree T and any serial and nagged exploration of T under a myopic search procedure, there exists a nondecreasing function $g : \mathcal{N} \rightarrow \mathcal{N}$ that satisfies the following conditions:*

1. *Let F be the set of nodes on the fringe after i nodes are expanded under the serial search. If F' is the set of nodes on the fringe after the master expands $g(i)$ nodes under the nagged search then $F' \subseteq F$.*
2. *If node n is expanded both by the serial search and by the master under the nagged search, then $\exists i \in \mathcal{N}$ such that n is the i^{th} node expanded in the serial search and the $g(i)^{th}$ node expanded under the nagged search.*
3. *$g(i) \leq i$.*

Proof Sketch: *The function g is constructed by induction on the sequence of node expansions. Both the serial and nagging searches begin with the root node. If node n is expanded under both serial and nagging searches, it generates the same set of children in both and the the nagging fringe remains a subset of the non-nagging fringe. The same is true if n is expanded in only the serial search.*

Proof Sketch for Theorem 2.3: *First, it is shown that any solution found in the serial search must also be found in the nagging search. The definition of $\mathcal{F}$ guarantees that nagging cannot remove a solution or a solution's ancestor from the fringe. Lemma 2.4 implies that any solution or ancestor thereof that is expanded in the serial search must must also be expanded by the master in a nagged search. Lemma 2.4 is then used to show that the ordering of these solutions is identical in both searches.*

The solutions found under nagging are not guaranteed to match those found without it because the nagged search may discover solutions that the serial search omits. Clearly, this can only occur when the underlying serial search procedure is incomplete. Figure 2.5 demonstrates how this may happen. A depth-first search would become trapped in the infinite subtree $T(n_b)$. If transformation function f is reductive, $f(n_b, T)$ may be finite even though $T(n_b)$ is not. If $T(n_b)$ contains no solutions, then exploring $f(n_b, T)$ could permit it to be pruned, thus freeing the master's depth-first search to advance to the solution n_c.

This example shows that, in special cases, the potential search reduction afforded by nagging is unbounded. More generally, it can be shown that a nagging process will not cause the master to explore more of the search tree than it would without nagging. Theorem 2.5 makes this precise.

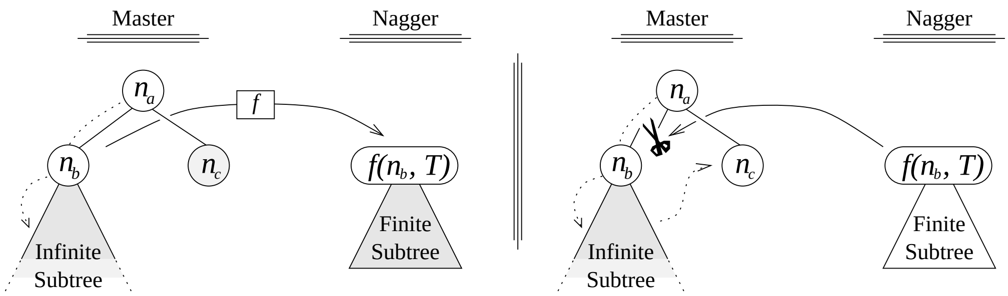


Figure 2.5: Ordinary depth-first search is incomplete when the search tree is infinite. If a problem transformation can produce finite search spaces given infinite ones, nagging can free the master from infinite subtrees and restore search completeness.

Theorem 2.5 (Non-Increasing Search) *As long as the master's search procedure is myopic, then, for any solution n, if n is the i^{th} node expanded without nagging then n will be found within j node expansions with nagging where $j \leq i$.*

Proof Sketch: *Theorem 2.3 guarantees that n will also be found in the nagged search. If n is reached in both the serial and parallel searches, Lemma 2.4 guarantees that n will be found at least as early in the nagged search.*

If performance is measured purely as the number of nodes expanded, then nagging may improve performance, but it will never hurt it. In practice, reduction in solution time is a more pertinent measure of performance than node expansion alone. Nagging is not guaranteed to reduce solution time since augmenting a search implementation with a nagging component may adversely impact the node expansion rate and, as a result, increase the search time even while reducing the number of nodes expanded. However, as argued previously, nagging is designed around a policy of maintaining low overhead on all processes involved. A sufficiently great reduction in search should translate into a reduction in solution time.

Naturally, one of the most vital requirements of nagging is that it must preserve the soundness and completeness of the underlying search procedure. Soundness is usually not problematic since any solutions generated under nagging must be recognized by the master's search procedure. Completeness is less obvious. Theorem 2.3 serves as a proof that completeness is maintained for myopic search procedures. It would be desirable to show that nagging retains completeness for arbitrary search procedures, but such a proof is not possible; a counterexample is deferred until Chapter 5. A weaker result which will is nearly as useful is shown here instead.

Results addressing the impact of nagging on the exploration of an arbitrary search tree under an arbitrary, complete search procedure are elusive. Completeness is a constraint on the solutions discovered by a search procedure, not on the operation of that search procedure. Exploration of T under nagging may differ not only from the serial search of T but also from the serial search of any tree. Thus, constraints on the output of a serial search on individual trees are insufficient to justify claims about its behavior under nagging. The following is an operational constraint that, although neither necessary nor sufficient for completeness, is typically satisfied by all but the most contrived search procedures.

Definition 2.5 (Punctilious Search Procedure) *A search procedure is said to be* punctilious *if, for any n and any $n' \in c(n)$, the presence of solutions in $T(n')$ implies that n' is generated whenever n is expanded.*

A punctilious search procedure will never discard paths to a solution regardless of the search history. In particular, it will not throw away solutions simply because nagging tampers with the search fringe. This condition is so fundamental to the design of nagging that it will usually be assumed without comment. In particular, it qualifies the following useful property:

Theorem 2.6 (Completeness for Finite T) *Given a finite search tree T and a punctilious search procedure that discovers all solutions in T, a nagged search based on the same search procedure will also discover all solutions in T.*

Proof Sketch: *Proof is by contradiction. For solution n, consider the deepest ancestor of n that is generated in the nagging search. Once placed on the fringe, this node cannot be removed by a prune message. Since T is finite, it must eventually be selected from the fringe and expanded. This expansion must generate n or a deeper ancestor of n.*

2.6 Distributed Computing

Nagging was designed with a loosely-coupled network of computers as the target architecture. The healthy abundance of this type of computation resource was a prime reason for its selection as was its realistic potential for scalability. Nagging exhibits many properties that are consistent with a distributed model of parallel computation.

Among these is a low demand for inter-process communication. Under the nagging protocol, communication is only necessary when a nagging process completes its assigned search problem and, therefore, may be fairly infrequent. Additionally, the size of communicated messages may be kept reasonably low. The idle message may be quite terse, requiring only an indication of which nagging process has become idle. If the master retains an record of the transformed subtree being explored by each nagger, then the prune message also requires only an indication of which nagger produced the message.

The problem assignment message is a bit more complicated since it must convey an entire search tree to the nagging process. A concise representation of this message is possible if the nagger, rather than the master, is made responsible for evaluating the problem transformation function. To generate its own transformed subproblem, the nagger must have access to both T and some $n \in T$. For any given problem, the same search tree T is an argument to all applications of the transformation function. If the naggers are charged with performing the transformation, then the problem description may be transmitted to all nagging processes at the start of search. Nagging processes may use this problem description to construct parts of T as needed. To compute each $f(n, T)$, a nagger also needs an appropriate node $n \in T$. Transmission of this n can be simplified by the nagger's access to T; rather than communicating n directly, it can be described in terms of its location in T. Any node is uniquely determined by its path from the root. Each non-root node can be given an index among its siblings. As indicated in Figure 2.6, the master can name n by transmitting the indices of each node on this path from the root to n. The nagger may then reconstruct n by expanding nodes along this path. Exchanging work by reconstructing the search state rather than copying it can permit large search problems to be transmitted with comparatively small messages. This approach has been taken in other search parallelization techniques to reduce communication overhead [AL91,SLK90,Clo87,CA88]. This policy is also consistent with an interest in maintaining low overhead on the master process. When generating a problem message, the master must select an appropriate node from T but does not have to perform the potentially expensive computation of f.

Finally, the nagging protocol can be seen as partially asynchronous. When a nagger is idle, it may have to wait for a problem message from the master, but the master is never required to wait for messages. This partial asynchronicity fosters fault-tolerant properties that make nagging well suited to a distributed computing environment. Given a distributed computing model in which messages may be lost or delayed, but not altered, Theorem 2.7 formalizes these properties.

Theorem 2.7 (Fault Tolerance) *Theorems 2.3 and 2.6 apply even if messages under the nagging protocol are lost or delayed.*

Proof Sketch: *From the master's point of view, lost messages of any type ultimately result in missing prune messages. Theorems 2.3 and 2.6 only require that*

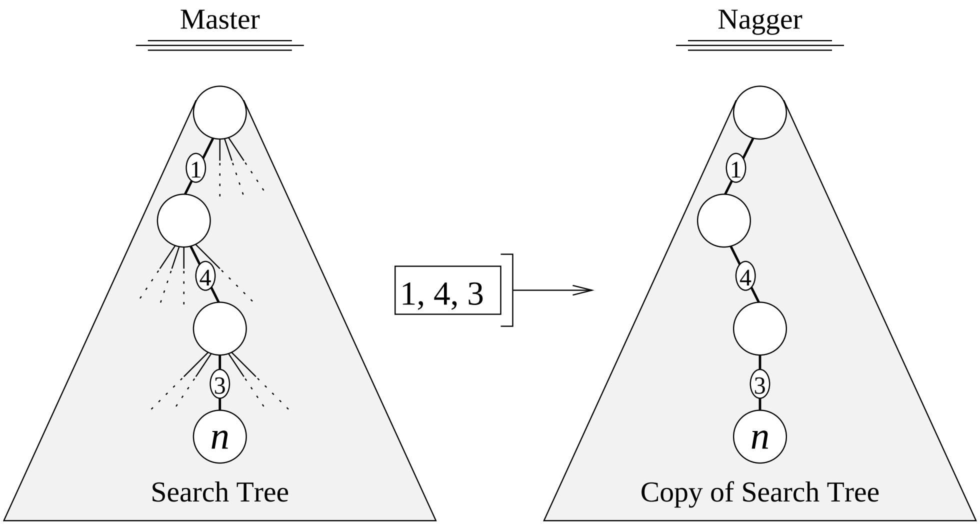


Figure 2.6: Communication of n by its path in T

prune messages received by the master meet the preconditions of the nagging protocol. Omission of a prune message does not violate the protocol or jeopardize completeness.

Theorem 2.7 implies that nagging can accommodate arbitrary delay or loss of messages and can even tolerate the quiet failure of a nagging process. Such failures may cause the master process to explore portions of T that it would not otherwise explore, but it will not cause the master to fail, overlook solutions or generate invalid ones.

Failures of a more malevolent nature are a bit more problematic. If naggers generate invalid prune messages, the master may be compelled to skip regions of the search that contain solutions. An obvious remedy would be to require the master to recreate the nagger's transformed subproblem and verify its results whenever a prune message was received. Any naggers that began to produce erroneous prune messages would be discarded. This would still offer the potential for search reduction in the presence of reliable communication as long as transformation was sufficiently effective at reducing search. It would, however, compromise the non-increasing search result of Theorem 2.5.

Chapter 3

First-Order Inference

Nagging has been implemented as part of the *Distributed, Adaptive Logical Inference* (DALI) theorem prover. DALI is a search engine built around the model-elimination inference procedure [Lov78]. More generally, DALI is intended to be an architecture for combining a variety of search reduction techniques. It features a number serial performance enhancements in addition to its parallel component.

The first-order search problems addressed by DALI represent only a single formal problem, but one with considerable expressive power. A large number of problems in a variety of domains have natural encodings in first-order logic. By applying nagging directly to a general first-order inference engine, this large class of problems is made accessible by virtue of their first-order encodings.

3.1 Language

First-order logic is a general-purpose formalism for expressing facts about some domain of interest and justifying conclusions based on those facts. A thorough introduction to the subject as it relates to search problems may be found in several introductory texts [RK91,Shi92,Win92]. Model elimination operates on first-order formulae in *conjunctive normal form* (CNF). The relevant notation and definitions are outlined here to disambiguate terminology and to facilitate subsequent discussion.

First-order formulae in CNF are composed of various syntactic objects and are defined inductively as follows:

constant A *constant* corresponds to some particular element in the domain of interest such as a chair or the number three. The symbols a, b and c are used as typical constant symbols throughout this text.

variable *Variables* are symbols used to stand for an arbitrary elements in the domain. They permit general statements about the domain. The Prolog convention of identifying variables with initial capitals is used here, with X, Y and Z being the most common variable names.

function *Functions* describe mappings between elements of the domain. The application of a function represents some element in the domain and is written $f(t_1, \ldots t_n)$ where f is a function symbol and each t_i is a constant, a variable or another function application. The terms $t_1 \ldots t_n$ are called the *arguments* of the function and the number n is called its *arity*. Constant symbol can be considered zero-arity functions. The symbols f, g and h are used in this text as generic function symbols.

term Constants, variables and function applications are collectively identified as *terms*.

atom *Atoms* capture relationships between elements of the domain and are written as $q(t_1, \ldots t_n)$ where each t_i is some term. In $q(t_1, \ldots t_n)$, the symbol q is called a *predicate* and denotes a particular relation. Atoms can be considered either true or false. The atom $q(t_1, \ldots t_n)$ is true if $\langle t_1, \ldots t_n \rangle$ is in the relation q and false otherwise. The symbols q and r are used as anonymous predicates throughout this text as is p when there is little risk of confusion with the parent relation on tree nodes.

literal A *literal* is either an atom (*e.g.,* $q(t_1, \ldots t_n)$) or the negation of an atom (*e.g.,* $\neg q(t_1, \ldots t_n)$). The negated atom $\neg q(t_1, \ldots t_n)$ is true exactly when $q(t_1, \ldots t_n)$ is false. A negated literal is called a *negative literal*, while a literal without negation is called a *positive literal.* To limit notational tedium, it is tacitly assumed that $\neg\neg q(t_1, \ldots t_n) = q(t_1, \ldots t_n)$. The symbol l will frequently be used to stand for unnamed literals. For any literal l, l and $\neg l$ are said to be *complementary.*

clause A *clause* is a finite set of literals. Logically, a clause is the disjunction of its elements and is true if at least one of its members is true. Variables within a clause are assumed to be universally quantified. Of particular interest are some restricted forms of clauses including *definite clauses*, which contain exactly one positive literal, *Horn clauses*, which have at most one positive literal and *unit clauses*, which contain only one literal. Anonymous clauses will be denoted by the letter C.

theory A *theory*, sometimes called a *domain theory* or a *program*, is a finite set of clauses. Logically the theory is the conjunction of its members and is true when none of its clauses are false. The symbol S is used to stand for the set of clauses constituting the theory.

Although this syntax is among the more restrictive characterizations of first-order logic, it is sufficiently expressive to admit any statement that can be made in the more embellished forms of the language.

Common theorem proving concepts such as *substitution* and *unification* are defined as usual [CL73,NS93]:

Definition 3.1 (Ground) *A term or first-order formula is said to be* ground *if it contains no variables.*

Definition 3.2 (Substitution) *A* substitution *is a set of term-variable pairs, written* $\{t_1/v_1, \ldots t_k/v_k\}$*, where every* v_i *is a different variable and every* t_i *is a term that does not contain* v_i*. If* $\theta = \{t_1/v_1, \ldots t_k/v_k\}$ *is a substitution and* t *is some term, then the application of* θ *to* t*, written* $t\ \theta$*, is the result of simultaneously replacing each occurrence of some* v_i *in* t *with its corresponding* t_i*. The new term* $t\ \theta$ *is said to be an* instance *of* t*. Substitutions can be applied to literals, clauses and the like with similar effect. The symbol* θ *is used to represent a substitution.*

Definition 3.3 (Substitution Composition) *The composition of substitutions* θ_1 *and* θ_2*, denoted by* $\theta_1 \circ \theta_2$*, is the function that is equivalent to first applying* θ_1 *and then applying* θ_2 *to any formula.*

Definition 3.4 (Syntactic Variants) *Terms* t_1 *and* t_2 *will be called* syntactic variants *if they are instances of each other.*

Definition 3.5 (Unification) *Terms* t_1 *and* t_2 *are said to* unify *if there exists a substitution* θ*, called a* unifier*, such that* $t_1\ \theta$ *and* $t_2\ \theta$ *are identical. If* t_1 *and* t_2 *unify, then there exists a* most general unifier θ_{mgu} *such that* $t_1\ \theta_{mgu} = t_2\ \theta_{mgu}$ *and, for any other unifier* θ*, there exists some substitution* θ' *such that* $\theta = \theta_{mgu} \circ \theta'$*. For any unifiable, first-order terms, there exists an efficiently computable most general unifier. Consequently, although most general unifiers are not unique, they will be treated as if they are the unique output of one such deterministic algorithm.*

3.2 Proof Calculus

Model elimination is a first-order inference procedure that, although not properly a resolution procedure, is closely related to resolution and its variants [Lov72]. Like resolution procedures, model elimination performs proof by refutation. To show that a given statement is true, it adds the negation of the statement to the theory S and attempts to show that the augmented S is now contradictory. Model elimination differs from resolution in that it operates on proof objects called *chains* rather than on clauses. It also does not require factoring [1] and retains completeness in the presence of the *input restriction.* To achieve completeness, resolution proof procedures must permit derived clauses to be resolved with other derived clauses. Under model elimination's input restriction it is never necessary to combine two derived chains. Instead, all inference operations combine a clause from S with either another clause from S or with some derived chain. This restricted focus in the available inference steps is largely responsible for the high inference rates enjoyed by model elimination implementations [Sti89].

3.2.1 Connection Tableaux

Each model elimination chain has a natural mapping onto a tree [LSBB94]. This correspondence is captured in *connection tableaux.* Although model elimination chains and their associated inference operations are easily defined without reference to this mapping, connection tableaux make explicit some of the structure inherent in the chain. This structure simplifies the description of model elimination variants and will be useful in the discussion of relevant problem transformation functions needed by nagging.

A connection tableau $\Delta = \langle \tau, \mu \rangle$ consists of a finite tree τ along with a function μ defined on the nodes of τ. The function μ labels each non-root node of τ with a literal. Although tableaux are not, themselves, search trees, much of the notation introduced in Chapter 1 for search trees can be applied to the tree component of a tableau. However, the symbol T, when in isolation, will be reserved as a referent for the entire search tree. The symbols Δ and τ will play similarly exclusive roles for connection tableaux.

The labeling μ of a tableau is required to satisfy two conditions. For each non-root node $n \in \tau$, either $c(n) = \emptyset$ or $\{\mu(n') \mid n' \in c(n)\} = C\ \theta$ for some clause $C \in S$ and some substitution θ. Model elimination requires, additionally, that for any non-root, non-leaf $n \in \tau$ there exists some $n' \in c(n)$ such that $\mu(n')$ and $\mu(n)$ are complementary.

[1] In resolution theorem proving, factoring is a process by which matching literals within a clause are unified. For completeness, resolution procedures must factor the clauses they derive [Shi92]. Model elimination's freedom from factoring is a convenience when implementing an efficient search procedure [Sti88].

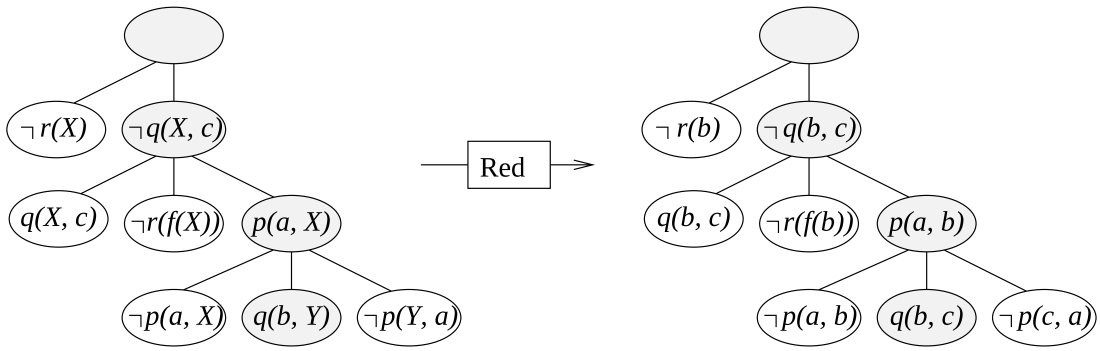


Figure 3.1: Example of the reduction operation. Shading indicates the target, reduced branch.

Definition 3.6 *A* branch *of tableau τ is the sequence of nodes on some simple path $n_1, \ldots n_j$ in τ where n_1 is the root of τ and n_j is a leaf. It is convenient to exploit the one-to-one correspondence between branches and leaves in the tableau. The notation $leaf(\beta)$ indicates the member of β that has the greatest depth.*

Definition 3.7 *A branch in a connection tableau is considered* closed *if it contains nodes n_a and n_b such that $\mu(n_a)$ and $\mu(n_b)$ are complementary. A subtree $T(n), n \in \tau$ is closed if all branches containing n are closed, and a tableau is closed if all its branches are closed. A tableau, subtree or branch is considered* open *whenever it is not closed.*

3.2.2 Inference Operations

The fundamental inference operations of model elimination are *extension* and *reduction*. These are easily defined in terms of the connection tableau. Let $\Delta = \langle \tau, \mu \rangle$ be an open tableau featuring β as an open branch. Reduction may be seen as the act of closing β and is defined as follows:

Definition 3.8 (Reduction) *Given tableau Δ with branch β and node $n \in \beta$ such that $\mu(n)$ and $\neg\mu(leaf(\beta))$ are unifiable, $Red_{\beta,n}\Delta$ is defined as a new tableau $\Delta' = \langle \tau, \mu' \rangle$. The new tableau Δ' differs from Δ only in the labeling of its nodes. With θ representing the most general unifier of $\mu(n)$ and $\neg\mu(leaf(\beta))$, $\mu'(n) = \mu(n)\,\theta$ for all $n \in \tau$.*

Observe that reduction is only applicable when there exists a branch containing complementary, unifiable literals. The reduced branch β is rendered closed in the resulting tableau $Red_{\beta,n}\Delta$, since the modified labeling function makes n and $leaf(\beta)$ complementary. Figure 3.1 demonstrates the operation of reduction.

While reduction closes branches in the tableau, extension creates new, potentially open ones. The definition of extension is a bit more complicated. Reduction is sensitive to the state of the tableau only, but extension brings clauses of S into play. To prevent the naming of variables in Δ and S from having inappropriate, coincidental effects that might obfuscate the assumed universal quantification of clauses in S, variables of the tableau are renamed as part of extension [Bun83].

Definition 3.9 (Standardized Apart) *Let Δ be a tableau and S the set of theory clauses. Tableau Δ is* standardized apart *from S if it contains none of the variables occurring in S.*

Given Δ, S and an infinite supply of variable names, it is always possible to construct a syntactic variant Δ' of Δ that is standardized apart from S. Although there are an infinite number of qualifying tableaux Δ', when a version of Δ is needed that is standardized apart from S, it is assumed to be unique.

Definition 3.10 (Extension) *Let clause $\{l_1, \dots l_k\} = C \in S$ and literal $l \in C$ such that $\mu(leaf(\beta))$ and $\neg l$ are unifiable. Let θ be the most general unifier of $\mu(leaf(\beta))$ and $\neg l$; when $leaf(\beta)$ is the unlabeled root of Δ, unifiability is not required and θ is taken as the empty substitution. Extension first constructs a new tableau $\langle \tau', \mu' \rangle$ based on Δ. The tree τ' is identical to τ except for the addition of nodes $n_1, \dots n_k$ as children of $leaf(\beta)$. 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 $i = 1, \dots k$. Finally, the extended tableau, denoted by $Ext_{\beta,C,l}\Delta$, is a syntactic variant of $\langle \tau', \mu' \rangle$ standardized apart from S.*

Extension replaces a single open branch β by one or more new branches. Extension is closely related to resolution within Prolog. However, while Prolog permits resolution only on the single positive literal of each clause, extension may operate on any literal within the clause [Sti89]. After extension, at least one of the newly created branches is guaranteed to be closed. The substitution applied in $Ext_{\beta,C,l}\Delta$ renders l and $\mu(leaf(\beta))$ complementary. For this reason, accounts of model elimination typically omit the complementary child from $c(leaf(\beta))$ after extension. Figure 3.2 demonstrates the operation of extension.

The tree structure of the tableau changes as extensions are applied. Branches that exist in one tableau may no longer be legitimate branches after some number of inference operations. To simplify discussion of a set of nodes β across related tableaux, some of the branch-related notation has been defined so that it is meaningful even after extension replaces one branch with a number of longer branches. The notation $leaf(\beta)$, for example, is meaningful even when β contains no leaf nodes.

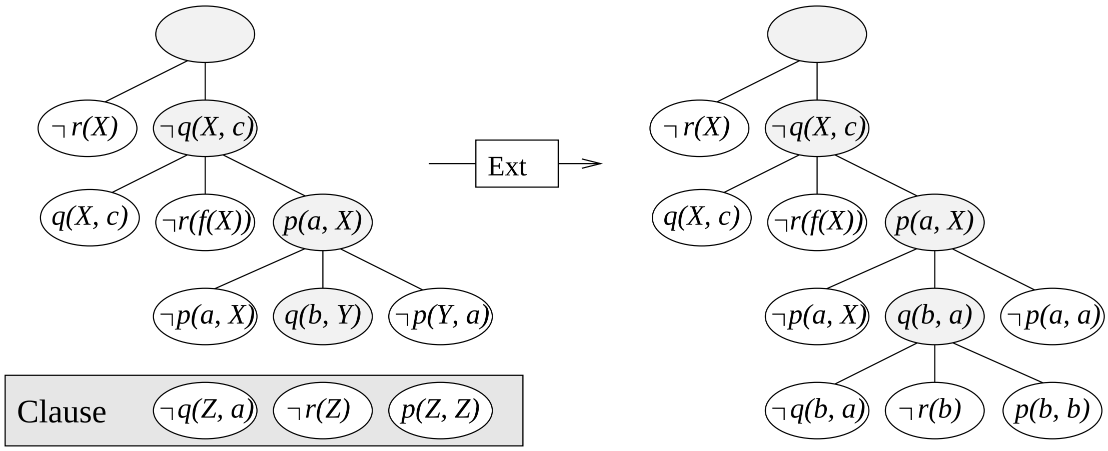


Figure 3.2: Example of the extension operation. Shading indicates the target, extended branch.

The status of β as closed or open can be generalized similarly; when not a branch itself, β is considered closed if $T\,(\mathit{leaf}(\beta))$ is closed.

The inference operators $Red_{\beta,n}$ and $Ext_{\beta,C,l}$ are the building blocks of model elimination proofs. The following definitions describe the construction and notation of such proofs:

Definition 3.11 (Composition of Operators) *Let op_1 and op_2 stand two for model elimination inference operations, and let Δ be a tableau. The notation $op_2(op_1\Delta)$ represents the tableau that results when op_2 is applied to tableau $op_1\Delta$. Inference operators are assumed to be right-associative, so parentheses will usually be omitted.*

Definition 3.12 (Empty Tableau) *The* empty tableau *consists of a single, unlabeled root node and is denoted here by $\Delta_\emptyset$.*

Definition 3.13 (Partial Proof) *Given a theory S, a* partial proof *for tableau Δ is a sequence of reduction and extension operations $\langle op_1, \ldots op_k \rangle$ such that $op_k \ldots op_1\Delta$ is defined.*

Definition 3.14 (Proof) *Given theory S, a model elimination* proof *is a partial proof $\langle op_1, \ldots op_k \rangle$ for $\Delta_\emptyset$ such that $op_k \ldots op_1\Delta_\emptyset$ is closed.*

Definition 3.15 (Subproof) *A* subproof *of branch β in tableau Δ is a partial proof $\langle op_1, \ldots op_k \rangle$ for Δ such that β is closed in $op_k \ldots op_1\Delta$ and each op_i operates on some tableau branch β' for which $\beta \subseteq \beta'$.*

3.3 Search Space

Construction of a model elimination proof features many sources of nondeterminism. For any open tableau there may be several reduction and extension operations that are applicable. The nondeterministic components reflect the parameters of the $Red_{\beta,n}$ and $Ext_{\beta,C,l}$ notation.

- It may be necessary to choose between reduction and extension if both are applicable.
- Both reduction and extension operate on an open branch of the tableau. An open tableau may feature more than one open branch.
- In reducing branch β there may be more than one node $n \in \beta$ that can be made complementary to $leaf(\beta)$.
- In extending branch β there may be more than one clause $C \in S$ and literal $l \in C$ that can be made complementary to $leaf(\beta)$.

Because of these sources of nondeterminism, discovering a model elimination proof usually requires some form of search. The necessary choices prescribe a model elimination search tree. Each node of the search tree corresponds to an entire tableau, with $\Delta_\emptyset$ as the root. The children of each tableau Δ are the tableaux $op\Delta$ for all applicable inference operations op. Solution nodes are the closed tableaux.

Trees constructed in this manner are consistent with the search tree model employed in Chapters 1 and 2. Since each of the parameters specifying an extension or reduction represents a choice among a finite number of alternatives, each node in the model elimination search tree has a finite number of children. Of course, some theories may still entail search trees containing an infinite number of nodes. As a result, determining if a model elimination proofs exists is, in general, an undecidable problem. When a solution exists, a sufficiently persistent search procedure will eventually find it. When there are no solutions, a similarly persistent search procedure may have to perpetually explore an infinite search space.

One particularly powerful refinement to this search tree involves the choice of an open branch for each operation. The choice among available open branches has no bearing on the presence of solutions in T. If there is a proof for S, then there is a proof in which the open branch for each inference operation is selected by any arbitrary rule. Thus, for any node Δ in the model elimination search tree, $c(\Delta)$ only needs to include tableaux derived by extending and reducing a single open branch. From each tableau the branch by which all children are derived is chosen by a deterministic *selection function*, g. For open tableau Δ, $g(\Delta)$ is some open branch in Δ. As will be shown later, the particular function g used to construct the model elimination search tree may have profound influence on the structure of the tree but will not change the number of solutions it contains.

3.4 Search Mechanism

DALI's search for a proof may be naturally interpreted as an exploration of the model elimination search tree. The following sections detail the search procedure DALI employs, the ordering it imposes on the search tree and the efficient implementation mechanisms it uses to take advantage of various properties of model elimination.

3.4.1 Iterative Deepening

DALI explores the search tree through a depth-first search procedure. This permits an exponentially large search space to be explored with only a polynomial amount of storage, but, since it is not complete for infinite trees, simple depth-first search is inappropriate for arbitrary first-order theories. To cope with potentially infinite trees, DALI uses *depth-first iterative deepening* [Kor85]. Under iterative deepening, a potentially infinite search tree T is explored by searching successively larger, finite portions of T. The exploration of each of these finite search trees is called an *iteration*, and, in the case of DALI, is carried out through depth-first search. The price paid for the favorable storage requirement is that each iteration must duplicate some of the search done in previous iterations. However, this repeated search at each iteration induces only a constant-factor overhead in the number of nodes explored [ST85].

The portion of T explored at each iteration is delimited by a bound on some search depth metric. Any nodes in T that exceed the current bound are ignored. At each iteration, the bound is increased by at least some constant, and a new search is initiated. In this way, any node of T will eventually be visited. Model elimination offers a number of obvious features for measuring and bounding search depth. Many approaches favor a limit on the number of inferences [Sti88]. This measure is analogous to a limit on the depth of a node in T, permitting an additional level of T to be explored with each iteration. It also meets the essential requirement that each iteration present a finite search space. An alternative depth measure, called *A-literal depth* according to the nomenclature of model-elimination chains, is based instead on the structure of the tableau [FLSY74,AS92]. A-literal depth bounds the height of tableaux explored at an iteration.

The choice of depth measure may have profound effect on the ease with which solutions are found. The depth measure dictates the shape of the finite portion of T explored in each iteration. Some solutions will be found at an early iteration under one depth measure but not reached until a much later iteration under a different measure. Some efforts have sought to avoid the disadvantages of particular depth measures by employing a weighted combination of several [LMG92]. Although DALI can employ a variety of depth measures, it is designed around A-literal depth. The principal reason for this bias is that the A-literal measure avoids some unfortunate interactions between the depth bound and some of DALI's search reduction mechanisms.

3.4.2 Search Ordering

At each choice point it is only necessary to consider inference operations pertaining to a single branch. Although DALI was designed to be flexible in the rule it uses to select this branch, Prolog's policy of choosing the leftmost open branch [Shi92] is standard. DALI assumes a left-to-right ordering on the children of each node in the tableau. If some n_1 is to the left of n_2 then $T(n_1)$ is closed before open branches in $T(n_2)$ are selected. As in Prolog, the ordering on the tableau is inherited from the syntactic specification of the theory. When branch β is extended by some clause written $l_1 \vee \ldots \vee l_k$, the added children of $leaf(\beta)$ are ordered from left to right as $n_1 \ldots n_k$.

When some search node $\Delta \in T$ has more than one child, DALI's depth-first search order means that the subtrees rooted at each child are explored one after another. Again, DALI offers some flexibility in the order in which these children are considered. By default, children resulting from reduction are visited first. Competing reduction operations are attempted according to the ordering of a node's ancestors; for tableau branch β, $Red_{\beta,p^k(leaf(\beta))}$ is considered before $Red_{\beta,p^{k+1}(leaf(\beta))}$. Once all applicable reductions have been tried, extensions are considered, with extensions by unit clauses given precedence [Sti88]. Finally, extensions by non-unit clauses are attempted. As in Prolog, competing extension operations are tried according to the ordering of the clauses in the syntactic specification of S. Among unit and non-unit clauses, those listed earlier in the theory are considered first.

3.4.3 The Warren Abstract Machine

Like many other theorem provers built around model elimination, DALI makes use of the *Warren Abstract Machine* (WAM), an efficient implementation technology developed within the Prolog community [War83,AK91,AL91,ABCM88]. The WAM shares Prolog's orientation toward definite-clause programs and is, therefore, not directly applicable to unrestricted first-order clauses. Model elimination, however, is operationally very similar to Prolog and necessitates relatively few modifications to the WAM [Sti88].

Among the architectural advantages of the WAM is the ability to explore the search space through incremental modification of various data structures. Although proof tableaux are finite, they are often inconveniently large. Creating several such objects with every node expansion would entail a substantial amount of computation. Fortunately, tableaux change in fairly specific ways with each inference. The WAM maintains a representation of only a single tableau, which it modifies as it traverses T. The WAM is built around a depth-first search discipline, which is conducive to incremental maintenance of this tableau representation. Depth-first search explores T one subtree at a time; once exploration of some $T(n)$ begins, all nodes in $T(n)$ are explored before nodes outside $T(n)$ are visited. Whenever a nondeterministic choice is encountered, the WAM creates a structure called a *choice point frame* in which it records information about the current tableau and which of its children have been investigated. When the search enters a subtree $T(n)$ the tableau for n is generated by applying the appropriate inference operation to the tableau for $p(n)$. If the WAM completes the exploration of $T(n)$ without finding a solution, *backtracking* is used to restore the tableau at $p(n)$ according to the record kept at the relevant choice point frame. Some unvisited sibling of n is then identified and its tableau, like that of n, is constructed as a variant of $p(n)$.

This policy necessitates efficient mechanisms for applying inference operations upon entering a subtree and for restoring the state of the tableau during backtracking. Various components of the WAM contribute to making modification of the tableau efficient in both directions. Since the child's tableau differs from its parent's by only a single inference, the corresponding tableaux differ in few respects. The only possible modification to a tableau's tree component is the addition of children to an extended branch. The labeling function of the tableau may be changed whenever an inference operation applies a substitution. While nodes added through extension represent local changes, Figures 3.2 and 3.1 demonstrate that applying a substitution can have a more broad effect.

When the application of a substitution replaces a variable X with something other than itself, it is said to *bind* X. The effect of an inference on the labeling function consists entirely of variable bindings. In fact, as Figures 3.2 and 3.1 demonstrate, seemingly global changes to μ may represent the binding of only a few variables. The WAM limits the computational overhead of binding multiple occurrences of a single variable by representing each distinct variable by a single record.

All occurrences of the variable must refer to this canonical record. Whenever the value of variable X is needed, a process called *dereferencing* is used to follow these references to a variable's actual value. If X is bound, only a single structure must be changed in order for all occurrences to reflect the binding.

An unfortunate consequence of this constant-time variable binding scheme is that variable dereferencing is not a constant-time operation. Furthermore, the need to efficiently retract variable bindings upon backtracking precludes the use of path compression during dereferencing [CLR90]. Whenever a variable is bound, the binding is recorded on a data structure called the *trail*. When backtracking requires that the binding be retracted, the trailed information indicates the single variable record that must be restored. If path compression were used to reduce dereferencing overhead, retracting a variable binding might require the restoration of several records.

The WAM further expedites extension through a specialized set of abstract machine instructions. Each type of abstract instruction performs some elementary operation on the WAM's data structures, and, collectively, they implement both the necessary inference operations and the depth-first search strategy. For each clause in the theory, the WAM compiles a sequence of abstract instructions that perform extension by that clause. In Prolog, these instructions implement unification with a clause's lone positive literal and instantiation of the remaining literals accordingly. Since model elimination is not restricted to definite-clause programs, it is necessary to build a sequence of WAM instructions to match on each literal in a clause [Sti88]. Compiling WAM code for every literal in the domain theory is typically worth the computational overhead. Model elimination's accommodation of the input restriction means that each potential extension corresponds to a particular clause and literal and can therefore be performed by a sequence of WAM instructions compiled at the start of search. By pre-compiling abstract machine code for each possible extension, much of the computation that might otherwise be repeated with each extension can be performed only once before search actually begins. The cost of compiling code for a given extension is amortized over every use of that extension.

The demands of model elimination require a few additional modifications to the fundamental WAM architecture. In an effort to maintain a high inference rate, the WAM employs an unsound unification procedure which permits the creation of cyclic terms. Although this is consistent with Prolog's programming model, it is inappropriate for a general-purpose theorem prover. This disadvantage is easily overcome by adding the necessary *occurs check*. When constructing the most general unifier, the occurs check ensures that some variable X is not bound to a term containing X. The computational overhead of the occurs check can be reduced by employing a set of necessary conditions for the generation of a cyclic term [Pla84].

These conditions may be tested for each literal in the theory and the WAM code written for that literal may reflect its need for the occurs check.

The prototypical WAM also supports no analog for the reduction operation since Prolog's definite-clause programs never require it. Unlike extension, the applicable reductions depend on the structure of the tableau and bear no simple relation to local features of the theory. Because of this, each possible reduction cannot be accommodated by a specialized sequence of WAM instructions. Instead, DALI simply uses a single procedure to attempt all possible reductions. This approach does not benefit from the WAM's compilation of the theory as much as extension, but it can be implemented with little programatic machinery other than what the WAM already provides in support of extension.

3.5 Nagging Component

Both the model elimination search space and DALI's mechanism for exploring it are consistent with the framework described in Chapter 1. Exploiting nagging within DALI requires implementation of appropriate problem transformation functions and accommodation of the nagging protocol within the search procedure.

3.5.1 First-Order Problem Transformation

Just as Chapter 2 presented transformation functions that were specific to N-queens, exploiting nagging in DALI requires a transformation function sensitive to the relevant proof calculus. Unlike the N-queens problem, however, a well-designed transformation function for model elimination will be applicable to a large class of search problems by virtue of their encodings in first-order logic.

In the context of model elimination, membership in $\mathcal{F}$ means that the transformed search space $f(n, T)$ contains a solution whenever the original subtree $T(n)$ contains a closed tableau. Model elimination provides a rich framework in which to define such functions. The expressive power of first-order logic is of particular value here. Instead of generating completely new search problems, nagging transformations may be engineered to map from model elimination search trees to other model elimination search trees. When this is the case, both master and nagger may use DALI as the basis for their search procedures. For example, functions f_4 and f_5 enjoy this property. Function f_4 works by modifying the tableau, while f_5 works by modifying the theory S.

$$f_4(\langle \tau, \mu \rangle, T) \; = \; T(\langle \tau, \mu' \rangle) \text{ where } \forall\, n \in \tau, \mu(n) = \mu'(n)\, \theta \text{ for some substitution } \theta.$$

$f_5(\Delta, T) = T'$ where T' is the model elimination search tree rooted at Δ and using theory $S' = S \cup C'$ for some new clause C'.

The function f_4 replaces some of the non-variable terms in the tableau with variables; the nagger is free to bind these variables as needed in attempting to close all branches. The problems generated by f_5 begin with the same tableau as the master but permit the nagger to use an additional clause in attempting to close it.

At each iteration, the master's search is bounded by some depth limit. Ideally, the nagger's search of its transformed tree could be similarly bounded. This would guarantee that the nagger, like the master, is always presented with a finite search tree. DALI's principal measure of search depth is based on the structure of the tableau and may be applied directly to the nagger's model elimination search just as it is to the master's. Even when the trees they generate are bounded in accordance with the master's current depth limit, f_4 and f_5 are members of $\mathcal{F}$. They are also informative. However, proofs of these properties are not interesting because neither f_4 nor f_5 are reductive; that is, if $T(\Delta)$ contains a solution, then it may be possible to find a solution to the transformed problem more quickly, but the transformed search space will always be at least as large as the original search space. Under DALI's depth-first search procedure, the master could always be expected to finish exploring a subtree before nagger was able to prune it.

These functions demonstrate the tendency for membership in $\mathcal{F}$ to conflict with the reductive property. The definition of $\mathcal{F}$ may be seen as a requirement that solution nodes be preserved across problem transformation. Reductive functions, however, must eliminate nodes through the transformation. Since the locations of solutions cannot be known in advance, transformations designed to preserve all solutions parsimoniously may tend to preserve all non-solutions as well. Both f_4 and f_5 exhibit this problem. For any node $n \in T$, there is an injection from the nodes of $T(n)$ to the nodes of $f_4(n, T)$ or $f_5(n, T)$. Thus, any transformed search problem contains a virtual copy of the original.

Functions f_6 and f_7, defined below, avoid this problem; each node in the transformed search space can be seen as representing a set of nodes in the original.

$f_6(\Delta, T) = T(\Delta')$ where Δ' omits the leaf of the leftmost open branch in Δ.

This transformation effectively eliminates the tableau's leftmost open branch. Under DALI's search procedure, this is the branch that the master would ordinarily attempt to close next. Thus, f_6 permits the nagger to skip a branch and move on to closing the remaining ones.

The definition of f_7 is slightly more involved. Assume that c_1 and c_2 are two distinct constants appearing in the theory S.

$f_7(\Delta, T) = T'$ where T' is the model elimination search tree defined by a new theory S' and rooted at Δ'. The new tableau and theory are identical to the original except that all occurrences of c_1 in S and Δ are replaced by c_2 in S' and Δ'.

This function takes two symbols that are distinct in the master's problem and makes them indistinguishable in the nagger's problem. In this respect, the effects of f_7 may be compared to the notion of abstraction in abstract interpretation [CC77,Mel86]. Search in the nagging process can be understood as a form of abstract interpretation of the master's theory.

Functions f_6 and f_7 are examples of two general approaches to problem transformation. These two approaches form the basis of nagging for first-order model elimination as implemented in DALI. The first class of transformations, called the *permutation class* and exemplified in f_6, encompasses various policies for abbreviating the tableau and perturbing the branch selection policy. Functions in the second class, called the *abstraction class*, assign each nagging process an abstracted version of the master's problem.

3.5.1.1 Permutation Class

In the context of nagging, the class of permutation functions is defined as follows:

Definition 3.16 (Permutation Class) *Function f is a* permutation transformation *if $f(\Delta, T) = T'$ where T' is a model elimination search tree rooted at tableau Δ'. Tableau Δ' is identical to Δ except for the possible deletion of the leaves of some open branches in Δ. Search tree T' is also permitted to use a different branch selection function than that used in $T(\Delta)$. The set of all such functions f is written as $\mathcal{P}$.*

Figure 3.3 demonstrates the effects of a typical transformation in $\mathcal{P}$. Closed branches are indicated by shading. Deleting some leaf nodes will permit closing of the tableau without closing the truncated branches. Changes in the branch selection order are captured here by reordering children in the tableau.

As with f_4 and f_5, a function in $\mathcal{P}$ is a member of $\mathcal{F}$ even if the trees it generates are bounded by the same depth limit in effect on the master process. Theorem 3.1 shows the containment of $\mathcal{P}$ in $\mathcal{F}$. Membership in $\mathcal{P}$ does not, however, guarantee that a function is informative or reductive; in fact, functions f_1 and f_2 given in chapter 2 both satisfy the definition of $\mathcal{P}$.

Theorem 3.1 $\mathcal{P} \subseteq \mathcal{F}$

The proof of this result is aided by two definitions:

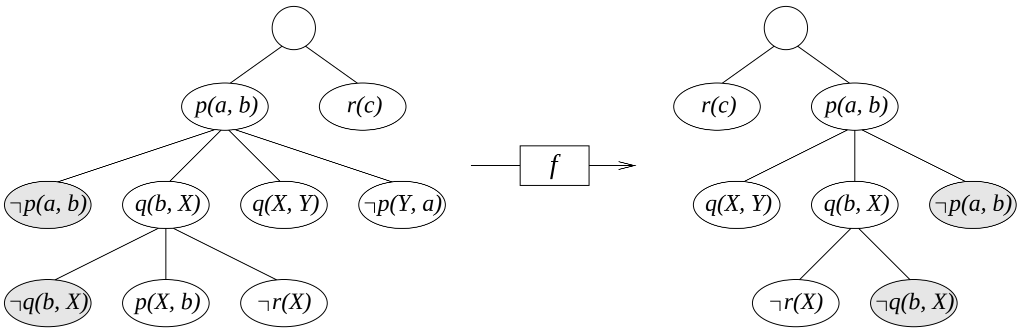


Figure 3.3: Example transformation in $\mathcal{P}$. Closed branches are indicated by shading. Changes in the branch selection function are illustrated by reordering each node's children.

Definition 3.17 (Nilpotent Reduction) *A* nilpotent reduction *operates on a branch that is already closed. Operation $Red_{\beta,n}$ is nilpotent for tableau Δ if the labels on n and $leaf(\beta)$ are complementary in Δ.*

Definition 3.18 (Bloated Proof) *A proof or sequence of inference operations is* bloated *for tableau Δ if it contains exactly one extension or reduction for every branch that it closes when applied to Δ.*

Ordinarily, operations on one branch of the tableau may close other branches as a side-effect of the substitutions they apply. Nilpotent reductions are useful because they make explicit the closing of a branch but cause no real changes to the tableau. Thus, for any proof that is not bloated, it is possible to build an equivalent, bloated proof by simply inserting nilpotent reductions.

The proof of Theorem 3.1 follows easily from the following lemma. Lemma 3.2 shows that it is possible to effect local transpositions in a sequence of inference operations without fundamentally changing the resulting tableau:

Lemma 3.2 (Operator Transposition) *Let Δ be some tableau containing two different branches β_1 and β_2. Let op_1 and op_2 be two model elimination inferences that operate on β_1 and β_2 respectively such that $op_2 op_1 \Delta$ is defined. The tableau $op_1 op_2 \Delta$ is defined and is a syntactic variant of $op_2 op_1 \Delta$.*

Proof Sketch: *The substitutions applied under op_1 and op_2 must each be a most general unifier for some pair of literals. Composed, these substitutions constitute a witness to the unifiability of both pairs of literals. This witness can be used to show that $op_1 op_2 \Delta$ is defined and is an instance of $op_2 op_1 \Delta$. A similar argument from $op_1 op_2 \Delta$ to $op_2 op_1 \Delta$ shows that this instantiation is reciprocated.*

Proof Sketch for Theorem 3.1: *Let f be a transformation function in $\mathcal{P}$ and let Δ_s be a closed tableau in subtree $T(\Delta)$. Starting from a bloated sequence of inference operations that derives Δ_s from Δ, it is possible to construct a new, bloated sequence of operations that derives a closed tableau from the root of $f(\Delta, T)$. Each step of this construction involves a local transposition of inference operations permitted under Lemma 3.2.*

The functions in $\mathcal{P}$ exploit two mechanisms for reducing the size of transformed search spaces. The most obvious of these is the deletion of tableau branches. Through these omissions, tableaux that are distinct in the original search space may have a single representative in the transformed search. If two tableaux in the master's search differ only in their respective subproofs for some branch β, elimination of β in the nagger's tableau obscures this distinction. Thus, transformation under $\mathcal{P}$ may realize a many-to-one mapping from states of the original search to states of the new search.

The second mechanism for search reduction concerns the changes to the branch selection policy. The local reordering of inference operations permitted by the Operator Transposition Lemma can be used as a basis for justifying a claim in Section 3.3: using a deterministic branch selection function does not compromise completeness. However, even though the particular branch selection function does not affect whether or not proofs can be found, it may substantially influence how quickly they are found and, more significantly, how quickly their absence can be verified. The Horn-clause theory given in Figure 3.4 demonstrates this. There is no refutation proof for this theory, but the ordering of literals in the final clause may force a naive search to explore more than 10^6 nodes before determining this. For each subproof of the $p(X)$ branch, all possible combinations of subproofs for the $p(Y)$ and $p(Z)$ branches would be considered. Of course, none of these combinations will satisfy the rightmost branch. If the last clause was reordered to $\neg q(X) \vee \neg p(X) \vee \neg p(Y) \vee \neg p(Z)$, then the search space could be exhausted in little more than 100 nodes. Naggers using functions in $\mathcal{P}$ operate under the assumption that the master's default ordering of the search is poor. A nagger's use of an alternative branch selection function represents an attempt to find a better ordering.

3.5.1.2 Abstraction Class

The set of abstraction functions in first-order nagging is denoted by $\mathcal{A}$. Functions in $\mathcal{A}$ associate two or more distinct symbols from the master's theory with a single, new symbol in the nagger's theory. This mapping affects both constant and function symbols.

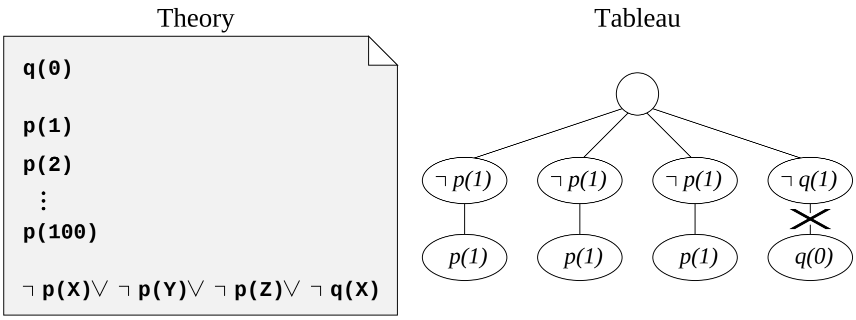


Figure 3.4: Theory demonstrating the influence of order on search space size. This theory has no proof, but its order makes this rather expensive to determine. Reordering the last clause can make the search space much smaller.

With S representing the set of clauses in the master's theory, let $=_{abs}$ be an equivalence relation on the constant and function symbols appearing in S. The relation $=_{abs}$ defines a class of mappings from first-order formulae to new first-order formulae under which the distinctions between symbols equivalent under $=_{abs}$ may be obscured. DALI uses a simple syntactic device to enforce this mapping.

Definition 3.19 (Abstraction Mapping) *Given S and $=_{abs}$ described above, the abstraction mapping g_{abs} associates with each first-order formula a set of abstracted formulae. Given formula F, $g_{abs}(F)$ is a set containing all formulae derivable from F with the following:*

- *Each occurrence of constant c in F is replaced by either $f_{[c]}(c, V_1, \ldots V_n)$ or $f_{[c]}(V_0, V_1, \ldots V_n)$ where n is the maximum arity of symbols in $[c]$, $V_0, \ldots V_n$ are new, unique variables and $f_{[c]}$ is a function symbol specific to the equivalence class of c that does not appear in S.*
- *Each occurrence of function symbol h in F of the form $h(t_1, \ldots t_k)$ is replaced by either $f_{[h]}(h, t_1, \ldots t_k, V_{k+1}, \ldots V_n)$ or $f_{[h]}(V_0, t_1, \ldots t_n, V_{k+1}, \ldots V_n)$ where n is the maximum arity of symbols in $[h]$, V_0 and $V_{k+1}, \ldots V_n$ are new variables and $f_{[h]}$ is a function symbol specific to the equivalence class of h that does not appear in S.*

Definition 3.20 (Abstraction Class) *Let function g_{abs} be an abstraction mapping. Function f is an* abstraction transformation *if $f(\langle \tau, \mu \rangle, T) = T'$ where, T' is the model elimination search tree defined by some modified theory S' rooted at tableau $\langle \tau, \mu' \rangle$. The labeling function μ' is constructed from μ so that for any $n \in \tau$,*

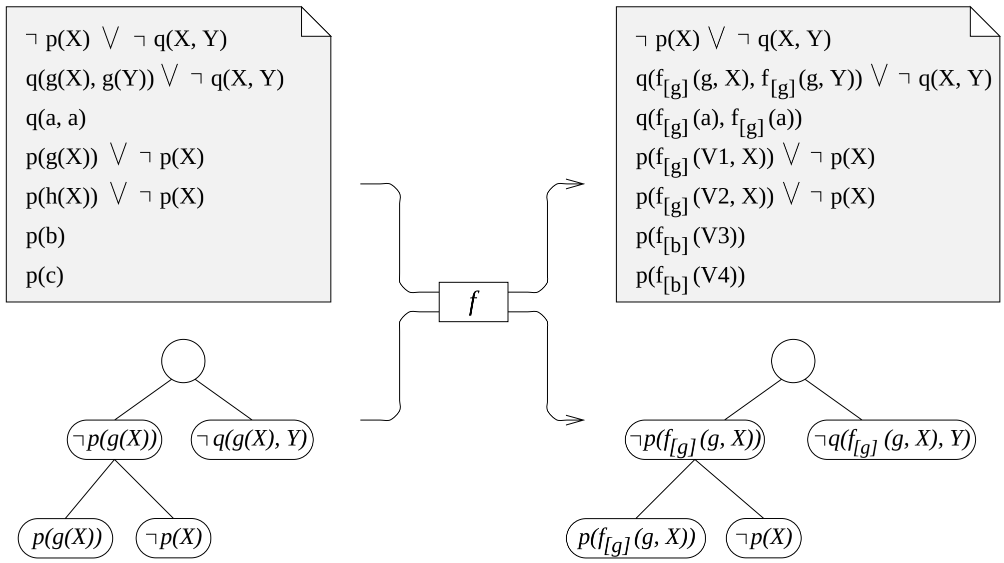


Figure 3.5: Example transformation in $\mathcal{A}$

$\mu'(n) \in g_{abs}(\mu(n))$. *The transformed theory is defined as some* $S' \in g_{abs}(S)$. *Abstracted node labels in* $\langle \tau, \mu' \rangle$ *must be chosen so that any branch closed in* $\langle \tau, \mu \rangle$ *is still closed in* $\langle \tau, \mu' \rangle$. *The set of all such transformation functions* f *is denoted by* $\mathcal{A}$.

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

Figure 3.5 demonstrates the effect of a transformation in $\mathcal{A}$. Here, constants b and c are identified. Replacing each occurrence of these symbols with a term like $f_{[b]}(V)$ removes the distinction between b and c. As a result, the last two clauses in the theory are rendered logically equivalent, and one of them could be discarded without changing the deductive closure. Similarly, association of function symbols g and h renders another clause redundant.

Definition 3.21 indicates when it is safe to discard a clause from the abstracted theory. A clause C may be removed from the abstracted theory as long as some other clause C' is retained, and C' is *as general* as C.

Definition 3.21 (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 $l \in C$, $l = g_C(l)\,\theta$.*

The generality relation on clauses is stronger than conventional subsumption. This more restrictive policy for eliminating logically redundant clauses is necessary for search completeness. Subsumption would be sufficient if the nagger were required only to find proofs in its transformed theory; $f(\Delta, T)$ is, however, rooted at a transformed version of the master's tableau and not at $\Delta_\emptyset$. To ensure that $f(\Delta, T)$ contains solutions whenever $T(\Delta)$ does, it is not always permissible to reduce the theory as much as might be justified by its deductive closure alone. This cautious approach to theory simplification permits the following mapping from elements of the original theory to elements of its abstraction.

Definition 3.22 (Abstraction Trail) *Let f be a problem transformation in $\mathcal{A}$. Let S be a theory and let S' be the abstraction of S under f. The* abstraction trail *for S and S' is a function g_{tr}. For clause $C \in S$ and literal $l \in C$, $g_{tr}(C, l) = \langle C'', l'' \rangle$ for some $C'' \in S'$ and some $l'' \in C''$. Let C' be the abstraction of C generated under f and let l' be the corresponding abstraction of l. Clause C'' is the member of S' which is as general as C' and, with g_C taken as the isomorphism from C' onto C'' identified in Definition 3.21, $l'' = g_C(l')$.*

The abstraction trail is a notational convenience that captures how clauses of the master's theory are represented by clauses in the nagger's smaller, abstracted theory. It is useful in the proof of the following theorem and in subsequent refinements of abstraction nagging.

Of course, it is necessary to show that functions in $\mathcal{A}$ are members of $\mathcal{F}$ even when redundant clauses are discarded wherever generality permits. Theorem 3.3 demonstrates that, like $\mathcal{P}$, $\mathcal{A}$ also satisfies the definition of $\mathcal{F}$ when search trees on either side of the transformation are subject to the same depth bound.

Theorem 3.3 $\mathcal{A} \subseteq \mathcal{F}$

Proof Sketch: *Let f be a transformation function in $\mathcal{A}$. Given a closed tableau Δ_s in subtree $T(\Delta)$, it is possible to construct a representative of Δ_s in $f(\Delta, T)$ that must also be closed. This construction uses the abstraction trail and a notion of* trivial abstraction *to relate tableaux in $T(\Delta)$ to their representatives in $f(\Delta, T)$. The trivial abstraction is simply the member of the abstraction mapping that introduces the fewest new variables. For any $\Delta^j \in T(\Delta)$ the trivial abstraction of Δ^j has the same number of open branches as Δ^j. For any Δ^j, it is possible to find an abstracted tableau $\Delta' \in f(\Delta, T)$ such that Δ' has the same number of open branches* *as the trivial abstraction of* Δ^j. *Since* Δ_s *has no open branches, its representative in* $f(\Delta, T)$ *must also be closed.*

Membership in $\mathcal{A}$ does not guarantee that a transformation is reductive. For example, the trivial abstraction outlined in the proof sketch for Theorem 3.3 is essentially the identity transformation. In less extreme cases, however, abstraction can greatly reduce search. In the context of a backward-chaining inference procedure like model elimination, most of this capacity for search reduction centers around its potential to eliminate clauses from the theory. Figure 3.5 illustrates this. Both the original and transformed theories entail no proofs, but under the abstracted theory it may be substantially easier to exhaust the search space. Under the standard search order, the master must close the leftmost branch before considering the rightmost one. Here, none of the solutions for the branch on the left can satisfy the branch on the right. Under the original theory, the number of subproofs for the first branch is exponential in the depth of search. A naive backtracking search must consider all of these. The abstracted theory eliminates two clauses and leaves only linearly many solutions to the leftmost branch.

3.5.2 Nagging Protocol

DALI was developed on a network of Unix workstations, with sockets providing the basic communication facility. During its search, the master process periodically polls for incoming messages and handles them as they arrive. The communication layer used by nagging has also been used by other external components of the theorem prover including a user interface and an explanation-based learning module. Thus, the overhead of this layer is effectively amortized over many components of the system.

As described in Chapter 2, when work must be exchanged between processes, DALI resorts to reconstructing the state rather than copying it. Before search begins, the master transmits the domain theory to each nagging process. When a nagging process reports that it is idle, the master selects some node $\Delta \in T$ that lies along its current search path and sends Δ to the nagger by transmitting the sequence of inference operations used to derive Δ from $\Delta_\emptyset$. The nagger then deterministically repeats the node's derivation according to the specifications sent by the master. DALI's depth-first search order is advantageous here. Each time a nagger receives a new problem, rather than reconstructing the node from the root of T, it derives the new tableau incrementally from the tableau given in the previous problem message. As illustrated in Figure 3.6, if the old and new nodes lie in a common subtree, the nagger simply backtracks to the root of that subtree and derives the new node

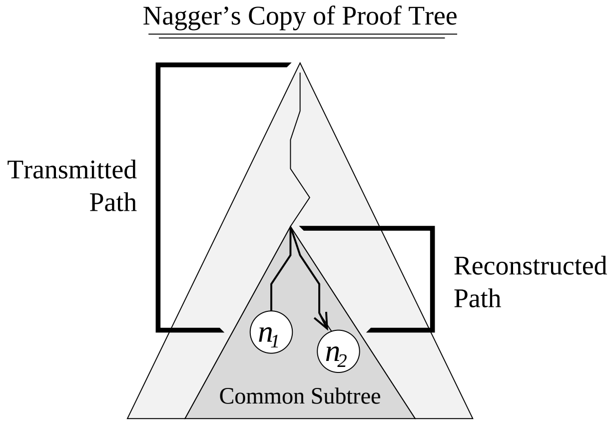


Figure 3.6: Construction of n_2 incrementally from n_1. If successive problem messages make reference to nodes in a common subtree, the nagger can reconstruct the new tableau as a variant of the old one.

from there. Since depth-first search explores the search space one subtree at a time, DALI maximizes the possibility that the tableau of one problem message will share a subtree with its predecessor.

Once the nagger receives some node $\Delta \in T$, it selects an applicable problem transformation in $\mathcal{P}$ or $\mathcal{A}$. Both $\mathcal{P}$ and $\mathcal{A}$ grant the nagging process a high degree of flexibility in transforming the master's tableau. DALI's mechanism for exchanging work through reconstruction provides the nagger with a detailed snapshot of the state of the master's search. In selecting a transformation, the nagger may analyze this snapshot and attempt to maximize its potential for search reduction. Various approaches for choosing a transformation have been implemented including ones that are simply random [SS94], based on elementary heuristics or tailored for a particular domain. This flexibility in selecting a transformation is intended to be one of the fundamental mechanisms by which DALI can exploit domain-specific knowledge when it is available.

DALI permits the master to assign the same node to more than one nagger, thus allowing two or more naggers to try to prune the same subtree. This duplicate node assignment may be worthwhile provided naggers working on the same subproblem employ different transformations; if one of them selects a poor transformation, it is still possible that some other will select a good one. To encourage this, DALI forbids two naggers from considering the same transformation of the same subproblem. As part of each problem message, the master informs the nagger of how many other naggers are already working on transformed versions of the same problem. All nagging processes have the same set of problem transformations at their disposal.

Upon receiving a new problem message, a nagger can, if necessary, reproduce transformations made by sibling naggers to ensure that the transformation it chooses is unique.

After transmitting a prune message, the master continues its search for a proof, and the nagger looks for a proof of its transformed problem. Since both $\mathcal{P}$ and $\mathcal{A}$ map from first-order search problems to other first-order search problems, the nagging processes can use the same search engine as the master. In DALI, master and nagger are implemented by the same program, each simply playing a different role. This sharing of the search procedure will be beneficial in Chapter 5, where the basic model elimination search model is enhanced. Because they share a single search procedure, enhancements to the master's search procedure automatically benefit the naggers.

If the nagger finds a proof, it discards it and reports as idle to the master. If the nagger exhausts its search space without a proof, it sends a prune message. As described in Chapter 2, the receipt of a prune message permits the master to discard nodes from the current search fringe. In DALI, nodes on the fringe are not actually constructed until they are needed. The fringe is maintained implicitly as untried alternatives at each choice point frame. DALI effectively discards nodes from the fringe by backtracking to discard choice point frames. If a prune message permits the master to abort the exploration of some $T(n)$, DALI simply backtracks to the choice point from which n was generated and continues the search from there.

3.5.3 Nagging Properties

DALI's use of iterative deepening is particularly advantageous where nagging is concerned. As mentioned in Section 2.1, nagging requires that $f(n, T)$ occasionally be finite. Although, for a given theory, the model elimination search space may be infinite, within each iteration any $T(n)$ is bounded by the depth limit and effectively finite. As justified by Theorems 3.1 and 3.3 the master's depth bound can be enforced on each nagging process. As a result, each of their model elimination search problems will be similarly finite.

Since the master's search space is finite at each iteration, Theorems 3.1 and 3.3 in conjunction with Theorem 2.6 ensure that, for a given iteration, nagging under $\mathcal{P}$ or $\mathcal{A}$ does not interfere with the completeness of the master's search procedure. Some care must be taken in extending this result to full iterative deepening search. If each iteration were carried out independently, completeness would be maintained. However, if messages sent in one iteration do not arrive until a later iteration, they may inappropriately prune part of the master's now deeper search tree. Consider a subtree $T(n)$ that does not contain a solution until the $k+1^{st}$ iteration. Nagging during the k^{th} iteration may generate a prune message for $T(n)$. If such a message does not arrive until the $k+1^{st}$ iteration, it may incorrectly discard $T(n)$ and the solution it now contains. This error may be easily avoided by tagging each message with the current iteration. Any prune message pertaining to a completed iteration may be ignored. DALI employs a variant of this tagging scheme in which each nagging attempt is given a unique identifier. When an identifier expires, messages tagged with it may be ignored. Chapter 4 describes how these tags are used to guarantee that incoming messages are pertinent.

Unlike the simple problem transformation function proposed for N-queens in Chapter 2, $\mathcal{P}$ and $\mathcal{A}$ prescribe functions of fairly general applicability. Both of these classes 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. However, there are some nontrivial Horn-clause theories that feature clauses of no more than two literals. Tableaux generated under these theories never exhibit more than one open branch. For such a theory, any transformation in $\mathcal{P}$ must be equivalent to one of the trivial transformations f_1 or f_2 given in Chapter 2. Other theories, in domains like combinatory logic [WM88], use only a small number of distinct symbols. In these cases, obscuring the distinction between any pair of non-variable symbols would represent a dramatic change to the theory, and nagging under $\mathcal{A}$ is likely to be ineffective.

More acutely, it is possible, through transformation, to render any theory impervious to nagging under $\mathcal{P}$ and $\mathcal{A}$. Post demonstrates that, for any deductive system, there is an equivalent system using only two symbols and in which each rule has only one premise [Pos43,Mas87]. This result can be easily applied to first-order logic. Thus, the applicability of nagging may be less a function of the problem under consideration than it is of the formulation of that problem and the particular transformation employed. The hope is that most natural problem formulations will lend themselves to nagging under $\mathcal{P}$, $\mathcal{A}$ or some variant.

Chapter 4

Nagging Protocol Extensions

The model of nagging presented in Chapter 2 and applied to first-order inference in Chapter 3 is attractive because of its simplicity, inherent fault-tolerance and scalability. However, this basic framework admits many natural refinements and extensions that support greater utilization of nagging processes and additional opportunities for search pruning. In many cases, these changes to the protocol retain or even strengthen the desirable properties associated with nagging and induce only a small amount of additional overhead.

This chapter demonstrates how the basic nagging protocol can be extended to be more effective in practice. These enhancements may be broadly classified under two headings. Some are completely general and are applicable to nagging in any search problem and with any transformation function. These are presented first, with DALI's nagging implementation as a reference. Next, extensions specific to nagging in model elimination are given. These exploit properties particular to the relevant problem transformation functions and often permit search pruning even when it is not justified by the basic protocol.

4.1 General Refinements

To expedite discussion and relevant proofs, the nagging protocol of Chapter 2 was kept quite simple. There are several natural refinements to this protocol. These include notifying naggers when their search becomes irrelevant, exploiting information about the search tree when selecting nagging targets and applying nagging recursively to help reduce search for the naggers themselves. These refinements can greatly improve performance in practice and, because of their generality, can be applied to nagging in all of its instantiations.

4.1.1 Irrelevant Search

The nagging protocol introduced in Chapter 2 sometimes permits a nagging process to perform work that cannot lead to pruning of the master's search. Once a nagger begins exploring some $f(n,T)$, it does not interact with other processes until its search of $f(n,T)$ is complete. While the nagger explores its $f(n,T)$, the master continues its serial search of T. Nagger and master effectively compete to show that $T(n)$ is void of solutions. If the nagger completes its search of $f(n,T)$ first and finds no solutions, it can cause pruning within the master process. However, if the master completes its search of $T(n)$ first, the nagger is allowed to continue exploring $f(n,T)$ even though this effort can no longer benefit the master.

A lazy policy for avoiding this type of irrelevant search can be easily incorporated into the nagging protocol. Whenever the master finishes searching some $T(n)$, it may inform any naggers working on transformed versions of $T(n)$ that their efforts are now futile. The nagging protocol could be augmented with a *moot* message to inform naggers when their search became irrelevant in this manner. Naggers receiving a moot message would be rendered idle and free to request new subproblems. However, since the receipt of a moot message always elicits an idle message from the nagger, the moot message is completely superfluous. Instead of waiting for the moot-idle round trip before assigning the nagger a new problem, the master can simply select a new subproblem for the nagger and transmit a new, *preemptive* problem message. If a nagger receives a new problem message during search, it simply discards the previous problem and adopts the new one[1].

This extension of the nagging protocol requires no new messages, but it does represent a change in policy. In particular, the master must, upon exhausting a subtree, identify any naggers still working on transformed versions of that subtree. Also, while naggers were previously free to work on their transformed search problems without interruption and without attention to communication with other processes, they must now be prepared to receive new assignments at arbitrary points in their search. Naturally, this need to monitor communication during search will detract from a nagger's performance. However, since the master process is already obliged to handle messages during search, extending this responsibility to the nagging process is consistent with the use of a single search procedure for both master and nagger. It is also a prerequisite for many other extensions to the nagging protocol.

Charging the master with informing naggers when their search becomes irrelevant means that communication is no longer initiated solely by the nagging processes. A

[1]In its original form, the nagging protocol permitted preemption. It was omitted from Chapter 2 to simplify presentation and proofs.

preemptive problem message is generated as a consequence of the master's search and not in response to prompting from a nagger. Communication on the master's side remains asynchronous, but the master must be prepared to preempt naggers when necessary. In DALI's implementation of nagging, this is easily accomplished. The master process retains a record of each nagging attempt at the corresponding choice point frame. If a nagger is working on some $f(\Delta, T)$, the master records this in the choice point frame from which Δ was generated. Since DALI explores T one subtree at a time, the search returns to this choice point frame only after it has finished exploring $T(\Delta)$ and is ready to consider a sibling of Δ. Thus, upon completing the exploration of some $T(\Delta)$, detecting outstanding attempts at nagging $T(\Delta)$ requires only a constant-time check of the current choice point frame.

The admission of master-initiated communication leads to ambiguity in the nagging protocol. Consider the cases covered by Figure 4.1. In both schedules the master completes exploration of $T(n_1)$ and assigns a second, preemptive problem. The nagger completes its search of its transformed subproblem $f_a(n_1, T)$ before it receives the new problem. In the schedule on the left, the nagger finds a solution to its transformed problem and transmits an idle message before it receives the new assignment. On the right, the nagger finds no solutions and transmits a prune message before the new problem arrives. In both cases, the master cannot tell if the nagger's message is in response to the first or second problem. If the master assumes that it is a response to the search of $f_b(n_2, T)$, the nagger on the left may be assigned a third problem, forfeiting any pruning its search of second problem may have provided. Worse still, the situation on the right might cause the master to prune a subtree that contains solutions. These situations were not previously possible because there could never be more than one message in transit between the master and any one nagging process.

These errors are easily avoided by including an extra token in each message. Whenever a nagger is given a new problem, the master generates a unique token and transmits it to the nagger along with the problem message. Upon receiving a problem message, the nagger adopts the new token and begins working on the new problem. The nagger includes its current token as part of each message it transmits. Whenever the master issues a preemptive problem message, it invalidates the nagger's previous token. Any message the master receives with an invalid token may be quietly ignored. Clearly, this time-stamping policy avoids the problems illustrated in Figure 4.1. Whenever the master receives a message, it can easily determine whether or not it pertains to the most recently issued problem[2].

[2]This is the mechanism mentioned in Chapter 3 for preventing prune messages from one iteration from bleeding over into the next. Completing an iteration requires that the master backtrack out

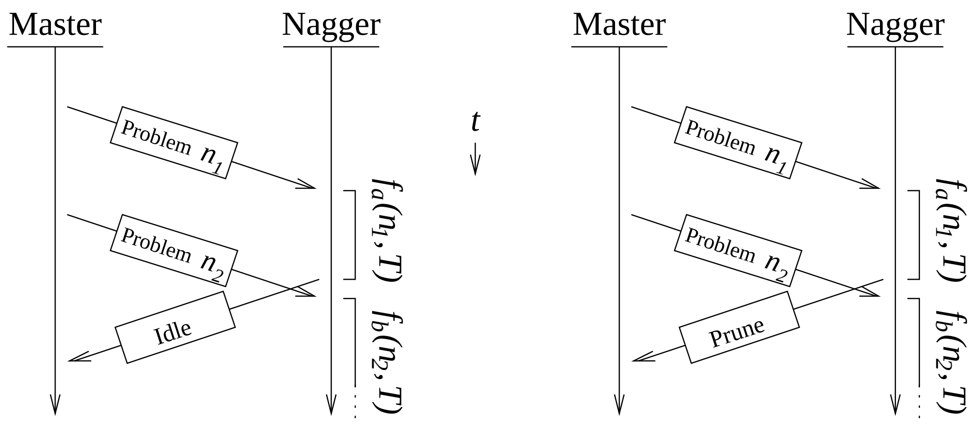


Figure 4.1: Ambiguity in the extended protocol. With the possibility of problem messages that are not nagger-initiated, the master may missattribute the context of a nagger's prune or idle message.

Preemption does not compromise nagging's solution ordering, non-increasing search and completeness properties. For any schedule that includes preemptive problem messages, there is an alternative schedule that omits them but preserves the sequence of nodes explored by the master. Figure 4.2 demonstrates this. In the left-hand schedule, the master issues a preemptive problem message to change the nagger's work assignment. There is an alternative schedule that exhibits the same search order for the master and is permissible under the basic nagging protocol of Chapter 2. In the schedule on the right, the nagger transforms $T(n)$ by the non-informative function f_1 given in Chapter 2. Consequently $f_1(n_1, T)$ contains a solution regardless of whether or not $T(n_1)$ does. Using f_1, the nagger on the right will find a solution immediately and, with sufficient liberties with message latency, the ensuing idle message can be made to arrive just as the master completes its search of $T(n_1)$. The well-timed arrival of this idle message renders the second problem message non-preemptive but does not otherwise change the master's search. From the master's point of view, any search with preemption is equivalent to a non-preemptive search that is padded with extra idle messages. The proofs of Theorems 2.3, 2.5 and 2.6 still hold under the extended protocol since they are concerned only with the schedule for the receipt of prune messages.

Informing naggers when their search problems become irrelevant is also conducive to fault tolerance. The form of fault tolerance described in Theorem 2.7 guarantees that the master will never produce erroneous results but promises little regarding

of all subtrees. In so doing, it invalidates any outstanding tokens.

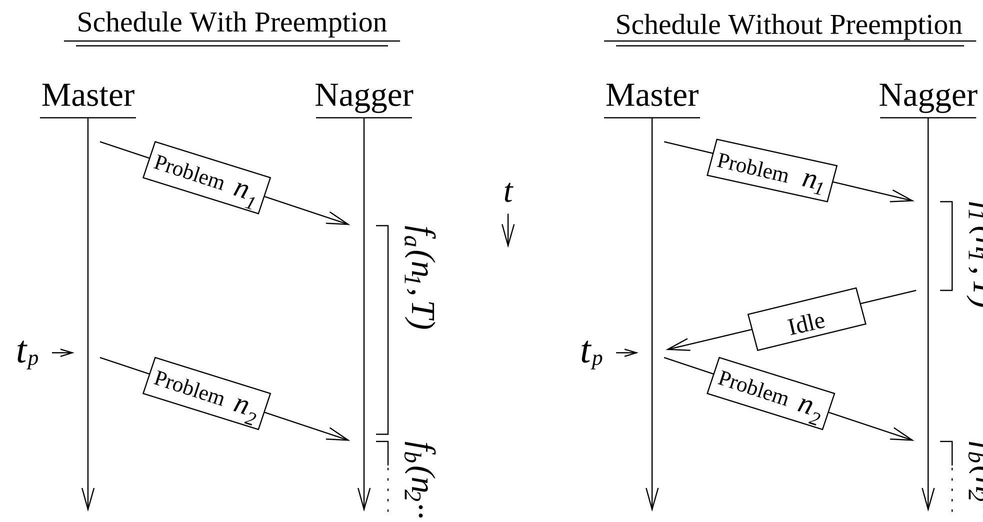


Figure 4.2: Equivalence between preemptive and non-preemptive protocols. The schedule on the left is preemptive. The one on the right is equivalent for the master but non-preemptive.

the behavior of the nagger. If a message of any type is lost, the nagger will continue waiting for the assignment of a new problem indefinitely, never again contributing to search reduction. Provision for a preemptive problem message offers the possibility to re-acquire naggers that have been separated from the search as a result of a lost message. To see this, consider the master's appraisal of a nagger's current state. The master always believes either that the nagger is idle or that it is busy:

1. If the master believes that the nagger is idle, it will assign a new, current problem. Afterward, the master assumes the nagger is busy. If the nagger receives the message, it will adopt the new problem regardless of its previous state.

2. If the master believes that the nagger is busy, it expects the nagger to be working on some $f(n,T)$. Whether or not this is actually the nagger's current task, if the master completes its search of $T(n)$ before the nagger gives a report pertaining to $T(n)$ the nagger's search will be deemed irrelevant and the master will be licensed to assign a new problem.

The loss of messages may cause master and nagger to disagree about the nagger's current task, but, if reliable communication is eventually restored, the nagger may eventually be reconciled. As covered under the second case, reinstatement of the nagger is contingent upon the master exhausting some subtree. Given DALI's iterative deepening search policy, all subtrees are finite, and, barring the discovery of a solution, the master will eventually emerge from any subtree it enters and issue new problems to any temporarily lost naggers. In general, any lost nagger will eventually be reinstated so long as T is finite.

4.1.2 Recursive Nagging

One of the limitations of nagging as proposed in Chapter 2 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 to work distribution is inherently inconsistent with an interest in scalability.

Recursive nagging is a strategy for reducing this bottleneck while increasing the effectiveness of nagging processes. All of the transformation functions introduced thus far generate alternative search problems in a domain that is either the same as or very similar to the original. The classes $\mathcal{P}$ and $\mathcal{A}$, for example, comprise mappings from model elimination search trees to other model elimination search trees. Not only does this permit master and nagger to use the same search procedure, but it also permits naggers to, themselves, be nagged. If nagging is effective at reducing search in the master's domain, it may be similarly effective at reducing search in the naggers' problems, each being a derivative of the master's problem. As illustrated in Figure 4.3, this arrangement permits a large number of processes to be utilized without requiring a single administrator to directly control them all. The top-level process acts as the master for a comparatively small number of naggers; each non-terminal nagger also serves as the master for its subordinate naggers. Although naggers that are indirectly connected to the top-level master do not directly assist in advancing the search, they may speed the operation of their parent naggers, thus permitting them to prune more quickly.

It would be possible to make use of recursive nagging even if master and nagger worked on different types of problems. This would, however, require the formulation of new transformation functions and the implementation of search procedures for each recursive level of nagging. Preserving the problem type across transformation is not only an implementational convenience, it also facilitates a uniform approach to nagging and recursive nagging. While a first-level nagger searches some tree of the form $f(n, T)$ for some $n \in T$, its second-level naggers may search trees of the form $f(n', f(n, T))$ for $n' \in f(n, T)$. 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 nagger, its second-level naggers can expect to

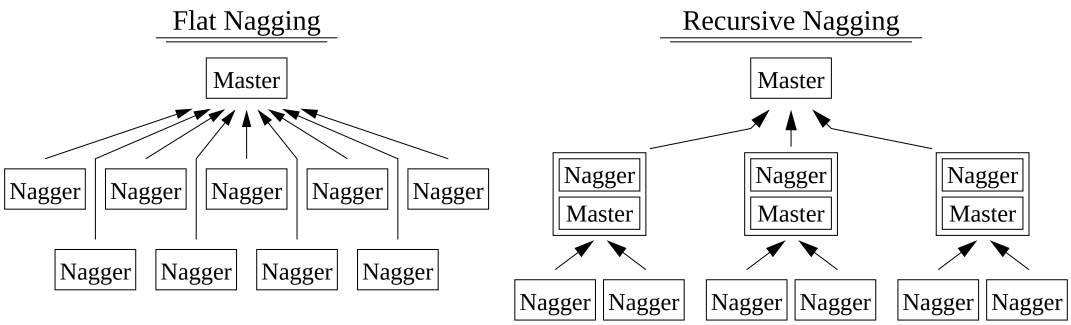


Figure 4.3: Recursive nagging model. Both arrangements use the same number of processes, but the recursive arrangement on the right does not require a single master to manage all of them.

see several subproblems.

This variation in problem granularity across the nagging hierarchy results in a nonuniform demand for interprocess communication. Naggers and recursive naggers can be arranged to capitalize on this nonuniform communication requirement. Processes demanding more frequent communication can be placed more locally in the distributed computing architecture. If necessary, multiple levels of recursive nagging can be employed to make use of more processing elements or better match the topology of their interconnections.

As in Section 4.1.1, recursive nagging requires that naggers communicate while performing search. More acutely, non-terminal naggers must play the role of both master and nagger. This is likely to exact a performance penalty, but, hopefully, one that is offset by additional search pruning. The retention of completeness in the context of recursive nagging is easily verified. If k levels of recursive nagging are in use, Theorems 2.3 and 2.6 guarantee that, for the identified classes of search procedures and trees, naggers at level $k-1$ will not skip solutions. Applied inductively, this argument demonstrates that at no level of recursive nagging will a process be compelled to skip a solution.

4.1.3 Deterministic Regions in T

So far, discussion of a nagger's subproblem $f(n,T)$ has concentrated on the formulation of some $f \in \mathcal{F}$. Little attention has been given to the problem of choosing an appropriate target $n \in T$ for nagging. A policy for selecting such a node is much akin to an OR-ordering heuristic. In fact, any completely reliable policy would be sufficient to eliminate search altogether. If a particular subtree contains solutions,

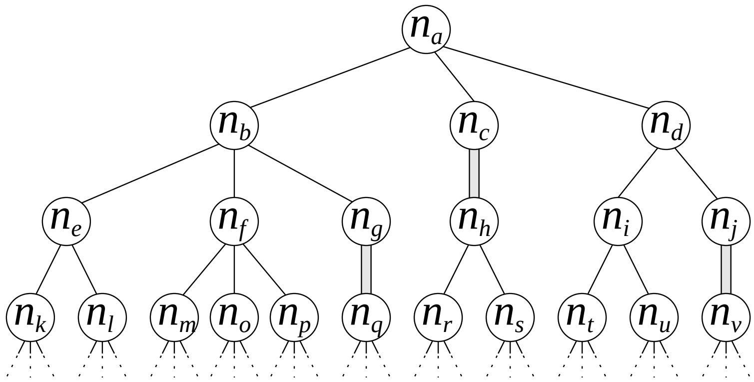


Figure 4.4: Determinism in the search tree. Although all search trees contain instances of nondeterminism, they may also contain some deterministic choices. Deterministic choices are highlighted here.

nagging on that subtree would clearly be unproductive; any heuristic able to reliably reject such subtrees could be used as a perfect search ordering heuristic. Thus, any realistic rules for selecting feasible nagging targets are likely to be approximate in nature.

There are, however, situations in which a particular node is clearly preferable to some others. Figure 4.4 demonstrates this. Informally, search problems are characterized as being uniformly nondeterministic, each choice leading to still other choices. This is often a fair account since it emphasizes a search tree's general inclination toward exponential growth, but it should not obscure the fact that many search trees feature local instances of determinism. The highlighted edges in Figure 4.4 indicate such deterministic choices. When a node has more than one child, there is a tradeoff between nagging on that node or on one of its children. In this example, pruning $T(n_b)$ is obviously preferable to pruning only $T(n_f)$, but, since $|T(n_b)| > |T(n_f)|$, exploring a transformed version of $T(n_b)$ is likely to demand more search than a transformed version of $T(n_f)$. Furthermore, $T(n_b)$ is at least as likely to contain a solution as is $T(n_f)$. Where a node has only one child, this tradeoff is cleanly resolved. Although pruning $T(n_c)$ or $T(n_h)$ facilitates virtually the same amount of search reduction, exploration of some $f(n_h, T)$ may require substantially less search than $f(n_c, T)$. The topmost choice in $T(n_c)$ is deterministic, but the first choice in some $f(n_c, T)$ may not be.

There are even cases in which $f(n_c, T)$ contains solutions, rendering it ineffective for nagging, while $f(n_h, T)$ does not. This is easily demonstrated in the context of model elimination. The theory given in Figure 4.5 is a variant of the one given in

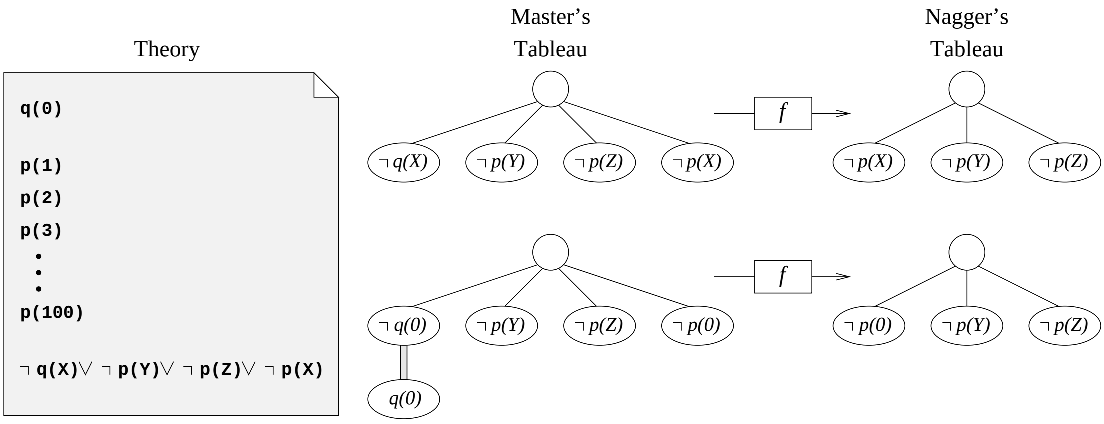


Figure 4.5: Determinism in first-order logic. Deterministic closing of the leftmost branch can have profound influence on the size of the transformed search space.

Figure 3.4. Here the master finds only one option for closing the leftmost tableau branch. The nagger's transformation under $\mathcal{P}$ is highly sensitive to whether or not the $\neg q(X)$ branch has been closed. The pictured transformation discards the leftmost branch and reorders the remaining branches. If the nagger is assigned the tableau before its leftmost branch is closed, transformation not only sacrifices the deterministic inference step but also introduces a solution where none exist in the original search space. Transformation after closing the leftmost branch grants the nagger a comparatively small search space which contains no solutions. Thus, even when a particular step is deterministic, its execution may constrain the problem in ways that survive transformation and assist the nagger's search.

DALI attempts to take advantage of local instances of determinism in the theory. This can be particularly valuable in some Prolog theories, where the true nondeterministic choices are often surrounded by liberal stretches of supporting, deterministic computation. This is evident in the Prolog formulation of N-queens given in Figure 4.6 [Tic91]. All real choices in this program are made when selecting queen placements in the `perm` procedure. In particular, they are made in selecting one of the two clauses defining `del`. Extensions by this clause are the points at which nagging will be most helpful; remaining clauses simply support initialization and threat detection. DALI can detect some deterministic choice points in a Horn-clause theory through compile-time analysis. Although such methods are incapable of detecting all determinism, they are helpful in many cases including this particular N-queens theory.

DALI supplements this approach with a run-time technique for detecting determinism. This run-time technique is typically more effective on theories that are less

```
queen(N, A) :- gen(N, L), perm(L, A), test(A).

gen(0, [ ]).
gen(N, [N | T]) :- N > 0, N1 is N-1, gen(N1, T).

perm([ ], [ ]).
perm([H | T], [A | P]) :- del(A, [H | T], L), perm(L, P).

del(X, [X | Y], Y).
del(X, [Y | Z], [Y | W]) :- del(X, Z, W).

test([ ]).
test([H | T]) :- safe(T, H, 1), test(T).

safe([ ], _, _).
safe([H | T], U, D) :- H-U ≠ D,
                       U-H ≠ D,
                       D1 is D+1,
                       safe(T, U, D1).
```

Figure 4.6: A naive formulation of N-queens in Prolog. Most clauses represent deterministic computation.

imperative and more logical in nature. Many theories may contain no consistently deterministic components or none that are easily detected through compile-time analysis. There are, however, choices that are rendered effectively deterministic as the search progresses. Node n_p in Figure 4.4 is one of several children of n_f, but once a left-to-right depth-first search reaches n_p, all of its siblings will have already been considered. In this case, nagging on either n_p or n_f stands to prune the same amount of search, the unexplored portion of $T(n_p)$. As above, however, nagging on n_p may afford a smaller search space and greater potential for search pruning. This special treatment of a node's rightmost child is called the *last-call rule* and is something of a nagging analog to the WAM's last-call optimization [AK91].

Definition 4.1 (Last-Call Rule) *If n_a is the rightmost child of $p(n_a)$ and the master has exhaustively explored all subtrees $T(n_b)$ for $n_b \in c(p(n_a))$, $n_b \neq n_a$, then $p(n_a)$ must not be chosen as a target node for a new problem message.*

In practice, it may not be obvious when a node's rightmost child is being explored. In the interests of efficiency, DALI inherits the WAM's policy of not explicitly generating a node until immediately before it is explored. Thus, it may not be clear which applicable reductions and extensions remain until the appropriate unification has actually been attempted. To get around this problem, DALI employs an optimistic estimate of a node's untried siblings that is based on some of the WAM's preexisting computational machinery. After considering all applicable reductions at each search node, DALI performs an indexing operation to determine which clauses and literals in the theory might be candidates for extension[3]. Once DALI begins considering a node's candidate extensions, indexing is used as an optimistic approximation for the number of extensions that actually remain. If the index indicates that the master is currently exploring a node's last child, the last-call rule is applied.

4.2 Transformation-Specific Refinements

In addition to the use of problem transformation functions specific to its operation, there are other opportunities to exploit features of the model elimination proof calculus within the nagging protocol. Because master and nagger perform search in closely related domains, search in $f(\Delta, T)$ may reveal more information about $T(\Delta)$

[3]This operation is in the spirit of the WAM's clause indexing in that it is intended to quickly filter literals that obviously cannot match the selected branch. Much of the computation implicit in the index may be done during theory compilation. As a result, the index is usually more efficient than full unification, and, whenever possible, rejecting an infeasible match via the index is preferable. While the WAM indexes literals on their predicate and leftmost constant or function symbol, DALI generalizes this scheme to exclude literals differing on any non-variable symbol.

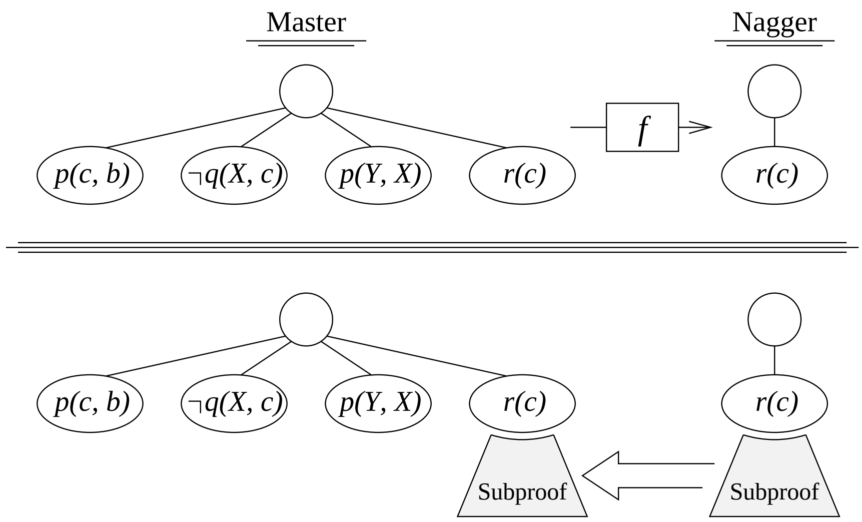


Figure 4.7: Potential utility of nagger-discovered solutions. When a nagger working under $\mathcal{P}$ discovers a solution, it has found subproofs for branches in the master's tableau. In some cases, the master can make productive use of these subproofs.

than what is promised by Theorem 2.1. It is possible to make use of a nagger's search results before $f(\Delta, T)$ has been completely explored and, in some cases, even when $f(\Delta, T)$ is found to contain a solution. These opportunities are particularly evident when using transformations in $\mathcal{P}$. DALI has been enhanced to take advantage of several such opportunities for increased search reduction.

4.2.1 Completed Subproofs

As described in Chapter 2, naggers assist the master when they fail to solve their transformed search problems. In first-order logic, they help to reduce search when they fail to find a proof. If a nagger finds a proof, it simply discards it and requests more work. In the general case, there is no simple relationship between solutions to the master's problem and the nagger's transformed problem. As a result, there is no obvious way to make constructive use of a nagger-detected solution.

For functions in $\mathcal{P}$, however, there are opportunities to benefit from solutions to the transformed problem. Consider the nagging attempt depicted in Figure 4.7. Here, 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 subproof, it would not have to re-derive it.

Even when problem transformation functions are drawn from $\mathcal{P}$, care must be taken in grafting a nagger's subproof into the master's tableau. Solutions found by such a nagger comprise subproofs for branches in the master's tableau, but it is not always straightforward or desirable to adopt them. Any solutions found in $f(\Delta, T)$, $f \in \mathcal{P}$ could be incorporated into Δ, but, by the time the nagger finds its solution, the master's search may have moved on to some $\Delta' \in T(\Delta)$. The nagger's subproofs may not be consistent with other inferences the master makes from Δ to Δ'. This might be the case if, for example, the nagger in Figure 4.7 discovered a subproof for the $p(Y, X)$ instead of $r(c)$. In its attempt to close all four tableau branches from left to right the master may have already started constructing a subproof for $\neg q(X, c)$ or even $p(Y, X)$. In so doing, the master may have bound the variable X. If the nagger's subproof of $p(Y, X)$ binds X to something different, it will not be possible it integrate it into the master's partial proof without retracting some inference operations.

Adopting a nagger's subproof may not be beneficial even when it is consistent with the master's current partial proof. With its standard selection function, the master ensures that branches in the left portion of the tableau are closed before attempting to close branches to the right. As a result, some potential subproofs for branches on the right may never need to be considered because they are inconsistent with all subproofs for branches to their left. If a subproof found by a nagger is among those that the master would never have to consider, imposing it on the master's tableau may actually increase search. In Figure 4.7, a foreign subproof for $p(Y, X)$ may force the master to try to complete the proof with a binding for X that it would never consider under the standard search order.

Notwithstanding these difficulties, there are situations where naggers produce unquestionably useful subproofs. For example, the nagger's subproof of $r(c)$ in Figure 4.7 can be applied to the master's tableau without risk of adversely affecting the search. This is because the branch the nagger manages to close shares no variables with other branches; its subproof cannot bind variables in a way that conflicts with subproofs of the other branches. Furthermore, the nagger's subproof cannot force the master to perform search that it would not otherwise perform.

DALI takes advantage of subproofs that, like $r(c)$, cannot adversely affect search performance. The following condition on subproofs is used to determine when it is safe to adopt a nagger's solution:

Definition 4.2 (Weak Locality) *Let B be the set of open branches in tableau Δ and let $\hat{B} \subseteq B$. A partial proof $\langle op_1, \ldots op_k \rangle$ is a* weakly local subproof *for B in Δ if the following conditions are met:*

- *The sequence $\langle op_1, \ldots op_k \rangle$ closes all branches in $\hat{B}$ when applied to Δ.*

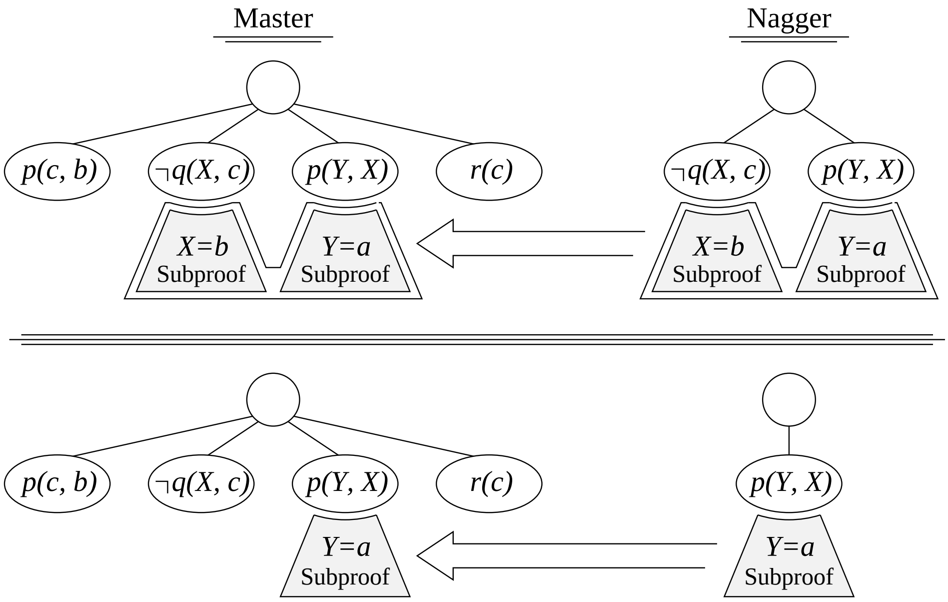


Figure 4.8: Utility of weakly local subproofs. The master can safely use a nagger-discovered subproof as long as it does not affect open branches elsewhere in the tableau.

- *For any branch $\beta \in (B - \hat{B})$, when $\langle op_1, \ldots op_k \rangle$ is applied to Δ, no children are added to β.*
- *The labeling of $\left\{ n \mid n \in \beta, \beta \in (B - \hat{B}) \right\}$ in $op_k \ldots op_1 \Delta$ is a syntactic variant of its labeling in Δ.*

Weak locality is weak in that it permits subproofs to bind variables occurring elsewhere in the tableau provided they do not affect other open branches. The nagger's subproof for $r(c)$ in Figure 4.7 qualifies since it only affects portions of the tableau in the $r(c)$ subtree. Weak locality covers other cases where the nagger's solution may be useful to the master. Figure 4.8 illustrates some of these. If a partial proof closing $\neg q(X, c)$ and $p(Y, X)$ is found, then it may safely be imported even if it binds Y and X. Variable X appears in more than one open branch, but its binding affects only $\neg q(X, c)$ and $p(Y, X)$. Similarly, a subproof for $p(Y, X)$ alone is acceptable provided it does not bind X.

Theorem 4.1 *Let Δ be an open tableau with at least two open branches. Let B be the set of open branches in Δ and let $\hat{B}$ be a nonempty, proper subset of B. If $\langle op_1, \ldots op_k \rangle$ is a weakly local subproof for $\hat{B}$ in Δ and op' is an inference operation for a branch in $B - \hat{B}$, then the following are true:*

1. *if op' is not applicable to Δ then op' is also not applicable to $op_k \ldots op_1\Delta$.*
2. *if op' is applicable to Δ then op' is also applicable to $op_k \ldots op_1\Delta$.*
3. *if op' is applicable to Δ then $\langle op_1, \ldots op_k\rangle$ is still a weakly local subproof for $\hat{B}$ in $op'\Delta$.*

Proof Sketch: *The first property follows easily from the operator transposition lemma. The second property relies on the fact that $\langle op_1, \ldots op_k\rangle$ cannot change the labeling on the target branch of op' apart from syntactic variation. Finally, showing that $\langle op_1, \ldots op_k\rangle$ is still weakly local for $op'\Delta$ requires that all three conditions of weak locality be maintained.*

Theorem 4.1 demonstrates that weak locality is sufficiently strong to exclude undesirable subproofs. Weaker conditions could be employed to more liberally exploit naggers' successes, but weak locality offers some attractive computational properties. In particular, it is a constraint on the subproof found by the nagger and does not depend on what subproofs exist for other open branches. The master does not have to encourage or even detect weak locality. The nagger, having all the necessary information, may check its own solutions to see if they are weakly local before reporting idle. If a nagging process discovers a solution $\hat{\Delta}$ in $f(\Delta, T)$, it checks the weak locality condition against Δ and $\hat{\Delta}$ and reports to the master if it is satisfied. Upon receiving such notification, the master may discard any incomplete attempts to close the nagger's branches and rely on the nagger's subproof instead.

Theorem 4.1 also demonstrates that weakly local subproofs are as good as any other subproofs for their associated branches. The presence of a weakly local subproof does not reduce the number of subproofs applicable to branches outside its influence. In fact, no matter how the rest of the tableau is filled in, a weakly local subproof will still be applicable. If a weakly local subproof for some set $\hat{B}$ is known to exist, then, since it can be applied on demand, a theorem prover may exercise some discretion in choosing when to actually integrate it. In exploiting a weakly local subproof found by a nagger, DALI defers application until all other branches have been closed. As a result, when a nagger finds such a subproof, it does not need to transmit it to the master immediately, but only needs to report that it has been found and indicate the pertinent branches. The master then confines its efforts to those branches not closed by the nagger's subproof. It requests the subproof only after it has closed all other branches. In waiting for all other branches to be closed, DALI is assured that it will have a complete proof once the missing, weakly local subproofs are supplied. Deferring the exchange of subproofs until a proof is known to exist reduces the load on the master and the demand for communication during search. The cost of exploiting a nagger-discovered subproof is kept low until it is clear that the subproof is useful.

This lazy exchange of subproofs may be easily generalized in the context of recursive nagging. If process b is a recursive nagger for some nagger a, and process b finds a weakly local subproof, it simply informs nagger a. If nagger a subsequently finds a local subproof containing the subproof of nagger b, it informs its master. The intermediate nagger does not need to request the subproof held by nagger b before making this report; the knowledge that nagger b holds its part of the subproof is sufficient to guarantee that the subproof identified by nagger a exists and can be explicitly constructed on demand. If the subproof of nagger a is eventually needed, the missing parts will be requested then. All exchanging of subproofs takes place after search has been completed.

There is one case in which the discovery of a weakly local subproof is particularly interesting. Transformation under $\mathcal{P}$ permits the deletion of open branches in the tableau but does not require it. There are viable transformations in which the nagger retains all open branches in its master's tableau but attempts to close them in a different order. If the nagger finds a solution in this case, then it has found a subproof for all of the master's open branches, albeit in a permuted order. In this case, no branches remain for the master to close. The nagger's solution constitutes a proof for the entire problem and may be imported immediately.

Although it may benefit search performance, making use of local subproofs in this manner compromises the solution ordering property of Theorem 2.3. Consider the Horn-clause theory given in Figure 4.9. Under a serial search with the standard ordering, DALI would close the $\neg q(Y)$ branch before the $\neg r(Y)$ branch. Expanding $\neg q(Y)$ by the first matching clause in the theory would bind Y to a and would result in a refutation proof of $q(a) \wedge q(b) \wedge r(b)$. Transformation under $\mathcal{P}$ might transpose the $\neg r(Y)$ and $\neg q(Y)$ branches, causing the nagger to close $\neg r(Y)$ first. With $r(a)$ as the first clause matching $\neg r(Y)$, adoption of the nagger's subproof would yield a proof of $q(a) \wedge q(a) \wedge r(a)$ first.

Provision for importing nagger-discovered subproofs demands rather dramatic extension of the nagging protocol. As implemented in DALI, three new message are added to the basic three-message protocol of Section 2.2:

subproof-found When the nagger finds a solution in some $f(\Delta, T)$, it checks the weak locality conditions. If they are satisfied, the nagger records a copy of its subproof and issues a *subproof-found* message to the master indicating the set of tableau branches $\hat{B}$ it has successfully closed. Upon receiving a *subproof-found* message, the master discards any attempt it has made to close branches in $\hat{B}$, and, while searching in $T(\Delta)$ considers these branches closed. As with

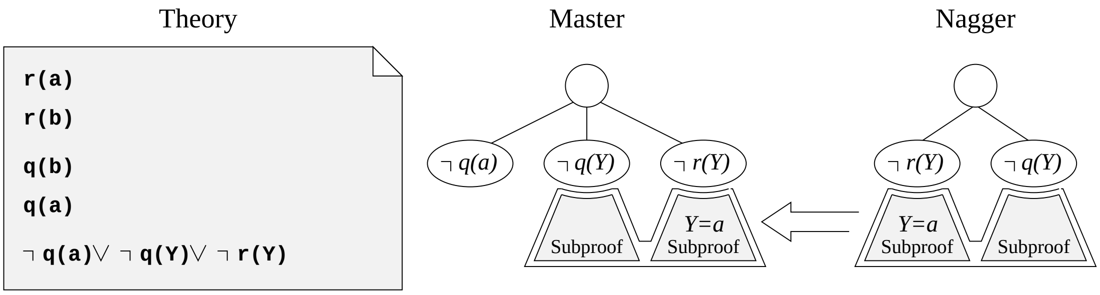


Figure 4.9: The solution ordering property is compromised when nagger-discovered subproofs are used by the mater.

the prune message, after issuing this message the nagger becomes idle. After receiving one the master is licensed to assign it a new subproblem.

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 actual subproofs from the appropriate nagging processes. The master then searches for solutions to these branches itself while it is waiting for the naggers to respond with the needed subproof.

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 with the handling of the problem message, this exchange of the subproof is achieved by transmitting the sequence of inference operations it contains. If one of these operations pertains to a branch already covered by the master's partial proof, the master's operation is retracted and the nagger's new operation is adopted instead.

The master's attempt to complete its proof even after issuing a subproof-request message represents an effort to maintain fault tolerance. The hope is that the nagger will supply the desired subproof before the master can complete its search. If, however, communication with the nagger is interrupted, the master will eventually discover a solution by itself.

In the interests of efficiency, the handling of the problem message is also modified slightly. When the nagger receives a new problem, it consults its cache of weakly local subproofs. Based on the location of the new problem in T, it may be possible to determine that some of these subproofs will never be requested. If the master issues a problem message for some tableau Δ', the nagger may safely discard copies of local subproofs pertaining to tableau Δ whenever $T(\Delta) \cap T(\Delta') = \emptyset$. The receipt of a problem message for Δ' indicates that the master is now searching in $T(\Delta')$. If $T(\Delta)$ and $T(\Delta')$ share no nodes, then the master must no longer be searching in $T(\Delta)$ and, according to the definition of the subproof-found message, will no longer consider the nagger's subproof relevant.

One subtlety in this extended nagging protocol deserves special attention. It admits the possibility that, when a global solution features more than one nagger-derived subproof, the sets of branches covered by these subproofs may overlap. When there is such an overlap, the master simply integrates naggers' subproofs as they arrive, potentially discarding portions of a previously received subproof when a new one is received. This kind of mutilation of a nagger's subproof may invalidate its weak locality. Fortunately, discarding parts of weakly local subproofs in this manner still results in the construction of a complete proof. Consider two weakly local subproofs, one closing branches $\hat{B}$ and another closing $\hat{B}'$ such that $\hat{B} \cap \hat{B}' \neq \emptyset$. Even if the subproof for $\hat{B}$ arrives before the subproof for $\hat{B}'$, all branches in $\hat{B} \cup \hat{B}'$ will still be closed. When the subproof for $\hat{B}'$ arrives, any operations in the first subproof that pertain to $\hat{B} \cap \hat{B}'$ will be discarded. If, however, the subproof for $\hat{B}$ is bloated, its surviving operations are sufficient to close all branches in $\hat{B} - \hat{B}'$. The subproof for $\hat{B}'$, being weakly local, will be applicable to the tableau even after the surviving portion of the subproof for $\hat{B}$ has closed $\hat{B} - \hat{B}'$.

Permitting the master to exploit subproofs found by its naggers provides new opportunities for performance improvement. Previously, the master was assisted only where backtracking was required. By importing subproofs found by its naggers, the master may avoid repeating the derivation of a nagger-discovered subproof. In Figure 4.7, even if nagging never causes the master to backtrack while closing 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. The master simply reconstructs the nagger's subproof once the other branches have been closed.

The protocol can be modified to help even in completely deterministic theories. The given extensions to the nagging protocol transfer all relevant subproofs to the master before a solution is reported. For some applications, however, the entire proof tree is not desired. In Prolog, for example, only a variable substitution is expected. In these cases, it may not be necessary to exchange weakly local subproofs in their entirety; it may be sufficient to simply transmit the substitutions they impose. This would free the master from even reconstructing the weakly local subproofs discovered by its naggers.

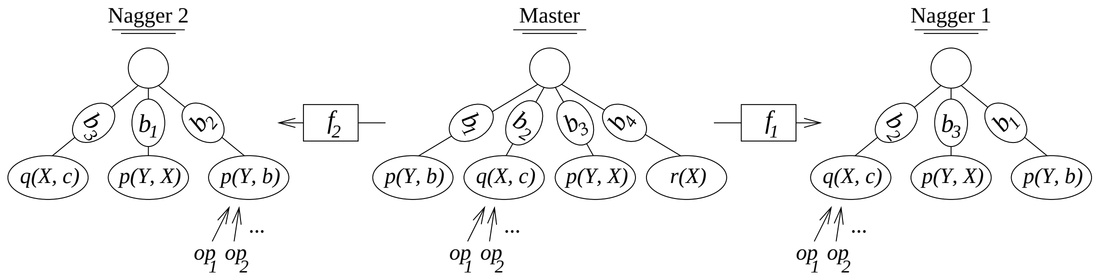


Figure 4.10: Potential for eager search pruning. The master shares search branches with its naggers. Inference operations that are rejected for these branches in one process' search space can also be rejected in another's.

4.2.2 Eager Pruning

Under the standard nagging protocol, the master is helped only after a nagger finishes exploring its $f(n,T)$, and, even then, only if $f(n,T)$ contains no solutions. This contributes to a fairly low communication overhead, since nagging processes only have to exchange messages when they finish their respective search problems, but it does leave untapped some potential for more aggressive search pruning. For a function $f \in \mathcal{P}$ there are opportunities to make use of information gained during a nagger's search even before it finishes exploring $f(n,T)$.

Consider the tableau and its transformation given in Figure 4.10. When the master explores $T(\Delta)$, it begins by trying to close the leftmost open branch β_1 and, if successful, moves on to β_2. When a nagger explores $f_1(\Delta, T)$, it begins by trying to close a different branch, β_2 and, if successful, moves on to β_3. If the nagger cannot close β_3 it backtracks and considers alternative subproofs for β_2. The nagger's attempt to close β_2 began with the first applicable inference operation, op_1. As attempts to close β_2 or other branches fail, the nagger may be forced to backtrack, reject op_1, and consider other inference operations on β_2. Thus, even when op_1 is applicable to β_2, it may not lead to a closed tableau within the current depth limit. Information about the nagger's rejection of op_1 might be useful to the master process.

If the master is successful at closing β_1, it moves on to β_2 and attempts to close it. Like the nagger, the master will try op_1 on β_2 first. If applicable, it will move on and try to close the remaining branches. The nagger's determination that applying op_1 to β_2 cannot lead to successful closing of its three branches implies that it cannot participate in closing all four of the master's branches. In fact, this knowledge may even be of use to sibling naggers. If a second nagger is exploring some $f_2(\Delta, T)$ which also features branch β_2, the knowledge of the infeasibility of op_1 on β_2 could similarly reduce its search.

Theorems 4.2 and 4.3 justify this intuition about the exchange of partial search information between processes.

Theorem 4.2 *For problem transformation function $f \in \mathcal{P}$ and tableau $\Delta \in T$, let Δ' be the root of $f(\Delta, T)$, β be an open branch in Δ' and op be an inference operation applicable to β in Δ'. If $T\,(op\ \Delta')$ contains no solutions of height j or less, then $T\,(\Delta)$ contains no solutions of height j or less that include op.*

Proof Sketch: *This proof exploits the mapping from solutions in $T\,(\Delta)$ to solutions in $f\,(\Delta, T)$ developed in the proof of Theorem 3.1. If $T\,(\Delta)$ contains a closed tableau derived using op, then this mapping exhibits a closed tableau in $f\,(\Delta, T)$ that is also derived using op. From this solution, the operator transposition lemma can be used to construct a solution that is derivable from $op\Delta'$. Since this contradicts the theorem statement, there must be no proofs in $T\,(\Delta)$ that use op.*

Theorem 4.3 *Let f be a transformation in $\mathcal{P}$, and let tableau Δ_1 be a node in T and Δ_2 a node in $T\,(\Delta_1)$. If $T\,(op\ \Delta_1)$ contains no solutions of height j or less, then either $f\,(\Delta_2, T)$ contains solutions of height j or less that do not include op or $T\,(\Delta_2)$ contains no solutions of height j or less.*

Proof Sketch: *Proof is by contradiction. Assume that all solutions in $f(\Delta_2, T)$ contain op. If $T\,(\Delta_2)$ contains solution $\langle op_1, \ldots op_k \rangle$, then the mapping developed in the proof of Theorem 3.1 exhibits a corresponding solution in $f(\Delta_2, T)$. Since this solution in $f(\Delta_2, T)$ consists of a subset of the operations in $\langle op_1, \ldots op_k \rangle$ and all solutions in $f(\Delta_2, T)$ are assumed to contain op, $\langle op_1, \ldots op_k \rangle$ must also contain op. The operator transposition lemma can be used to transform $op_k \ldots op_1 \Delta_2$ into a solution in $T\,(op\ \Delta_1)$, which contradicts the theorem statement.*

Theorem 4.2 shows that pruning the master's search in accordance with a nagger's results on its first selected tableau branch will not compromise search completeness. Theorem 4.3 shows that pruning the nagger's search in accordance with the master's results on its first selected branch may cause the nagger to miss solutions. However, these missed solutions will not affect the completeness of the master's search. Together, these theorems demonstrate that information about failed attempts to close a first selected branch can be safely shared among sibling naggers.

To facilitate the exchange of information about infeasible avenues of proof, the nagging protocol must be supplemented with one new type of message:

infeasible-choice The *infeasible-choice* message includes a tableau Δ and a set of operations $\{op_1, \ldots op_k\}$. When transmitted, an infeasible-choice message indicates that, for all op in $\{op_1, \ldots op_k\}$, $T\,(op\ \Delta)$ contains no solutions up to the current search depth limit. Messages of this type are transmitted in five situations:

1. If a master process issues a problem message for tableau Δ, it follows it with an infeasible-choice message containing Δ and $\{op \mid op < op'\}$ where the master is currently exploring $T\,(op'\Delta)$. The notation $op_a < op_b$ indicates that op_a is considered before op_b in the search order.
2. Let Δ' be the result of transforming the master's tableau Δ under some function in $\mathcal{P}$. Whenever the nagger exhausts some $T\,(op\ \Delta')$, it transmits a infeasible choice message to the master featuring Δ and $\{op\}$.
3. When a master process receives an infeasible-choice message from one of its naggers, it may forward an equivalent message to some of its remaining naggers. An infeasible-choice message for Δ and $\{op_1, \ldots op_k\}$, is forwarded to any naggers working on transformations of Δ' for any $\Delta' \in T\,(\Delta)$.
4. If a nagging process is assisted by a number of recursive naggers, any infeasible-choice messages it receives from its master are automatically forwarded to its naggers.
5. Master processes compile a list of messages received under cases 3 and 4. When a master process issues a problem message for tableau Δ, it follows it with any infeasible-choice messages it has received for tableaux Δ' where $\Delta \in T\,(\Delta')$.

If a process receives an infeasible-choice message indicating that $T\,(op\ \Delta)$ contains no solutions, it discards any partial proofs that contain op as long as it is searching in $T\,(\Delta)$ or one of its transformed subtrees. If a process' partial proof already contains op when it receives such a message, it backtracks as far as necessary to remove op.

This extension to the nagging protocol preserves search completeness; all five scenarios for the exchange of an infeasible-choice message are covered by Theorems 4.2 and 4.3. However, not all infeasible-choice messages permitted under this protocol are useful to the receiver. Since naggers compute their own problem transformations, a master generally cannot know what parts of the tableau its naggers have chosen to examine. The nagger may see infeasible-choice messages that apply to branches that it has chosen to ignore. Thus, under all cases but the second, some of the permissible, infeasible-choice messages will have no relevance to the receiving nagger's search. A more precise determination of which messages pertain to each nagger would require more communication and represent more overhead for the master. In DALI, these irrelevant messages are permitted, and the nagger is given the fairly easy task of determining their relevance upon arrival. If a nagger receives an infeasible-choice message pertaining to a branch that it discarded during transformation, it does not even need to forward this message to any of its recursive naggers since their tableaux would also omit the branch.

The infeasible-choice message permits search pruning in cases where naggers would otherwise be unable to assist their master. Under the basic protocol, a nagger only reduces search when its transformed problem does not contain a solution. Exploiting weakly local subproofs extends this potential somewhat, but the eager search pruning provided through infeasible-choice messages extends it further. This will occur if a nagging process finds a solution in $f\,(\Delta, T)$ only after rejecting some operations applicable to its leftmost open branch. Even though the master may not be able to prune $T\,(\Delta)$, it may ignore thesen infeasible operations while searching in $T\,(\Delta)$.

Eager pruning is also applicable to nagging under $\mathcal{A}$, although it has not been implemented in DALI. The clauses in the nagger's abstracted theory correspond to sets of clauses in the original theory. When the nagger backtracks to retract an inference operation that did not lead to a proof, it points to a set of infeasible operations in the master's search. Likewise, as particular sets of inference operations are rejected by the master, either through search or the receipt of infeasible-choice messages, it may demonstrate the infeasibility of individual operations in the nagger's search. When nagging under $\mathcal{A}$, infeasible-choice messages can be handled much like they are under $\mathcal{P}$. In fact, the treatment of these messages by the top-level master process is unchanged. Transmission of infeasible-choice message is handled differently in only one of the above cases:

2 Let Δ' and S' be the result of transforming the master's tableau Δ and theory S in a way consistent with $\mathcal{A}$. Whenever the nagger exhausts some $T\,(op\ \Delta')$, it transmits an infeasible choice message to the master. If op is a reduction, then the message contains Δ and $\{op\}$. If op is of the form $Ext_{\beta,C',l'}$, then let g_{tr} be the abstraction trail from S to S'. In this case, the infeasible-choice message contains Δ and $\{Ext_{\beta,C,l} \mid g_{tr}(C,l) = \langle C', l' \rangle\}$.

When an abstraction nagging process receives an infeasible-choice message, either from a recursive nagger or from its master, it must determine if discarding one of its abstracted inference operations is warranted. Reductions specified by the message may be handled as they are under $\mathcal{P}$, but the treatment of extensions depends on the abstraction trail. Extension $Ext_{\beta,C',l'}$ may be discarded by the nagger once infeasible-choice messages have been received, implicating all extensions $Ext_{\beta,C,l}$ such that $g_{tr}(C,l) = \langle C', l' \rangle$. Theorems 4.4 and 4.5 justify this policy.

Theorem 4.4 *For problem transformation function $f \in \mathcal{A}$ and tableau $\Delta \in T$, let Δ' be the root of $f(\Delta, T)$, β be an open branch in Δ' and op' be an inference operation applicable to β in Δ'. Let S be the theory corresponding to T, S' be the abstracted theory constructed under f and g_{tr} be the abstraction trail from S to S'. If $T\,(op'\Delta')$ contains no solutions of height j or less, then $T\,(\Delta)$ contains no solutions of height j or less that include op where op satisfies the following:*

- *If op' is of the form $Red_{\beta,n}$ then $op = Red_{\beta,n}$.*
- *If op' is of the form $Ext_{\beta,C',l'}$ then $op = Ext_{\beta,C,l}$ where $g_{tr}(C,l) = \langle C', l' \rangle$.*

Proof Sketch: *Proof is by contradiction. Given a proof in $T\,(\Delta)$ containing op, local operator transpositions can be used to construct an equivalent proof in $T\,(op\Delta)$. The construction used in the proof of Theorem 3.3 can be used to transform this solution in $T\,(op\Delta)$ into a solution in $T\,(op'\Delta')$. Since the theorem statement stipulates that no solutions exist in $T\,(op'\Delta')$, there must be no proofs in $T\,(\Delta)$ that use op.*

Theorem 4.5 *Let f be a function in $\mathcal{A}$, and let $\Delta_1 \in T$ and $\Delta_2 \in T\,(\Delta_1)$. For theory S, let S' be the abstraction of S on which $f\,(\Delta_2, T)$ is based. Let op' be an inference operation under S' and let O be a set of inference operations under S that satisfy the following:*

- *If op' is of the form $Red_{\beta,n}$ then $O = \{Red_{\beta,n}\}$.*
- *If op' is of the form $Ext_{\beta,C',l'}$ and g_{tr} is the abstraction trail from S to S', then $O = \{Ext_{\beta,C,l} \mid g_{tr}(C,l) = \langle C', l' \rangle\}$.*

If $T\,(op\ \Delta_1)$ contains no solutions of height j or less for any $op \in O$, then either $f\,(\Delta_2, T)$ contains solutions of height j or less that do not include op' or $T\,(\Delta_2)$ contains no solutions of height j or less.

Proof Sketch: *Proof is much like that of Theorem 4.3. Assume that all solutions in $f(\Delta_2, T)$ contain op'. If $T\,(\Delta_2)$ contains solution $\langle op_1, \ldots op_k \rangle$, then the mapping developed in Theorem 3.3 exhibits a corresponding solution in $f(\Delta_2, T)$. Again, this construction insures that, since this solution in $f(\Delta_2, T)$ contains op', $\langle op_1, \ldots op_k \rangle$ must contain some $op \in O$. The operator transposition lemma can be used to transform $op_k \ldots op_1 \Delta_2$ into a solution in $T\,(op\ \Delta_1)$.*

Theorems 4.4 and 4.5 are more general than they need to be to justify cooperative search pruning under $\mathcal{A}$. They allow for the possibility that infeasible choice messages may pertain to open branches that are not leftmost. Since abstraction does not directly affect the branch selection order, this degree of generality is unnecessary for $\mathcal{A}$ alone. It does, however, demonstrate the compatibility of cooperative search pruning under $\mathcal{A}$ and $\mathcal{P}$. Whenever an infeasible-choice message is transmitted, rejected operations are described in the context of the original search space. It is the responsibility of the sending process to reconcile its transformed search space with the original. Similarly, the treatment of an incoming infeasible-choice message depends only on the transformation used by the receiving process. As a result, sibling naggers may employ different classes of transformations and may still freely exchange infeasible-choice information with the master and each other.

Chapter 5

First-Order Search Refinements

DALI's basic model-elimination search procedure has been refined in several respects. These refinements are intended to reduce search in the basic theorem proving procedure and to increase DALI's effectiveness on typical problems. In principal, refinement of the serial search procedure is orthogonal to nagging. However, any improvements to the basic search engine will benefit both master and nagger. As a result, naggers may be able to explore their transformed problems more effectively and, thereby, to prune the master's search more quickly. Indeed, if the M:N-queens results of Chapter 1 are representative, the bias introduced by refinement of the search procedure may make nagging even more effective.

Unfortunately, the peaceful coexistence of multiple sources of search reduction is not automatic. Unconstrained modifications of the search procedure might jeopardize some of the properties enjoyed by nagging and its extensions. To retain completeness, some mutual accommodation between nagging and serial search-reduction schemes is often required. One benefit of this is that there is some potential for cooperation between nagging and these serial search-reduction mechanisms. Exploiting this potential can permit DALI's serial and parallel components to each work more effectively.

5.1 Intelligent Backtracking

A completely naive search procedure must be prepared to examine every node in the tree until a solution is found. Under depth-first search, this implies a policy of *chronological backtracking*. Each time the search reaches a leaf node, n, without finding a solution, it backtracks to the most proximate ancestor of n for which $T\left(p^i(n)\right)$ has not yet been completely explored. In general, this meticulous traversal of the search tree is necessary because the search procedure has no advance information

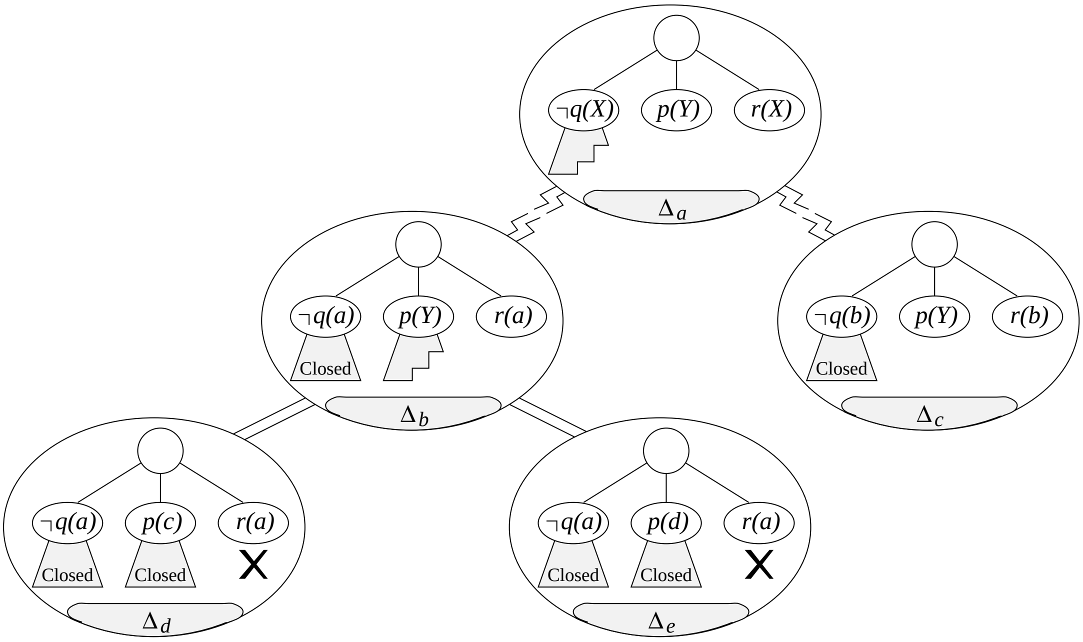


Figure 5.1: Intelligent backtracking scenario. The rightmost branches of Δ_d and Δ_e are guaranteed to have the same labeling. Because this branch cannot be closed in Δ_d, there is no need to explore Δ_e since it would fail for the same reason.

about where solutions might lie. Leaving a subtree before all of its nodes are examined could leave a solution undiscovered. For many individual problems, however, the search tree exhibits structure and inter-node relationships that can be exploited to safely avoid exploring some parts of the space. In particular, adjacent nodes in a model elimination search tree are related by various inference operations. In some cases, the failure to find a solution at one leaf node might implicate other, nearby nodes as certain failures. Information about these guaranteed failures can permit the search to evacuate some subtrees before they have been completely explored. A search can sometimes backtrack further than simple, chronological backtracking would allow. Search-reduction mechanisms of this type are classified as *intelligent backtracking*.

Figure 5.1 illustrates an opportunity for intelligent backtracking in model elimination. In node Δ_d the rightmost branch, the only open branch, has no applicable inference operations. Once the search reaches Δ_d, it must backtrack. Since Δ_d has a sibling Δ_e that has not yet been visited, chronological backtracking would return to node Δ_b, the parent of Δ_d and Δ_e. Nodes Δ_d and Δ_e, however, share the same rightmost branch. Since this branch could not be closed from Δ_d, it will be similarly unyielding in Δ_e. This inevitable failure at Δ_e is evident as soon as the search backtracks to Δ_b since Δ_b shares the same troublesome, rightmost branch with Δ_d and Δ_e. Since the labeling of this branch is ground in Δ_b, it will be the same in any tableau derived from Δ_b. Noticing this, a search procedure would be justified in backtracking past Δ_b and simply leaving node Δ_e unexplored; if the rightmost branch cannot be closed in Δ_d, it cannot be closed in any tableau derived from Δ_b. In fact, it would be safe to backtrack as far as necessary to change the labeling of this rightmost branch. In Figure 5.1, the search may backtrack all the way to Δ_a, skipping any unexplored nodes in the rightmost subtree of this partial search tree.

DALI features an intelligent backtracking component that is sensitive to the variable binding structure in the tableau. It is a model-elimination analog to many similar schemes developed for Prolog [Bru78,CD85,KL86,KL87]. DALI's scheme works by maintaining marks at various choice point frames. Whenever the exploration of a subtree is completed, some choice points may be marked. When search reaches a node Δ for which all applicable reductions or extensions have been considered, the following marking procedure is performed:

Definition 5.1 (Marking Procedure) *If branch β is the selected target of inference operations from Δ, the set of marked nodes for Δ and β is denoted by $\psi(\Delta, \beta)$. The set $\psi(\Delta, \beta)$ contains $p^i(\Delta)$ if any of the following are true:*

- *Branch β is a branch in $p^{i-1}(\Delta)$ but not in $p^i(\Delta)$.*
- *Branch β is a branch in $p^i(\Delta)$, but the labeling of β in $p^{i-1}(\Delta)$ is not a syntactic variant of its labeling in $p^i(\Delta)$.*
- *Node Δ is a solution.*

Upon backtracking, the search returns to the nearest ancestor node with a marked choice point. Whenever backtracking skips an unmarked choice point, the marking procedure is not applied at that node.

Informally, a mark means that the decision made at a choice point might be the reason a proof could not be reached. The mark at Δ is cleared whenever the search enters some $T(op\Delta)$. The intuition being that when *op* is first applied there is not yet a reason to suspect it as an incorrect choice. Alternatives to *op* are only considered if the search of $T(op\Delta)$ gives cause to doubt *op* by placing a mark at Δ.

Intelligent backtracking schemes permit the search to exit some subtrees before they have been exhaustively explored. In practice, the mark at a choice point prevents the search from leaving a subtree of T while it contains undiscovered solutions. A marking scheme does not need to guarantee that Δ will always be marked if $T(\Delta)$ contains solutions; it simply must insure that the mark is there whenever the search has to choose whether or not to backtrack out of $T(\Delta)$. If $T(\Delta)$ contains a solution, it is sufficient that Δ be marked each time the search emerges from some $T(op\Delta)$. Thus, the set of marks actually required for completeness depends not only on the locations of solutions in T but also on the part of T currently being explored. The set of *sufficient marks* relates the solutions in T to the choice points that must be marked within each subtree of T. The sufficient marks account not only for those choice points that are marked while exploring a given subtree, but also for those that will be marked elsewhere in the search space.

Definition 5.2 (Sufficient Marks) *For node Δ, the set of sufficient marks is denoted by $\psi_{\Downarrow}(\Delta)$. If $T(\Delta)$ contains solutions, then $\psi_{\Downarrow}(\Delta) = \{p(\Delta)\}$. Otherwise, $\psi_{\Downarrow}(\Delta)$ is the set satisfying the following:*

1. *Let $\langle op_1, \ldots op_k\rangle$ be the partial proof for which $\Delta = op_k \ldots op_1\Delta_\emptyset$. For $j \in \{0, \ldots k-1\}$ let $\langle i_1, \ldots i_{j'}\rangle$ be the subsequence of $\langle 1, \ldots j\rangle$ that includes i if and only if $op_{i-1} \ldots op_1\Delta_\emptyset \notin \psi_{\Downarrow}(\Delta)$. Node $op_j \ldots op_1\Delta_\emptyset$ is in $\psi_{\Downarrow}(\Delta)$ if $T\left(op_{i_{j'}} \ldots op_{i_1}\Delta_\emptyset\right)$ contains solutions but $T\left(op_{j+1}op_{i_{j'}} \ldots op_{i_1}\Delta_\emptyset\right)$ contains none.*
2. *Let j and k be positive integers and let β be the selected branch from $p^k(\Delta)$. If $T\left(p^j(p^k(\Delta))\right)$ contains no solutions, while $p^j(p^k(\Delta)) \in \psi_{\Downarrow}\left(p^k(\Delta)\right)$ and $p^j(p^k(\Delta)) \in \psi\left(p^k(\Delta), \beta\right)$, then then $p^k(\Delta)$ is in $\psi_{\Downarrow}(\Delta)$.*

For any tableau Δ, the set of sufficient marks is well defined since membership of some Δ' in $\psi_{\Downarrow}(\Delta)$ depends only on the inference operations deriving Δ and the ancestors of Δ' that are in $\psi_{\Downarrow}(\Delta)$. The sets of sufficient marks focus on the differences between a non-solution node in T and the nearest solution. They insure that each solution is found even when intelligent backtracking permits part of the search space to be ignored. Solution Δ_s will not be overlooked so long as, for every tableau Δ, some member of $\psi_{\Downarrow}(\Delta)$ is marked when the search exits $T(\Delta)$. DALI's marking scheme guarantees that each subtree meets its sufficient mark requirement. Naturally, it is desirable that marking favor the choice points of distant ancestors rather than near ones. This lets intelligent backtracking skip more choice points and achieves greater search reduction. Ideally, only those nodes along the path to a solution would be marked. In effect, this would lead the search directly to each proof. In practice, DALI's marking policy cannot demonstrate this kind of omniscience, but, even with this approximate mechanism for avoiding futile subtrees, intelligent backtracking can greatly reduce search.

The definition of $\psi_{\Downarrow}$ does reveal an opportunity for more aggressive search pruning in one special case. If node Δ is a non-solution leaf and β is the selected branch at Δ, backtracking from Δ may safely skip any choice points that are not

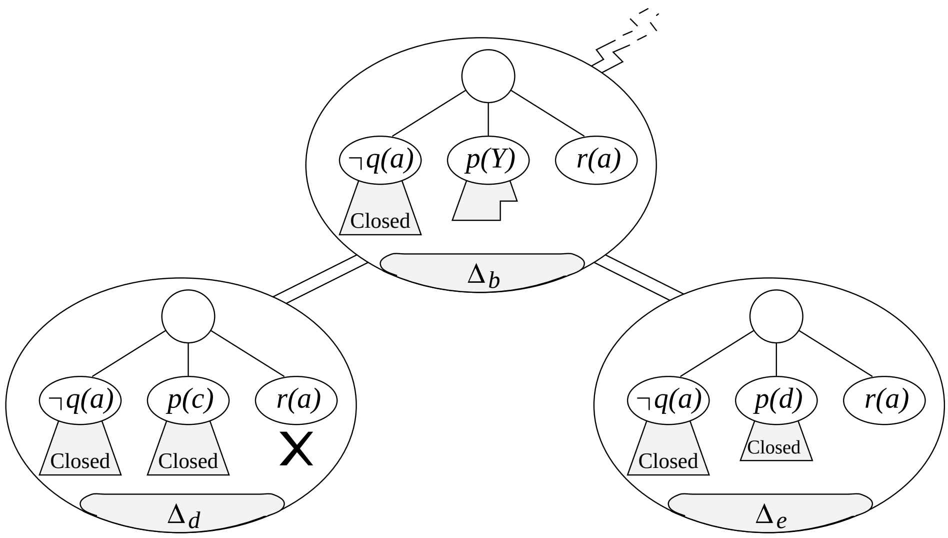


Figure 5.2: Interaction of depth metric and intelligent backtracking. If number of inferences is used to bound search depth, changing the size of a subproof for the middle branch may influence the ability to close rightmost branch even when its labeling is unchanged.

in $\psi(\Delta, \beta)$. This is because, if $T(p(\Delta))$ does not contain a solution, Definition 5.2 insures that $\psi_{\Downarrow}(p(\Delta)) \subseteq \psi_{\Downarrow}(\Delta)$. In general, $\psi(\Delta, \beta)$ must intersect $\psi_{\Downarrow}(p^i(\Delta))$ as long as $p^j(\Delta) \notin \psi(\Delta, \beta)$ for all $1 \leq j \leq i$. Thus, when a leaf is reached, DALI can safely skip over any ancestors not directly marked on behalf of that leaf. Any node skipped in this manner must have a member of its set of sufficient marks covered when the marking procedure is done at the leaf.

Compatibility with intelligent backtracking is the principal reason A-literal depth is DALI's standard basis for iterative deepening [Sti89]. Under A-literal depth, the potential to close a particular branch at an iteration does not depend on the specifics of how other branches are closed; it only depends on the labeling of the branch itself. This property is crucial to DALI's intelligent backtracking scheme. If the size of a tableau rather than its height is used to bound search depth, the ability to close a branch within the current depth bound may also depend on the size of the subproofs of other branches. Figure 5.2 illustrates this. Closing the rightmost branch of Δ_d might fail not simply because there are no applicable inference operations but because all such operations would exceed the depth bound. A smaller proof of the middle branch, $p(Y)$, might permit the rightmost branch to be successfully closed.

Even under A-literal depth, the use of intelligent backtracking presents complications when combined with nagging. Consider the situation presented in Figure 5.3.

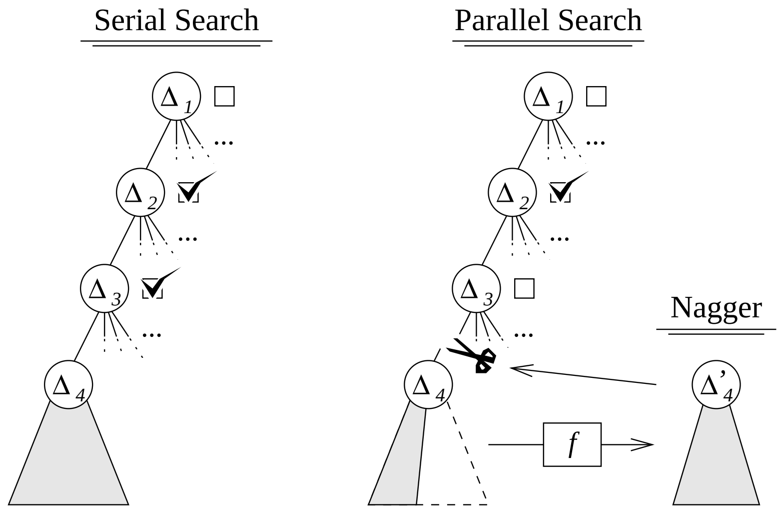


Figure 5.3: Interference of nagging with intelligent backtracking. Nagging permits the master to evacuate a subtree prematurely. This may prevent intelligent backtracking from placing all of the marks necessary while exploring that subtree.

Under the serial search, exploring $T(\Delta_4)$ causes the choice points at Δ_2 and Δ_3 to be marked. Nagging may, however, force the master to leave $T(\Delta_4)$ before it has been completely explored. The master may not reach the failure in $T(\Delta_4)$ that prompts marking of the choice point frame at Δ_3. As a result, the master may fail to consider other tableaux derived from Δ_3. If solutions lie within $T(\Delta_3)$ then nagging-induced search pruning may cause them to be missed. Performing nagging without concern for its interaction with intelligent backtracking compromises search completeness. This is an example of why the nagging search completeness results of Chapter 2 are dependent on certain properties (*e.g.* punctilious or myopic) of the search procedure. By itself, DALI with its intelligent-backtracking component executes a complete search. However, when nagging is added, completeness is lost.

The most obvious means of avoiding this interaction would be to forbid the use of intelligent backtracking in combination with nagging. A policy this exclusive is clearly unsatisfactory since both mechanisms have the potential to greatly reduce search. As a less extreme compromise, intelligent backtracking could be disabled only where nagging actually produced search pruning. When nagging prunes $T(\Delta_4)$ in Figure 5.3, all nodes $p^i(\Delta_4), i > 0$ could be marked. This employs the safe assumption that exploring $T(\Delta_4)$ in its entirety could mark, at most, all ancestors of Δ_4. Of course, this would forfeit the search reduction intelligent backtracking might otherwise provide.

Fortunately, for functions in $\mathcal{P}$ and $\mathcal{A}$, there is a more surgical means of updating backtracking marks while retaining completeness. The nagging process must exhaust some $f(\Delta, T)$ as a prerequisite to pruning the master's search. In building $f(\Delta, T)$, the nagger first reconstructs Δ by following the same derivation path as the master. The nagger does not have to worry about the search space outside $T(\Delta)$, but this reconstruction automatically equips it with a copy of the master's choice-point frames for Δ and all its ancestors. While searching in $f(\Delta, T)$, the nagger may be permitted to mark these choice-point frames according to a variant of the usual marking procedure. These marks will never affect the nagger's backtracking behavior, but they may be of use to the master. If the nagger succeeds in pruning Δ, any nodes in $\{p^i(\Delta) \mid i > 0\}$ that it has marked are then marked in the master's search. In fact, marks made during the master's partial search of $T(\Delta)$ are superseded by the nagger's marks and may be safely erased.

In general, the marks placed during search represent a conservative estimate of the choices that must be reconsidered. When the master explores a subtree, it emerges with a superset of the marks it needs to insure search completeness. As the nagger explores its transformed search tree, it develops its own pessimistic appraisal of what choices should be reconsidered. When nagging prunes some $T(\Delta)$, the master's search is interrupted prematurely and the entire set of marks that would have been made while exploring $T(\Delta)$ is not available. In this case, DALI simply adopts the nagger's estimate. Of course, to be correct the nagger's marks for $f(\Delta, T)$ must satisfy the same requirements as the master's marks for $T(\Delta)$. Were the master permitted to complete its search of $T(\Delta)$, it would emerge with some mark in the set $\psi_{\Downarrow}(\Delta)$. The marks placed on ancestors of Δ during the nagger's search of $f(\Delta, T)$ must also contain an element of $\psi_{\Downarrow}(\Delta)$.

The nodes of $f(\Delta, T)$ are not related to Δ and its ancestors through inference operations alone. In order for the nagger's backtracking information to be relevant to the master's search tree, the rules in the marking procedure must be adapted so that they apply across transformation. This is simplified by the fact that, for both $\mathcal{P}$ and $\mathcal{A}$, any branch in the root of $f(\Delta, T)$ has a representative in Δ. Consequently, the amended marking rules are virtually a restatement of the original in a new context. Let Δ' be some node in $f(\Delta, T)$, and let branch β be the selected target of inference operations from Δ'. The extended set of marks for Δ' is denoted by $\psi'(\Delta', \beta)$. This set includes all the nodes of $f(\Delta, T)$ that are contained in $\psi(\Delta', \beta)$. It also contains any ancestors of Δ that satisfy the following:

- If branch β is a branch of $p^{i-1}(\Delta)$ but not of $p^i(\Delta)$, then $p^i(\Delta) \in \psi'(\Delta', \beta)$.
- If β is a branch in $p^i(\Delta)$ and the labeling of β in $p^{i-1}(\Delta)$ is not a syntactic variant of its labeling in $p^i(\Delta)$, then $p^i(\Delta) \in \psi'(\Delta', \beta)$.

- If Δ' is a solution, then all ancestors of Δ are in $\psi'(\Delta', \beta)$.

DALI provides for the exchange of marks between master and nagger by associating a time stamp with each marked choice point. As T is explored, each attempted inference is counted. This count serves as the token with which choice points are marked and gives DALI some ability to determine where in T each mark originated. When nagging, these time stamps can be used to determine when one mark should take precedence over another. Once a choice point is marked with a token, it cannot be subsequently overwritten with a newer token until its mark is cleared. Each choice point records the earliest place in the search at which it was marked. When the master issues a problem message for Δ, it includes a record of all ancestors of Δ that were marked before the search of $T(\Delta)$ began. As the nagger explores some $f(\Delta, T)$, it operates under the amended marking policy and may place additional marks on these ancestors of Δ. A report of these marks is included with each prune message issued by the nagger. If the prune message arrives before the master has completed $T(\Delta)$, then $T(\Delta)$ is pruned as usual, and all of the master's marks are cleared and replaced with those reported by the nagger. Any discrepancy between the master's marks and the nagger's report results from the different failures encountered in an incomplete search of $T(\Delta)$ and exhaustive search of $f(\Delta, T)$. The nagger's report includes both those marks made by the master before searching $T(\Delta)$ and any marks made by the nagger while exploring $f(\Delta, T)$. Theorem 5.1 demonstrates that search completeness is retained under this policy.

Theorem 5.1 *For $f \in \mathcal{P}$ or $f \in \mathcal{A}$, the ancestors of Δ marked while exploring $f(\Delta, T)$ include an element of $\psi_{\Downarrow}(\Delta)$.*

Proof Sketch: *Proof depends on extending the definition of sufficient marks across problem transformation. This generalization of the sufficient marks classifies tableaux in T and $f(\Delta, T)$ as either dirty or clean. An induction from the leaves of $f(\Delta, T)$ to its root shows that each subtree in $f(\Delta, T)$ must mark some dirty node. The dirty ancestors of Δ are exactly the sufficient marks for $T(\Delta)$.*

The definition of sufficient marks also allows natural treatment of infeasible-choice messages. As with the prune message, the infeasible-choice message permits the master to neglect some portions of the search space. Again, care must be taken to insure that omitted search does not leave the sufficient marks unsatisfied. The situation here is complicated by the fact that a single infeasible-choice message from $f(\Delta, T)$ may permit many subtrees within $T(\Delta)$ to be pruned. Although the nagger has access to the choice point frames leading to Δ, it cannot reasonably supply markings for the choice points within $T(\Delta)$ along the path to every pruned, infeasible subtree. Fortunately, the nagger's marking of Δ and its ancestors is sufficient to maintain search completeness. With each infeasible choice message for $f(\Delta, T)$, the nagger includes a report of the current marks on the ancestors of Δ. The master tacitly assumes that node Δ should, itself, be marked. This is a necessary assumption since the nagger's amended marking procedure does not provide for marks on Δ. It is also a fairly risk-free assumption; if the root of $f(\Delta, T)$, the nagger's representative of Δ, is not marked, any infeasible-choice message for Δ will be immediately followed by a prune message for Δ.

Theorem 5.2 *Let $f \in \mathcal{P} \cup \mathcal{A}$ and let Δ_r be the root of some $f(\Delta, T)$. Let op' be an inference operation applicable to Δ_r. If a nagger exhausts $T(op'\Delta_r)$ and generates an infeasible-choice message implicating operation op in the original search space, then, for any $\bar{\Delta} \in T(\Delta)$, if $\psi_{\Downarrow}\left(op\bar{\Delta}\right)$ is defined and nonempty, it must contain one of the following:*

- *One of the ancestors of Δ marked during the search of $T(op'\Delta_r)$*
- Δ
- $\bar{\Delta}$

Proof Sketch: *Because of Theorem 5.1, it is possible to relate the marks made within $T(op'\Delta_r)$ to the sufficient marks for $T(op\Delta)$. These marks are, in turn, related to $\psi_{\Downarrow}\left(op\bar{\Delta}\right)$. It can be shown that $\psi_{\Downarrow}\left(op\bar{\Delta}\right)$ must include $\psi_{\Downarrow}(op\Delta)$ or $\left\{\bar{\Delta}\right\}$.*

Theorem 5.2 is sufficient to guarantee completeness in the presence of infeasible-choice messages. Marks on the ancestors of Δ are supplied by the nagger, while the master conservatively assumes a mark for Δ. Although the tableau $\bar{\Delta}$, identified in the theorem, is never explicitly marked, normal search operation automatically treats it as if it was. A mark on $\bar{\Delta}$ simply prevents the search from leaving $T(\Delta)$ while $\bar{\Delta}$ has unexplored children. If the master is exploring $T\left(op\bar{\Delta}\right)$ when it receives an infeasible-choice message for op, it is permitted to backtrack only as far as $\bar{\Delta}$. If the master reaches $op\bar{\Delta}$ after the receipt of a relevant infeasible-choice message, it simply skips $T\left(op\bar{\Delta}\right)$ and moves on to the next child of $\bar{\Delta}$. This behavior at $\bar{\Delta}$ is essentially the same as it would be if $\bar{\Delta}$ had been marked during a search of $T\left(op\bar{\Delta}\right)$; the master moves on to the next child of $\bar{\Delta}$.

Although DALI's intelligent backtracking scheme has the potential to significantly reduce search, it compromises myopia. Without myopia, the Solution Ordering and Non-Increasing Search properties are no longer guaranteed. Because of intelligent backtracking, a failure to close tableaux in one subtree of the search may permit later subtrees to be discarded. If nagging preempts the search in one subtree, it may change the marking of nodes outside that subtree and may interfere with intelligent backtracking in subsequent subtrees. Even with the precaution of importing naggers' marks, nagging in the presence of intelligent backtracking may actually cause the master to explore a larger portion of T than it would otherwise. Although this potential for adverse interaction exists, Chapter 6 demonstrates that nagging and intelligent backtracking can combine quite well in practice.

5.2 Structural Refinements

As described in Chapter 3, model elimination is refutation complete; for any set of contradictory clauses, there is a model-elimination proof. In fact, for some theories there may be several or even an infinite number of proofs. Some of these may correspond to different deductive arguments but many of them typically represent equivalent proofs of the same things. The most serious disadvantage of this redundancy in the proof calculus is that it leads to redundant lines of reasoning during search. For each redundant proof, there may be still more redundant partial proofs. In the interests of efficiency, it is usually desirable to prohibit all but one of these redundant partial proofs. If this causes a proof to be overlooked, then an equivalent proof must exist elsewhere. If the discarded line of reasoning does not lead to a solution, then some amount of futile search will be avoided.

For model elimination, there are several ways to identify redundant tableaux based on their structure. *Structural refinements* promise to eliminate some redundant partial proofs without reducing deductive power [Sti88]. Tableaux that satisfy any of the following conditions represent redundant lines of reasoning:

- Tableau Δ is redundant if it contains a branch with two identically labeled nodes.
- Tableau Δ is redundant if it contains a closed branch with two complementary, non-leaf nodes.
- Tableau Δ is redundant if it contains a closed branch with a non-leaf node that is complementary to an instance of a unit clause.

During search, any tableau satisfying one of these conditions may be safely discarded. This does not guarantee that the discarded tableau cannot lead to a complete proof. These restrictions simply insure that, if some tableau Δ is discarded and $T(\Delta)$ contains a solution, there will be an equivalent solution elsewhere in the search which is not barred by these structural constraints.

Since these refinements may discard a node that leads to a solution, they may actually increase the amount of search required to find a proof. A search procedure may discard many legitimate avenues of proof before reaching one that survives these conditions. If all solution paths in the left portion of T are deemed redundant, it might be more expedient to simply pursue potential proofs as they are encountered, regardless of their redundancy. Naturally, there is no reasonable way to know when these refinements will reject a line of reasoning that leads to a proof.

Checking each derived tableau against these conditions would entail significant overhead for the search procedure and would represent a pronounced deviation the WAM model. The WAM is designed to make individual inference operations very efficient. Typically, each inference requires only a small number of relatively local changes to the tableau. Checking each of these conditions in their full generality would require global examination of the tableau at each search node. To preserve the spirit of the WAM along with a high inference rate, DALI's structural refinements are based on weaker versions of these constraints that have similar search-reducing power [Sti88]. Let Δ be a node in the search tree and let β be the branch chosen by the selection function on Δ:

1. If the label on $\mathit{leaf}(\beta)$ is identical to the label of some other node in β, then Δ may be discarded and backtracking initiated.

2. If the label on $\mathit{leaf}(\beta)$ is complementary to some other node $n \in \beta$, then $\mathit{Red}_{\beta,n}\,\Delta$ is the only child of Δ that needs to be explored.

3. If the label on $\mathit{leaf}(\beta)$ is complementary to an instance of some unit clause $\{l\}$, then $\mathit{Ext}_{\beta,\{l\},l}\Delta$ is the only child of Δ that needs to be explored.

Since these weaker conditions all pertain to only a single branch in each tableau, they can be checked with fairly little computation before the tableau is expanded. DALI optionally supplements this policy with the following additional constraint:

4 If $\langle op_1, \ldots op_k \rangle$ is a subproof for β and the label on $\mathit{leaf}(\beta)$ in $op_k \ldots op_1\Delta$ is identical to one of its ancestors, then $op_k \ldots op_1\Delta$ may be discarded and backtracking initiated.

This condition is employed because it, like its proactive counterpart, is checked only once for each tableau. As subtrees of the tableau are closed, this constraint may be enforced. In some cases, a branch will contain unifiable but non-identical labels when it is open. A subproof for this branch may bind variables so that these unifiable labels match. Checking for identical labels at the end of a subproof as well as its beginning provides a closer approximation of the unrestricted structural constraints. Enforcing the constraint at the end of a subproof capitalizes on the fact that any variables of the tableau that are bound within a subproof for branch β must occur somewhere in β.

Adding these structural refinements to DALI only requires enforcing the first and last constraints. The other two are, in part, maintained by intelligent backtracking and the standard search order. Ordinarily, for search node Δ and branch β in Δ, reductions of β are attempted before any extensions. If β already contains a node, n, that is complementary to its leaf, then $Red_{\beta,n}$ will bind no variables. As a result, it will not change any labels on the tableau, and, since $Red_{\beta,n}$ is not an extension, the choice point at Δ will never be marked under the intelligent backtracking marking procedure. Once $T(Red_{\beta,n}\Delta)$ has been explored, backtracking will skip any remaining nodes in $T(\Delta)$. Similarly, extensions by unit clauses are attempted before any other extensions. If $leaf(\beta)$ is already an instance of $\neg l$ for some clause $\{l\}$ then $Ext_{\beta,\{l\},l}$ will also bind no variables in Δ. Even though a new branch is created by extension of Δ, that new branch is automatically closed, and, again, Δ will not be marked while searching in $T\left(Ext_{\beta,\{l\},l}\Delta\right)$. With no marks on Δ, once $T\left(Ext_{\beta,\{l\},l}\Delta\right)$ has been explored, backtracking will skip any remaining nodes in $T(\Delta)$.

There are some situations in which these approximations of constraints 2 and 3 are not as effective as the original. The standard search ordering does not guarantee that these operations will be the first ones investigated; it only helps to insure that they will be considered early. If the reduction or extension implicated under these conditions is not the first applicable operation, other, redundant paths of reasoning may be followed before a tableau's one, non-redundant child is reached. In spite of these flaws, the approximation provided by intelligent backtracking is attractive, particularly because it requires no additional computational machinery and imposes no additional overhead.

Constraints 1 and 4, called the *identical-ancestor refinement*, are implemented by simply checking their preconditions at the appropriate times and backtracking whenever it is warranted. Whenever backtracking occurs, the marking scheme of intelligent backtracking must be accommodated. The marks must represent an assigning of blame for an abandoned search path. When the identical-ancestor refinement discards a path, marks have to be placed on choices that, if made differently, might permit the proof to succeed or at least pass the identical-ancestor conditions. If tableau node n_1 and one of its ancestors, n_2, are found to have identical labels in Δ, a mark is placed on $p^i(\Delta)$ under the following conditions:

- Node n_1 is created by an extension of $p^i(\Delta)$.

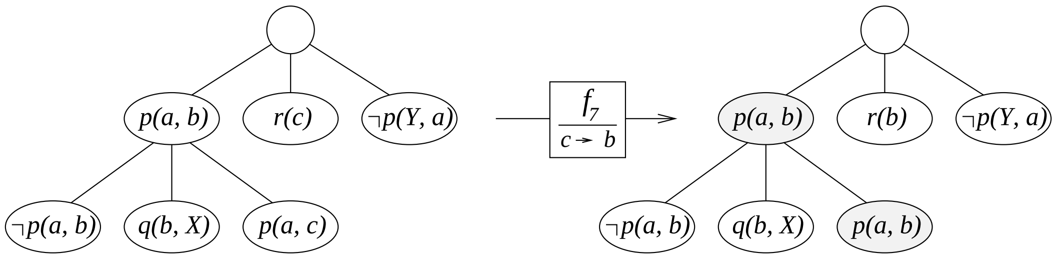


Figure 5.4: Conflict between structural refinement and naive abstraction. Mapping occurrences of symbols b and c to the symbol b makes the previously distinct labels $p(a,b)$ and $p(a,c)$ identical. No proof derived from this abstracted tableau can survive the identical-ancestor refinement.

- Both n_1 and n_2 exist in $p^i(\Delta)$ and their labeling in $p^{i-1}(\Delta)$ is not a syntactic variant of their labeling in $p^i(\Delta)$.

Theorems 5.3 and 5.4 demonstrate that nagging search remains complete even when these identical-ancestor refinements are enforced on both master and nagging processes. This holds for nagging under both $\mathcal{P}$ and $\mathcal{A}$ and is one of the reasons transformation via $\mathcal{A}$ is more surgical than the simple constant replacement scheme mandated by the function f_7 of Chapter 3. Continued membership in $\mathcal{F}$ is not automatic in the presence of structural refinements such as these. Figure 5.4 demonstrates the conflict between the identical-ancestor refinement and the abstraction scheme of f_7. Under transformation, the constant symbols b and c are identified. All occurrences of c are replaced by b. This makes the labels $p(a,b)$ and $p(a,c)$ identical and, since they lie on a common branch, ultimately classifies every derivation in the nagger's search tree as redundant. After exhausting its search space without finding any solutions, the nagger may prune the master's search. However, since the labels $p(a,b)$ and $p(a,c)$ differ in the original tableau, there may be solutions in the master's subtree that are not redundant and that the master would ordinarily reach. Transformations in $\mathcal{A}$ avoid this problem since they never make differing symbols identical; they only make them unifiable. If b and c are equivalent under abstraction, $p(a,b)$ and $p(a,c)$ may be replaced with $p(f_{[a]}(a), f_{[b]}(V_1))$ and $p(f_{[a]}(a), f_{[b]}(V_2))$ where V_1 and V_2 are unique variables. Thus, $p(f_{[a]}(a), f_{[b]}(V_1))$ and $p(f_{[a]}(a), f_{[b]}(V_2))$ can only match if V_1 and V_2 are subsequently bound to the same term.

Theorem 5.3 *Let $f \in \mathcal{P}$ and let Δ be a node of T. If $T(\Delta)$ contains a solution that is derivable in the presence of the identical-ancestor refinement, then $f(\Delta, T)$ also contains a solution that survives the pruning of the identical-ancestor refinement.*

Theorem 5.4 *Let $f \in \mathcal{A}$ and let Δ be a node of T. If $T(\Delta)$ contains a solution that is derivable in the presence of the identical-ancestor refinement, then $f(\Delta, T)$ also contains a solution that is derivable in the presence of the identical-ancestor refinement.*

Proof Sketch: *For both of these theorems, proof depends on first showing that if $T(\Delta)$ contains a closed tableau that does not violate the identical-ancestor constraints, then $f(\Delta, T)$ also contains a closed tableau that does not violate the identical-ancestor constraint. Then it can be shown that, if no branch in solution Δ_s contains two identically labeled nodes, then no tableau on the derivation path for Δ_s has a branch with identically labeled nodes.*

These structural refinements could be seen as inconsistent with a punctilious search procedure. A punctilious search is forbidden from discarding any line of reasoning that leads to a solution; the identical-ancestor refinement permits discarding of some solutions. This apparent conflict may be remedied by simply redefining what tableaux constitute solutions. Since the structural refinements operate uniformly regardless of where a node appears in the search space or when it is expanded, the punctilious property is satisfied if only those tableaux that survive the identical-ancestor constraints are considered proper solutions. Modulo this tighter constraint on solutions, DALI's search procedure remains punctilious and retains the nagging-related properties that this affords. In fact, by themselves these structural refinements simply reject suboptimal tableaux. They do not otherwise alter the search order, and their effects do not depend on which parts of T are explored or the order in which they are examined. As a result, they do not compromise myopia, and all the nagging properties contingent on myopia are preserved as well.

5.3 Subgoal Caching

Intelligent backtracking represents an effort to use information gleaned from failures in one part of the search tree to avoid search elsewhere. It exploits the fact that, in some situations, tableau branches in one node of T are certain to occur elsewhere in T. Caching exploits the same phenomenon. While intelligent backtracking identifies nearby nodes that are guaranteed to fail because they share features with the current tableau, caching 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 a bounded-overhead caching scheme developed for Horn-clause theories [SS93]. It differs from other notions of caching and lemmaizing [AS92] in its criteria for permitting the use of cached information and in its bounding of the cache size. Once the number of entries reaches the effective size limit, old entries must be discarded to make room for new ones. This provides a mechanism for controlling the memory consumed by the cache and the cost of looking for matching entries. The hope is that, under a reasonable cache replacement policy, the most useful entries will be retained while less useful entries are discarded.

In general, caching schemes of this type compromise myopia. 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.

One convenience of caching in a theorem proving environment is that, unlike a cached memory system, it is not usually necessary to insure cache consistency between parallel processes. When performing caching in combination with nagging, master and nagger do not need to worry about their caches containing different entries. Each may manage its cache independently, populating it with entries specific to its own experience. 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 at reducing its 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.

5.3.1 Success Caching

Each time an attempt is made to close some branch β in tableau Δ, an entry is placed in the cache. Whenever β is successfully closed, the cached entry indicates both that β can be closed and the labeling of β immediately after it is closed. If the search later encounters a tableau Δ' with an open branch β' labeled identically to the cached labeling for β, it is not necessary to search for a subproof of β'. This recycling of successful subproofs is called *success caching*, and avoiding search by exploiting a match in the success cache is called a *success-cache hit*. The cached entry for β is evidence that the desired subproof for β' exists. Furthermore, the subproof found for β must be weakly local for β' in Δ'. Thus, no alternative subproofs for β' need to be sought upon backtracking. As with a nagger's local subproof, a success-cache hit need not actually modify the tableau; branch β' may simply be assumed closed. The overhead of exploiting a success-cache entry can be made fairly low.

More generally, a cached success for branch β can be used even when β' is only an instance of β. In this case also, the subproof for β is weakly local for β' in Δ'. In contrast, although it is possible, it is not always beneficial to permit a cache hit when this instantiation is reversed. When β is an instance but not a syntactic variant of β', the subproof for β could be fashioned to close β', but it is not guaranteed to be weakly local. Alternative subproofs for β' may eventually need to be considered. As with nagger-discovered subproofs, permitting a cache hit that is not weakly local may force the search to either explore partial solutions that it would not ordinarily consider or to follow the same line of reasoning more than once.

For Horn-clause theories a less specific form of success caching can be employed without loss of correctness. Under a Horn-clause theory, the reduction operation is never necessary. Hence, the only feature of β that determines whether or not a subproof exists is the labeling of its leaf node. When a branch is successfully closed under such a theory, only the labeling of the leaf node has to be entered in the cache. More significantly, a success cache hit requires only that the leaf node of the new branch be an instance of the cached leaf for β. This decreases the amount of memory consumed by the cache, simplifies the process of looking up patterns and tends to permit each cache entry to be matched more often.

The advantages of matching only on leaves are so compelling, DALI's caching policy has been engineered to permit it even in non-Horn theories. This exacts a penalty with respect to which successful subproofs may be cached, but it yields a caching mechanism that is much more efficient and generally of greater utility. The only way a subproof for branch β can depend on labels other than the one on $leaf(\beta)$ is if it contains reductions with ancestors of $leaf(\beta)$. If a subproof contains no such reductions, it is guaranteed to be applicable to any branch β' with a matching leaf. Even if β' disagrees with β on nodes other than the leaves, these disparate labels have no relevance to the particular subproof discovered for β. The maximum scope of reductions within a subproof can be computed incrementally as branches of the tableau are closed. Each time a subproof for β is completed, the locality of its reductions is checked. If it contains no reduction to ancestors of $leaf(\beta)$, a success entry is made for the label of $leaf(\beta)$.

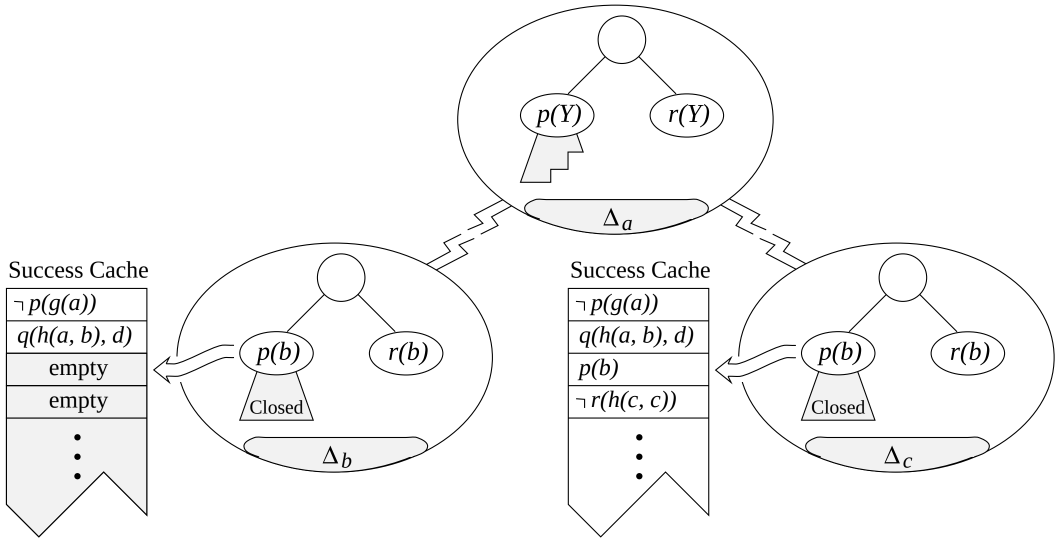


Figure 5.5: Potential for redundancy avoidance through the success cache. The subproof for the leftmost branch of Δ_c makes the same variable bindings as the subproof for this branch in Δ_b. The success cache can be used as a mechanism for detecting different subproofs that make the same variable bindings.

5.3.2 Redundancy Avoidance

The success cache is primarily a mechanism for avoiding the reconstruction of a subproof when an applicable, weakly local subproof is known to exist. A success-cache hit on branch β' eliminates search on behalf of β'. It frees the search procedure from deriving an initial subproof for β' and from considering alternative subproofs for β'.

There are opportunities to wrest additional search-reducing power from the success cache. Figure 5.5 demonstrates some of this potential. In node Δ_b, the leftmost branch is closed, and the variable Y is bound to b. Search in $T(\Delta_b)$ is directed toward building a subproof for the rightmost branch that is consistent with this binding of the variable Y. If an appropriate subproof cannot be found, the search will backtrack to consider alternative subproofs for the first branch. Unfortunately, there is no guarantee that these alternative subproofs will yield different variable bindings. A second subproof for $p(Y)$ cannot lead to a closed tableau if it makes the same variable bindings as the first one. As argued in Section 5.2, a redundant theory may admit many proofs for each fact it entails. This redundancy may lead to many, equivalent ways of closing each branch. When the search reaches a dead end, DALI's intelligent backtracking mechanism is designed to encourage prompt retraction of the offending bindings, but it does not prohibit eventual reinstatement of the same bindings.

In some cases, the success cache preserves a record of the variable bindings made within a subproof. DALI's success-cache *redundancy-avoidance* mechanism uses this record to detect the repetition of equivalent variable bindings. 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 implicitly recorded in this entry. In node Δ_c, the variable Y is again bound to b via a different subproof. Insertion of this new success into the cache finds $p(b)$ already present. If it was clear that the existing $p(b)$ entry had been made on behalf of the same branch, the search would be justified in abandoning a repeated attempt to close the $r(b)$ branch. To permit this, DALI temporarily retains information about which branches are responsible for a given success-cache entry. Each time branch β inserts a success-cache entry, β and that entry are associated. This association is established even if the success pattern for β was already present in the cache or associated with a different branch; each cache entry may have several associated branches and each branch may be associated with more than one cache entry. An association between a cache entry and a branch β persists until one of the following occurs:

- the cache entry is discarded by the cache replacement policy.
- the search backtracks far enough to retract the extension that created the node, $leaf(\beta)$.
- while β is open, the search selects some branch that is not a superset of β.

Each time a success pattern for β is inserted, the caching mechanism checks to see if that pattern is already in the cache and associated with β. If it is, the search immediately backtracks to find a different subproof for β. As a side-effect, DALI's cache insertion mechanism automatically detects when an inserted pattern is already present in the cache. Thus, this kind of redundancy avoidance is a natural part of the cache insertion procedure.

To be correct, this policy must insure that all variable bindings that are subject to change are represented in the label on the leaf node. If β is a branch of tableau Δ, then, as long as β is associated with a cache entry, all bindings made while β is open must be to variables appearing in $leaf(\beta)$. Conveniently, the restrictions already in place for success caching help to enforce this. If branch β is not closed, selection of any branch that does not contain β retracts all associations with β. The only way an operation on some branch $\beta' \supseteq \beta$ can affect the labeling of Δ is through the binding of variables in β; the only way it can bind variables outside

$leaf(\beta)$ is through reduction with some ancestor of $leaf(\beta)$. The standard success caching policy excludes any subproofs that contain reductions above $leaf(\beta)$. Thus, if a subproof of β is permitted to enter the cache, it must bind no variables in Δ apart from those contained in $leaf(\beta)$. Likewise, detection of redundant subproofs is performed under the same conditions as success caching. A subproof is rejected as redundant only when all of its variable bindings are accounted for in its leaf label.

Of course, a mechanism of this type could be less particular in its avoidance of redundant search paths. The exclusion of nonlocal reductions from the success cache leaves some redundant subproofs undetected. With a more specialized facility for recording variable bindings, it would be possible to tolerate these reductions as well as less orderly behavior from the selection function. There are also opportunities to relax the requirements for a branch's labeling being deemed redundant. If a subproof that labels β with $\langle l_1, \ldots l_k \rangle$ does not lead to a closed tableau, then any subproof labeling β with an instance of $\langle l_1, \ldots l_k \rangle$ will also preclude a complete proof. Indeed, the anti-lemmata used in SETHEO permit backtracking in many of these more general cases [LMG92]. These opportunities for more aggressive search reduction are unexploited in DALI because they necessitate more radical augmentation of the success cache. In its more limited form, the redundancy-avoidance mechanism is very consistent with the existing cache facility and represents little additional overhead. DALI uses the more conservative redundancy-avoidance scheme in an effort to balance search-reduction power and implementation efficiency in a manner consistent with the WAM architecture.

In isolation, the success cache does not interfere with other components of the prover, but some effort must be made to make redundancy avoidance compatible with other search reduction mechanisms. Like the structural refinements, redundancy avoidance permits some search paths to be discarded because, even if they lead to a solution, that solution can be found elsewhere in T. Here also, intelligent backtracking must be accommodated. When a search path is discarded prematurely, some set of choice points must be blamed for the failure to find a solution. If one subproof for β exhibits the same labeling of $leaf(\beta)$ as a previous subproof of β, backtracking should return to the most recent choice that is responsible for $leaf(\beta)$ and its labeling. When a redundant subproof for $\beta \subseteq \Delta$ is detected, a mark is placed on $p^i(\Delta)$ under the following conditions:

- Node $leaf(\beta)$ is created by an extension of $p^i(\Delta)$.
- Branch β exists in $p^i(\Delta)$ and its labeling in $p^{i-1}(\Delta)$ is not a syntactic variant of its labeling in $p^i(\Delta)$.

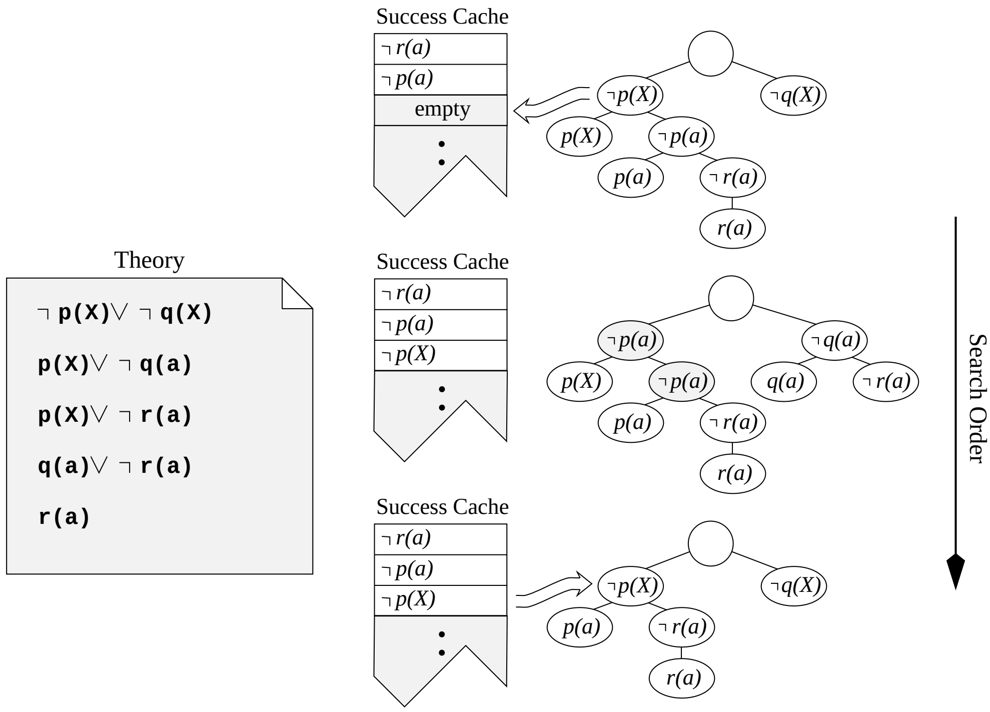


Figure 5.6: Conflict between identical-ancestor refinement and redundancy avoidance. Both mechanisms can reject tableaux that are on the path to a solution. Aggressive enforcement of the identical-ancestor refinement can cause these techniques to prohibit all solutions.

Some argument must be made to demonstrate the compatibility of redundancy avoidance and the identical-ancestor refinement. Both reject lines of reasoning when it is clear that some other line of reasoning is at least as promising. It would be unfortunate if, together, these two mechanisms rejected all representatives of a given proof. Figure 5.6 illustrates how this could occur if the identical-ancestor refinement was enforced in its full generality. Under this somewhat implausible Horn-clause theory the first subproof found for the $\neg p(X)$ branch is acceptable under the identical-ancestor condition and is cached. As the remaining branch is closed, however, the binding of X causes the existing left-hand subproof to violate the identical-ancestor condition. If this violation is detected, backtracking will permit an alternative subproof of the rightmost branch, but redundancy avoidance will reject this second subproof. DALI's less diligent enforcement of the identical-ancestor refinement avoids this undesirable interaction. Fundamentally, the redundancy-avoidance mechanism depends on the adequacy of variable bindings to capture all of a subproof's influence on the closing of the rest of the tableau. All subproofs that produce the same variable bindings should be operationally equivalent. This is violated by the liberal application of the identical-ancestor conditions shown in Figure 5.6. The two subproofs for the leftmost branch differ structurally but produce the same variable bindings. The success cache is insensitive to these structural differences, but they eventually decide whether or not an inference path will be permitted to reach a solution. A similar problem could occur if the depth metric was dependent on the number of inferences; the size of a subproof would determine the amount of search depth that remained for closing other branches. DALI avoids this type of interaction through adherence to a particular design philosophy. If a subproof for branch β in tableau Δ is cached, the only influence $T\,(\mathit{leaf}(\beta))$ is permitted to have on search in $T\,(\Delta)$ is through its binding of variables. As a consequence, all enforcement of the identical-ancestor conditions within a subproof must occur before the subproof is cached.

Theorem 5.5 verifies a form of search completeness in the serial case. Redundancy avoidance has the potential to discard solutions from T. Any claims of completeness must account for this potential omission of solutions. Theorem 5.5 promises *representative completeness*; for any solution discarded by the cache, there will be a different solution that shares structural properties with the first but is not discarded. The theorem also suggests a policy by which a similar form of completeness can be retained in the parallel case. The construction of some f ensures that $f\,(\Delta, T)$ contains a solution whenever $T\,(\Delta)$ does. For functions in $\mathcal{P}$ and $\mathcal{A}$ each transformed subproblem is, itself, a model elimination search space. If no branch-cache associations exist when the search of $f\,(\Delta, T)$ begins, theorem 5.5 provides that a solution will be found in $f\,(\Delta, T)$ whenever one exists in $T\,(\Delta)$. To capitalize on this, naggers are required to discard all associations between tableau branches and cache entries whenever they receive a problem message.

Theorem 5.5 (Representative Completeness) *Let Δ be a node in T such that there are no associations between the branches of Δ and the cache when the search of $T(\Delta)$ begins. If $T(\Delta)$ contains a solution Δ_s that is pruned by the redundancy-avoidance cache, then $T(\Delta)$ must contain a solution Δ'_s that is not pruned by the cache and is reached before Δ_s. Furthermore, Δ_s and Δ'_s must have the same number of first-level nodes with the same labels.*

Proof Sketch: *Proof is by induction. For any solution $\Delta_i \in T(\Delta)$ that is prohibited by the redundancy-avoidance cache, there must be a structurally similar solution Δ_{i+1} that occurs before Δ_i and such that the search comes at least as close to finding Δ_{i+1} as it does to Δ_i. Since $T(\Delta)$ is finite for depth-bounded search, this induction from Δ_s must terminate with a solution that is found.*

Even with Theorem 5.5, some nagging-related properties do not persist in the presence of redundancy avoidance. Because its effects are dependent on the contents of the cache, redundancy avoidance sacrifices myopia and jeopardizes properties that depend on it. Like the identical-ancestor refinement, redundancy avoidance can permit solution nodes to be discarded. This conflicts with the otherwise punctilious operation of the theorem prover. Unlike the identical-ancestor refinement, the particular solutions discarded here depend on the contents of the cache and, indirectly, on the search history. As a result, the technique used to retain the punctilious property for the identical-ancestor refinement, simply redefining the solution criteria, will not work with redundancy avoidance. Even without the punctilious property, however, there is a weaker form of completeness that is retained when nagging with redundancy avoidance. Theorem 5.5 promises that nagging will never directly prune solutions from T; the only way it can affect the solutions found by the master is by influencing the contents of the master's cache. Since nagging does not alter the way the master inserts cache entries or builds branch-cache associations, this change in the cache contents cannot compromise the set of solutions protected under representative completeness.

5.3.3 Failure Caching

Failure caching records branches for which no subproof can be found. If branch β in tableau Δ cannot be closed within the current depth limit, attempts to close an identically labeled branch elsewhere in the search will also fail unless the search procedure is willing to consider deeper subproofs. If failure caching indicates that

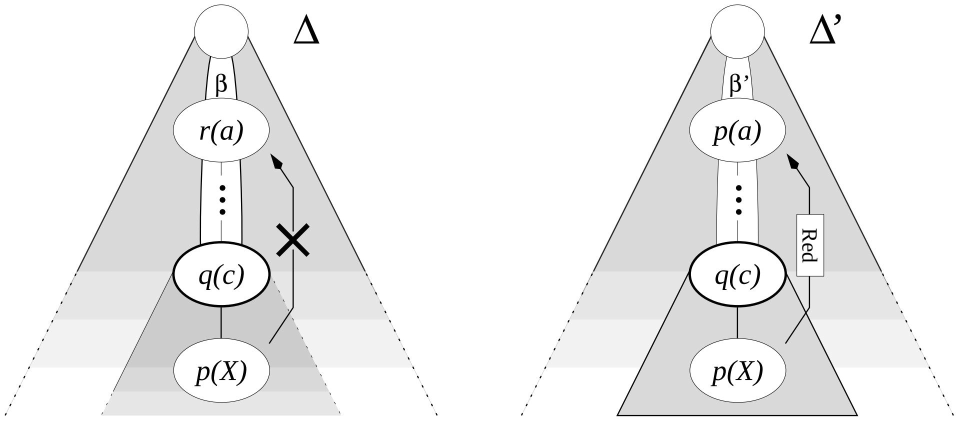


Figure 5.7: Inadequacy of leaf node for predicting branch failure. Branches β and β' have the same labels on their leaves, but different labels on non-leaf nodes permit different reduction operations. As a result, β' can be closed in the right-hand tableau while β cannot be closed in the left-hand tableau.

the current branch has no subproofs, the search may backtrack immediately. As with the success cache, a failure hit may be permitted even if the current branch is only an instance of the cached branch. If no subproofs can be found for branch β, then an attempt to close a more specific $\beta\,\theta$ will also be futile.

DALI's preference for caching only the leaf label of a branch is extended to its failure caching scheme. This contributes to an efficient implementation and even permits success and failure entries to be looked up at the same time. However, it ultimately restricts use of the failure cache to Horn-clause theories. If branch β cannot be closed, there may still be subproofs of some other branch β' even when the leaves of β and β' are identically labeled. As Figure 5.7 demonstrates, labels on non-leaf nodes of β' may permit reductions that were not possible in β. These reductions may, in turn, allow subproofs of β' that could not be constructed from β. Of course, this limitation can be remedied by caching the labeling of all nodes in a failed branch and matching on all labels during cache lookup. Although this would enable the use of failure caching in general first-order theories, the matching criteria under this policy would be much more restrictive. Requiring a match of all labels on a branch would significantly reduce the likelihood of a failure-cache hit. Consequently, DALI exploits failure caching only when matching on the leaf node is sufficient. Failure caching is simply disabled when non-Horn clauses are present.

The search reduction achieved through failure caching differs from that provided by redundancy avoidance. There are some cases in which the two techniques behave similarly, but, in general, they complement each other. This is demonstrated in

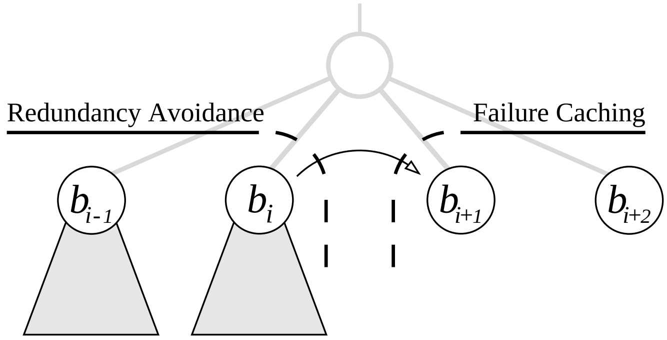


Figure 5.8: Comparison between redundancy avoidance and failure caching. Redundancy avoidance prunes the search after a branch has been successfully closed. Failure caching prunes search when an attempt to close a branch begins.

Figure 5.8. Here, redundancy avoidance prunes search when branch β_i is completed and failure caching prunes before work on closing β_{i+1} begins. Although these are simply different accounts of the same point in the search, redundancy avoidance is sensitive to the variables in $leaf(\beta_i)$ while failure caching is sensitive to variables in $leaf(\beta_{i+1})$. Redundancy avoidance at β_i can reject a search path that repeats variable bindings even when these bindings do not actually preclude a subproof for β_{i+1}. Likewise, failure caching can avoid repeated attempts to close β_{i+1} with the same labeling even when β_i is not solely responsible for instantiating the label on $leaf(\beta_{i+1})$. More generally, these two mechanisms differ in terms of their scope. Redundancy avoidance is fundamentally a local technique; when a cache entry is associated with branch β_i, that association only serves to eliminate search pertaining to β_i. Failure caching operates globally. A record of failure at one branch may help to avoid search for many different branches elsewhere in T. In some situations, results cached in one problem may even be used to curtail search in other, related problems [SS93].

Even when restricted to Horn-clause theories, some care must be taken in the creation and use of a failure-cache entry. A failure entry for branch β can be made only when it is known that no subproofs for β exist within the depth bound. Obtaining this kind of information about individual branches is not central to the normal operation of a theorem prover. DALI is responsible for closing the entire tableau not simply enumerating subproofs for each of its branches. In trying to close the entire tableau, some potential subproofs may not even be considered because they are inconsistent with closing other tableau branches. In the interests of efficiency, it is often desirable to exploit this global information about the tableau in order to avoid generating useless solutions for its components.

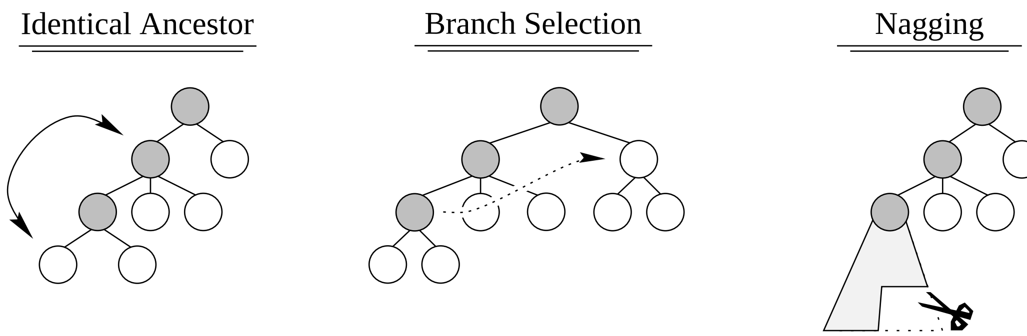


Figure 5.9: Conflict between failure caching and other search reduction mechanisms. Many of DALI's search-reduction mechanisms permit subproofs of a branch to be skipped. Failure caching must discriminate between branches that have no subproofs and those that might have subproofs that are rejected by other search-reduction mechanisms.

Many of DALI's search-reduction mechanisms make use of this kind of global problem information. These include redundancy avoidance, intelligent backtracking, structural refinements, nagging and the use of a non-standard branch selection function. All of these allow some subproofs for a branch to be overlooked. However, redundancy avoidance and intelligent backtracking do not present a problem for failure caching. Failure caching does not actually require the enumeration of all subproofs for every branch; it only requires that, if no solutions are found for a given branch, then none exist. Redundancy avoidance and intelligent backtracking may permit some subproofs to be skipped, but, alone, they do not prevent an initial subproof from being reached. The only accommodation of these mechanisms that is necessary is applying the standard choice point marking procedure whenever a failure-cache hit prunes the search.

Nagging, the identical-ancestor refinement and the branch selection function can present problems for failure caching. Figure 5.9 illustrates this. Shading indicates the branch β for which all subproofs may be overlooked. Under the identical-ancestor refinement, some subproofs may be rejected because they contain a label matching some node in $\beta - \{leaf(\beta)\}$. Subproofs barred by the identical-ancestor refinement in the context of β might be acceptable for some different branch β' that matches β only at its leaf node. DALI must be sensitive to situations like this in which some of the context responsible for a failure is omitted from the cache entry. Other problems result from the flexibility of DALI's branch selection function. This permits the search to divide its attention among several open branches before completely closing any of them. Thus, after starting work on β, the selection function might choose to work on some other open branch of the tableau. If efforts to close this other branch fail, the search might backtrack and never attempt to fill in the unfinished portions of a subproof for β. Even though an attempt to close β is initiated, excursions to other branches prevent potential subproofs for β from being completed. Finally, nagging may prune the search in a way that obscures information about whether or not individual branches have solutions. Before the master has found an initial subproof for branch β in tableau Δ, a nagger may report that $T(\Delta)$ contains no solutions. Based on this report alone, the master cannot know if a subproof for β would eventually be found. Caching a failure for β would be unjustified and might bar the discovery of a solution elsewhere in T.

Fortunately, all of these situations may easily be detected. In DALI, each branch is given a *locality flag*. When the selection function first chooses branch β, its locality flag is set. Thereafter, the locality flag is cleared according to the following rules:

1. If backtracking is initiated because there are identically labeled nodes in $T(\mathit{leaf}(\beta))$ and $\beta - \{\mathit{leaf}(\beta)\}$, the locality flag at β is cleared.

2. Whenever the selection function chooses some branch that does not contain β, the locality flag at β is cleared.

3. If β is the selected branch from Δ and nagging prunes some portion of $T(\Delta)$, the locality flag at β is cleared.

4. If β is the selected branch from Δ and a solution is found within $T(\Delta)$, the locality flag at β is cleared.

The first three of these guard against the problems posed by the identical-ancestor refinement, the branch selection function and nagging respectively. Together, the second and fourth also censor failure insertion when a subproof is found. Typically, closing one branch leads to the selection of some different branch. In this case, the second rule prohibits future failure insertions for the closed branch. The only case in which this policy is inadequate is when the last open branch is closed; no open branches remain for the selection function, but the failure insertion for the newly closed branch must still be prohibited. The fourth rule handles this case. With these restrictions, failure caching may be safely combined with nagging and DALI's other search reduction mechanisms.

Chapter 6

Evaluation of First-Order Nagging

DALI was originally implemented as a tool for measuring the effectiveness of nagging. In the following sections I describe a *prototype nagging implementation* in DALI that is was developed for the purpose of evaluation. The effectiveness of nagging is dependent on both the particular problem at hand and the suitability of the transformation function to that problem. The prototype nagging implementation is intended to capture general-purpose nagging policies under $\mathcal{P}$ and $\mathcal{A}$ that are suitable for many types of first-order theories. Descriptions of these prototypes make explicit some of the implementation parameters that were left unspecified in Chapter 3.

6.1 Prototype Problem Transformations

To this point, discussion of nagging in DALI has omitted many specifics of the problem transformation and has focused, instead, on broad classes of these transformations. This omission of detail made it possible to establish fairly general results concerning the correctness of nagging and its extensions. Any individual function in $\mathcal{P}$ or $\mathcal{A}$ is covered by these results. Of course, actually performing nagging in a system like DALI will require that individual functions in $\mathcal{P} \cup \mathcal{A}$ be implemented.

Section 3.5.3 made it clear that the effectiveness of nagging under $\mathcal{P}$ or $\mathcal{A}$ is ultimately dependent on features of the domain theory. Knowing something about the problem domain is useful in devising a good transformation, but, even with little specific knowledge, it is possible to formulate reasonable transformations based on general characteristics of first-order theories and assumptions about expected problems. The following sections describe in some detail the particular transformation functions used in the prototype DALI implementation and assumptions on which these transformations are based.

6.1.1 Permutation Nagging Prototype

Naggers operating under $\mathcal{P}$ can be seen as trying to identify subsets of the branches in the master's tableau that cannot be closed by a single proof. DALI's prototype member of $\mathcal{P}$ tries to identify such subsets by collecting interdependent tableau branches. Here, interdependence is defined in terms of logic variables common to more than one branch. This policy is based on the assumption that branches sharing many variables will be harder to consistently close.

A permutation nagger chooses both a subset and an ordering of the master's tableau branches at the same time. As branches are placed in the subset, they are placed at the end of the nagger's ordering. This order is enforced by the nagger's branch selection function. Branches that are chosen for membership in the subset first must be closed first during the nagger's search.

Once a nagger receives a tableau Δ through a problem message, it identifies a set of *candidate* branches within the tableau. The nagger then selects a branch from among the set of candidates to use as the first branch in its permuted subset. The nagger fills in the rest of this subset by repeatedly appending branches from Δ that share variables with the branches it has already chosen. A branch β is a candidate if all of the following are satisfied.

- The labeling of β was changed in the inference from $p(\Delta)$ to Δ.
- Branch β is not selected from Δ as the first branch in the permuted subset of tableau branches on some sibling nagging process.
- Branch β is not the branch selected from Δ by the nagger's master process.

The first two of these conditions help to prevent multiple nagging processes from working on the same or equivalent transformed subproblems. The third prevents a nagger from performing the identity transformation and, thus, attempting to close a tableau that is identical to the master's. All three of these conditions are unnecessarily strong; they prohibit some transformed subproblems that would not actually constitute redundant work between nagger and master processes. However, these conditions have the advantage that they can be easily enforced by each nagger as it computes its transformation. When a DALI master process issues a problem message for Δ, it includes an indication of its own selected branch in Δ and the number of other permutation naggers that have already been assigned to work on $T(\Delta)$. The nagger uses this augmented problem message to insure that the above conditions are met. If these conditions cannot be satisfied, the nagger is unable to produce a transformation in $\mathcal{P}$ and is rendered idle.

The first condition above is used to enforce what is called the *Failure Potential Rule*. In its most general sense, this policy can be written as follows.

Definition 6.1 (Failure Potential Rule) *Let B be the set of branches in both tableau Δ and $p(\Delta)$ such that the labelings of B in Δ and $p(\Delta)$ are syntactic variants. Naggers working under $\mathcal{P}$ are not permitted to work on transformations of Δ that retain branches in B only.*

This rule can be compared to the Last-Call Rule of Chapter 4. It enforces a policy for choosing between nagging on a node or its parent. The failure potential rule prohibits situations where naggers working on Δ and $p(\Delta)$ might select identically labeled sets of branches. Unlike the last-call rule, the failure potential rule favors nagging on the parent rather than the child.

The nagger builds its permuted branch subset of Δ by starting with the leftmost candidate branch in Δ that has the fewest number of distinct variables in the label on its leaf. The nagger then repeatedly appends branches with the highest *score*. Every time a branch is added, each remaining branch β of $\Delta = \langle \tau, \mu \rangle$ is given a score, computed as follows:

- The *base score* of β is the number of distinct variables in $\mu(\mathit{leaf}(\beta))$ that also appear in some $\mu(\mathit{leaf}(\beta'))$ where β' has already been placed in the nagger's permuted branch subset.

- If the base score is non-zero, the score is computed by subtracting from the base score the number of distinct variables in $\mu(\mathit{leaf}(\beta))$ that do not appear in any $\mu(\mathit{leaf}(\beta'))$ where β' has already been placed in the nagger's permuted branch subset.

- If the base score is non-zero and either β was created in the inference from $p(\Delta)$ to Δ or its labeling was changed beyond syntactic variation, the score of β is increased by 1/2.

The nagger continues enlarging its subset of branches as long as there are other branches a non-zero base score.

This score is designed to reflect the degree to which a branch is interrelated with other branches the nagger will be attempting to close. Branches sharing many variables are considered highly interrelated. If β contains variables that do not occur in other selected branches, it is penalized in score. In both cases, only the branch leaves are consulted, potentially overlooking some instances of branch interdependence. Although this focus on the leaves makes a small contribution to implementation efficiency, its primary intent is to improve the quality of nagger's branch subset. Variables appearing in $\mu(\mathit{leaf}(\beta))$ can be seen as more influential in the closing of β than those elsewhere in β. Extension depends only on the label of $\mathit{leaf}(\beta)$, while variables elsewhere in β can only be reached through reduction. This is particularly relevant when the theory is Horn. Furthermore, the heuristic value of this scoring mechanism would be dramatically curtailed if the distinctions in variable population at the leaves of two branches could be obscured by non-leaf nodes shared by the branches. The third component of the score enforces a policy of breaking ties in a way that is consistent with the Failure Potential Rule.

When the master assigns work to permutation naggers, it must choose from among all ancestors of its current search node. The master selects the one closest to the root of T that still contains a *candidate* branch.

6.1.2 Abstraction Nagging Prototype

Abstraction nagging works by re-mapping symbols from the original theory so that distinct clauses are made logically equivalent. After eliminating newly redundant clauses, the nagger's search space may be smaller than the master's. DALI's prototype member of $\mathcal{A}$ tries to identify clauses of the original theory that are structurally similar. If possible, symbols of the theory are re-mapped so that these clauses can be replaced by a single representative clause.

Naggers operating under $\mathcal{A}$ generate their abstraction mappings before making an initial idle report to the master. While permutation naggers compute a new problem transformation for each subproblem they encounter, abstraction naggers compute only a few abstraction mappings and then apply one of these mappings to each subproblem they see. This difference follows from a difference in computational expense. Identifying similar clauses of the theory can be very expensive for large theories. In contrast, the cost of applying an abstraction in DALI is proportional to the number of tableau nodes. If abstraction naggers are to be effective, they must be able to generate useful abstractions of the theory in a small number of attempts.

The theories generated under DALI's abstraction mechanism may be thought of as maximally conservative; symbols are re-mapped only in ways that contribute to making one clause *as general* as another. DALI builds abstraction mappings by repeatedly identifying an *abstractable clause pair.*

Definition 6.2 (Abstractable Clause Pair) *Clauses C_1 and C_2 are called an* abstractable clause pair *if, there exist abstraction mapping g_{abs}, abstracted clause $C_1' \in g_{abs}(C_1)$ and abstracted clause $C_2' \in g_{abs}(C_2)$ such that C_2' is* as general *as C_1'. The equivalence relation associated with g_{abs} is constrained so that symbols of differing arity cannot be placed in the same equivalence class.*

As DALI identifies abstractable clause pairs, it constructs an aggregate abstraction mapping that is capable of eliminating the less general member of each pair. Since each abstraction mapping is determined by an equivalence relation on symbols of the theory, constructing this aggregate mapping simply involves building the transitive closure of all equivalence relations needed to eliminate one member of each pair. When selecting the abstracted theory from among the members of $g_{abs}(S)$, new variable terms are included only where necessary to render one member of each clause pair as general as the other.

Nagging processes typically do not eliminate all clauses possible. At the beginning of search, each nagger randomly chooses a percentage $x \in [15, 85]$. The nagger randomly chooses x percent of the abstractable clause pairs and generates its abstracted theory to eliminate one clause from each pair. Thus, some naggers will generate rather mild abstractions of the theory while others will consider more extreme ones. The master keeps a record of the number of clauses in each nagger's abstracted theory. When assigning a subproblem to an abstraction nagger, the master chooses the tableau closest to the search tree root such that no abstraction nagger with the same theory size has already worked on that tableau. In this way, the master prevents two naggers from working on the same transformed problem. If naggers have theories of different sizes, these theories must differ and, therefore, the search spaces they entail will differ. As with DALI's management of functions in $\mathcal{P}$, this policy is unnecessarily restrictive. Two naggers may, of course, generate different theories that have the same number of clauses. Theory size does, however, represent a convenient metric with which to guarantee that naggers do not duplicate work.

Once a nagger has developed its abstracted theory, it reports idle and begins receiving problem messages from the master. Let g_{abs} represent the aggregate abstraction mapping generated by the nagger. During search, the nagger constructs the root of its transformed search tree by applying the trivial abstraction[1] in g_{abs} to the tableau it receives from the master. The nagger then attempts to close this tableau under its abstracted theory. In this way, information may be lost in the theory, but information is never discarded from the master's tableau. Although nontrivial abstraction of incoming tableaux is permitted by the definition of $\mathcal{A}$, it would typically give the nagger a larger search space than necessary.

As abstraction naggers consider successive master-supplied subproblems, they

[1]From the proof of Theorem 3.3, the trivial abstraction is the member of g_{abs} that introduces the fewest new variables. Informally, the trivial abstraction re-maps symbols in a way that does not discard information. Under the prototype abstraction function, the trivial abstraction never has to introduce new variables.

monitor their own effectiveness and try to ascertain the quality of their theory abstraction. With each nagging attempt, there are three possible outcomes:

1. The nagger may issue a prune message.

2. The master may complete the original subtree first and preempt the nagger.

3. The nagger may find a solution to the transformed problem and become idle.

Abstraction naggers use these categories to assess their effectiveness. Ideally, all outcomes would fall in the first category, but, realistically, the other cases will occur frequently. If more than 80 percent of an abstraction nagger's nagging attempts fall into the third case, this is taken as evidence that the nagger's theory abstraction is too extreme. The assumption is that the nagger has altered so many clauses that many things that are not provable in the original theory are provable in the abstracted theory. If this occurs, the nagger discards its current abstracted theory, cuts its abstraction percentage x by half and generates a new abstraction based on this new percentage.

If more than 80 percent of the nagging attempts fall into the second category, the nagger takes this as evidence that its abstraction is not sufficiently extreme. The correctness of this explanation is less certain than in the former case, but the nagger responds by trying to generate a new theory with fewer clauses. It discards its current theory and builds a new one with the abstraction percentage x taken as the average of its previous value and 100.

6.2 Evaluation Results

Evaluating the effectiveness of nagging in practice requires a collection of first-order test problems that can be used as a basis for comparing various serial and parallel versions of the theorem prover. In an effort to represent an unbiased sample of first-order logic problems, I have taken this suite of problems from the *Thousands of Problems for Theorem Provers* (TPTP) library [SSY94]. The TPTP is a collection of more than 2,500 first-order logic problem given in disjunctive normal form. It represents a broad range of domains including many theories given in literature. I have randomly chosen a 500-problem subset of the TPTP, which is used across all experiments.

All experiments were conducted on a local area network of Sun workstations. In the case of serial experiments, DALI was run on a single Sun Sparc 670MP. For parallel trials, the top-level master process was run on this machine and nagging processes were run on a variety of other machines that were computationally no more powerful than the master's processor. In all experiments, all machines were shared among a number of users. Although experiments were conducted at night to minimize contention for computation resources, DALI did not generally have exclusive access to all of its processors.

Problems in the TPTP vary greatly in size and complexity. While some are quite simple and are quickly solved by even the most naive configuration of DALI, others are much more complicated and are well out of DALI's reach in any reasonable amount of time. This is consistent with the model of resource-bounded search presented in Chapter 1. To deal with this extreme variation in problem difficulty, search on behalf of any single problem was bounded. In both its parallel and serial forms, DALI was permitted to spend no more than 10 minutes of elapsed time trying to solve each problem.

The evaluation of first-order nagging in DALI is reported in three sections. Section 6.2.1 compares a simple, serial configuration of DALI as described in Chapter 3 with an equivalent system using the basic nagging framework described in Chapters 2 and 3. Section 6.2.2 compares this simple serial system with serial and parallel systems equipped with the search refinements of Chapter 5. Finally, Section 6.2.3 compares a refined serial search with nagging systems using search refinements and the extended nagging models described in Chapter 4.

6.2.1 Naive Nagging

Figure 6.1 compares the performance of a 17-process parallel search with that of a serial search. Here, the serial search is considered naive; it uses the basic model-elimination search procedure described in Chapter 3, but includes none of the search refinements given in subsequent chapters. The parallel system includes a single master process and 16 nagging processes, all using the prototype transformation in $\mathcal{P}$. Both master and nagging processes are implemented with the same search procedure as the serial search. The nagging behavior is based on the minimal protocol presented in Chapter 2, augmented with preemptive problem messages. Preemption is permitted here for two reasons. It is used by the prototype abstraction mechanism as a prompt to adjust the abstracted theory. Also, it was part of the nagging protocol when it was originally developed and was excluded from Chapter 2 only to simplify presentation and proofs. An experiment at the end of this section focuses on the value preemption in nagging performance. None of the other nagging extensions defined in Chapter 4 are active in Figure 6.1. Accordingly, all 16 nagging processes operate directly under the single master.

The format of Figure 6.1 is used throughout this section, so it is described in some

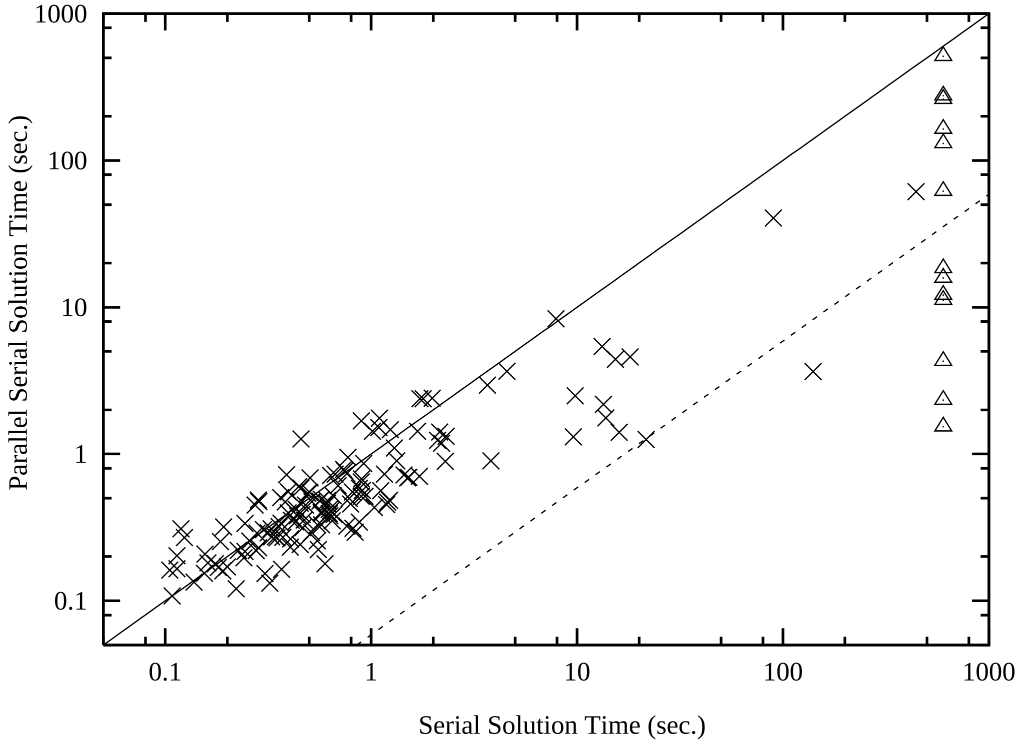


Figure 6.1: Master with 16 permutation naggers vs. naive serial search. The parallel system solved 13 more problems in the 10 minute time limit than the serial system.

detail here. This figure plots the performance of the parallel system against that of the serial system. Each plotted point represents the solution of a single problem instance from the 500-problem subset of the TPTP. The X coordinate of each point indicates its solution time on the serial system, measured as the elapsed time from the start of search to the moment a solution is reported. The Y coordinate reflects its parallel solution time, measured in the same way. Thus, 10 minutes of parallel search represents a greater computational resource than 10 minutes of serial search. Both axes are displayed in log scale.

Neither the serial nor the parallel system was able to solve all 500 test problems in the 10 minute time limit. The serial system solved 146 of them. The parallel system solved the same 146 problems plus an additional 13 problems that the serial system could not solve in 10 minutes. Figure 6.1 plots the performance on all problems solved by either system. Problems solved by both are represented by the $\times$ symbol. Those solved by only the parallel system are indicated by $\triangle$. Since the serial system did not solve these problem, the actual serial solution time is not available for plotting. Instead, the 10 minute time limit is used as a lower bound on the actual serial solution time. If these points were plotted at their true positions, they would be some margin to the right of where they now appear.

The upper diagonal line in Figure 6.1 represents equal parallel and serial solution time. Points above this line represent problems solved faster by the serial system, and points below are problems solved more quickly in parallel. The lower diagonal line represents the threshold for linear speedup. A point on this line would indicate a problem solved 17 times faster by the 17-processor parallel search.

Figure 6.1 suggests that nagging is not uniformly beneficial across all problems. Among the smallest problems, those solved in less than 5 seconds on the serial system, nagging does not display a clear advantage. Some of these small problems are helped, but a similar number seem to be hurt by parallelism. In both cases, nagging has only a small effect on solution time.

Of course, this non-uniform benefit is to be expected. Like any search refinement, nagging comes with a measure of overhead that is not directly related to problem difficulty. For smaller problems, the parallel search reduction cannot always compensate for this overhead. The noisy appearance of points on the left-hand side of Figure 6.1 can be explained through Figure 6.2. This graph compares the performance of the same serial system across two trials. Since the operation of the serial search engine is deterministic, any deviation from the diagonal must be attributed to noise in the measure. Typically, this is caused by other processes that are competing for shared computing resources.

Fortunately, these anomalies of nagging-induced overhead and noisy performance

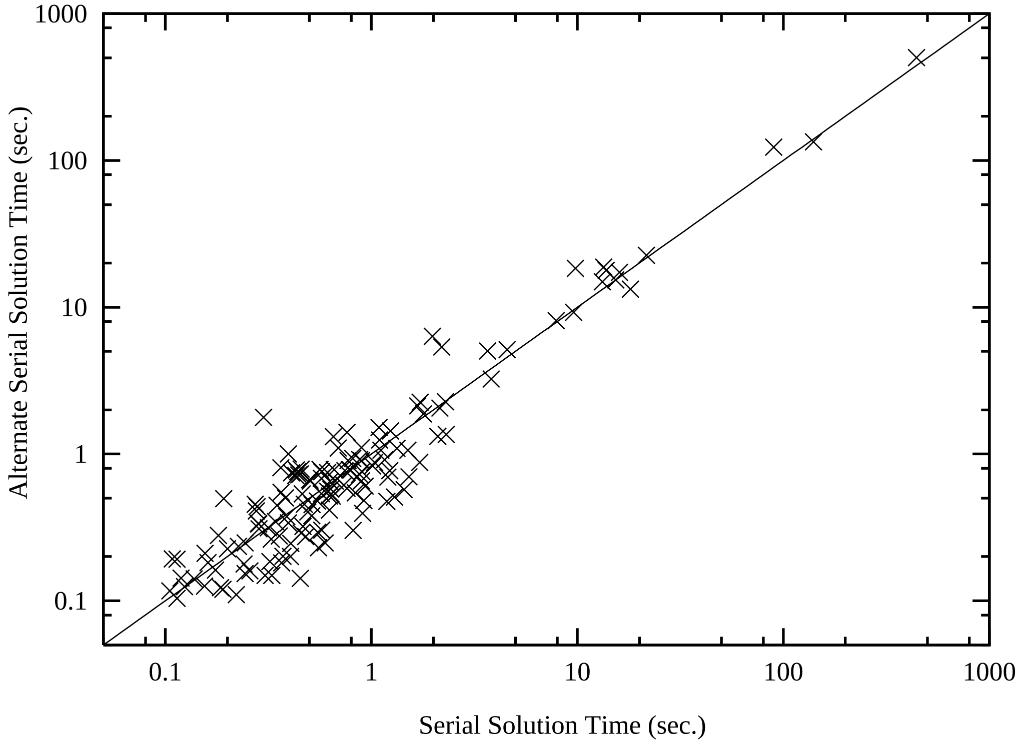


Figure 6.2: Comparison of identical serial searches across two trials. An abundance of noise is evident in the solution times for smaller problems.

measurement are most pronounced on the smaller problems. These are precisely the problems for which performance improvement is not crucial. If the intent is as suggested in Chapter 1, to increase the problem solving power of a search-based system, then performance improvement on larger problems is the most useful. Consequently, analysis of nagging performance should reflect this preference and will henceforth exclude those problems that took fewer than 5 seconds to solve on the serial system.

Figure 6.3 focuses on this smaller set of interesting problems. For comparison purposes, this figure includes a summary statistic for parallel performance. Performance improvement is measured as the speedup factor, serial solution time divided by parallel solution time. The thick, double line in Figure 6.3 reflects the average speedup across all problems in the graph[2]. On average, the permutation nagging system solves these problems 42.22 times as fast as the serial. However, since 13 of the points contributing to this statistic feature optimistic estimates of serial solution time, the true average speedup could be higher.

Although this average is encouraging, the general distribution of points in the figure indicate that nagging often falls short of perfect linear speedup. The average exceeds linear speedup because there are a few problems that vastly exceed linear speedup. This is particularly noticeable for problems solved by the parallel system only. If these points were plotted at their correct positions, they would be displaced to the right and, consequently, pulled down with respect to the diagonal lines. Any points below this lower diagonal represent problems solved more than 17 times as quickly on 17 processors. Under ordinary parallelization schemes, linear performance improvement is an upper bound. Of course, a nagging search may not represent a parallelization of any single serial search. As shown in Lemma 2.4, the master's search is related to its serial counterpart, but work done by a nagger may have no simple analog in the underlying serial search. Transformation can reduce a nagger's search in ways that are independent from the number of participating processes. This unbounded potential for search reduction accounts for the non-linear performance improvement in Figure 6.3. Greater numbers of nagging processes simply increase the opportunity for one to discover a highly effective transformation.

Figure 6.4 plots the performance of a parallel search with 16 abstraction naggers against the same, naive serial search used in previous figures. As before, the parallel system uses only preemptive problem messages from Chapter 4; all 16 naggers work directly under a single master.

[2]In many of the graphs, this line appears disproportionately influenced by points in the lower right corner. This appearance is a consequence of the log-log scale. While the parallel system demonstrates a modest disadvantage on some problems, it solves others more than 100 times as quickly as the serial system. The log-log scale obscures the degree to which the parallel system outperforms the serial on these problems.

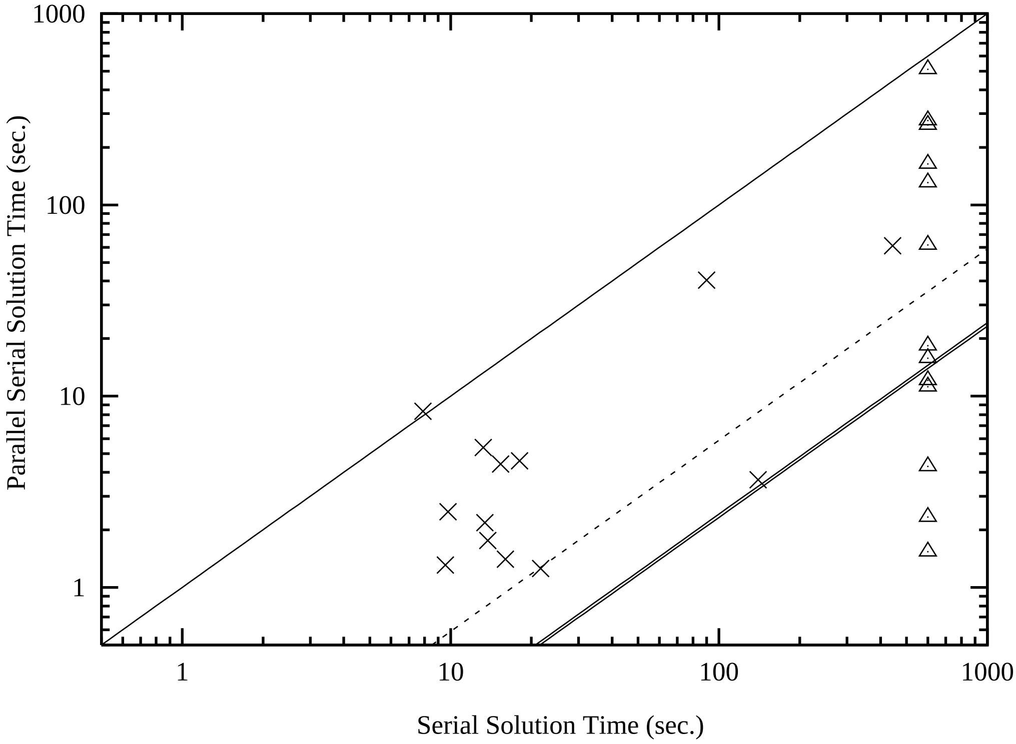


Figure 6.3: Master with 16 permutation naggers vs. naive serial search. Here, problems solved in less than 5 seconds by the serial search are excluded.

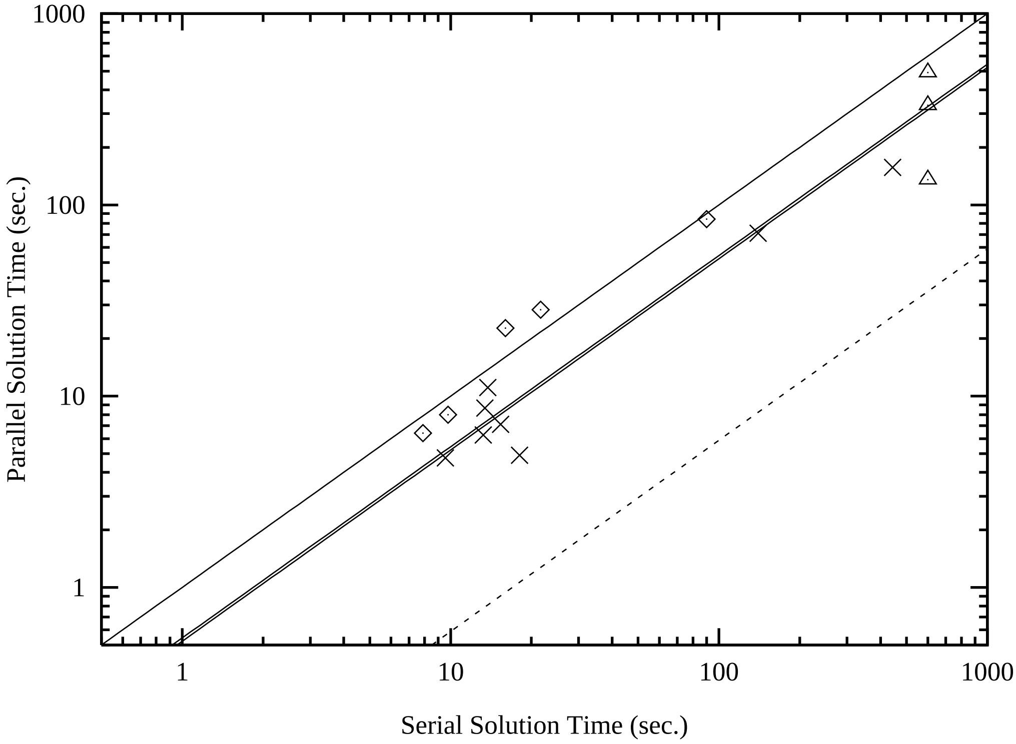


Figure 6.4: Master with 16 abstraction naggers vs. naive serial search. The parallel system solved 3 more problems in the 10 minute time limit than the serial system.

As suggested in Section 3.5.3, not all problems are amenable to abstraction nagging. Of the 500 problems in the test suite, only 301 are candidates for abstraction. The other 199 contain no abstractable clause pairs. When naggers determine that the current theory cannot be abstracted, they simply terminate and leave the master to solve the problem alone. Thus, problems that are not abstractable suffer from nagging-induced overhead but enjoy no search reduction. Figure 6.4 shows results for all problems. Problems that have no abstraction under the prototype abstraction function are indicated by the $\diamond$ symbol.

While permutation nagging permits 13 additional problem to be solved in the time bound, abstraction nagging provides only 3 new problems. Visual inspection of the figures also suggests that the prototype permutation nagging system is much more effective. This impression is supported by the average speedup under parallelism. While permutation nagging achieved an average speedup factor of 42.22, abstraction nagging achieved only 1.875. In general, these measures cannot be used to reliably compare the search-reducing power of two systems. Since both are conservative, there is no guarantee of how they would compare if all serial solution times were known. Here, the comparison is more meaningful because the abstraction nagging system solves a proper subset of the problems solved under permutation.

Some of this disparity between the prototype transformations can be explained by the fact that abstraction nagging is applicable to only a subset of the problems. On problems that are not amenable to abstraction, these naggers simply generate overhead. It is conceivable that abstraction nagging is effective, but only on a subset of the problem sample. To let each type of nagger contribute where it is most effective, the parallel system can be given access to both. Figure 6.5 presents such an experiment.

Under both of these prototype transformations, specifics of the transformation are left up to the nagger; the master does not need to be concerned with which class of transformation is used by each of its naggers. As a result, both types of naggers can be used at the same time with little special accommodation by the master. The parallel system in Figure 6.5 features 8 naggers working under the prototype permutation function and 8 under prototype abstraction. Abstraction naggers are permitted to convert to permutation if the theory is unresponsive to abstraction. Of the 500 test problems, only 301 are candidates for abstraction. As soon as a nagger determines that it is unable to abstract the theory, it is licensed to begin generating transformations under $\mathcal{P}$.

The performance of this mixed-nagger system is plotted against the same serial search used in previous figures. Here, the mixed-nagger system is able to solve 11 more problems than the serial system. This includes all but two problems solved

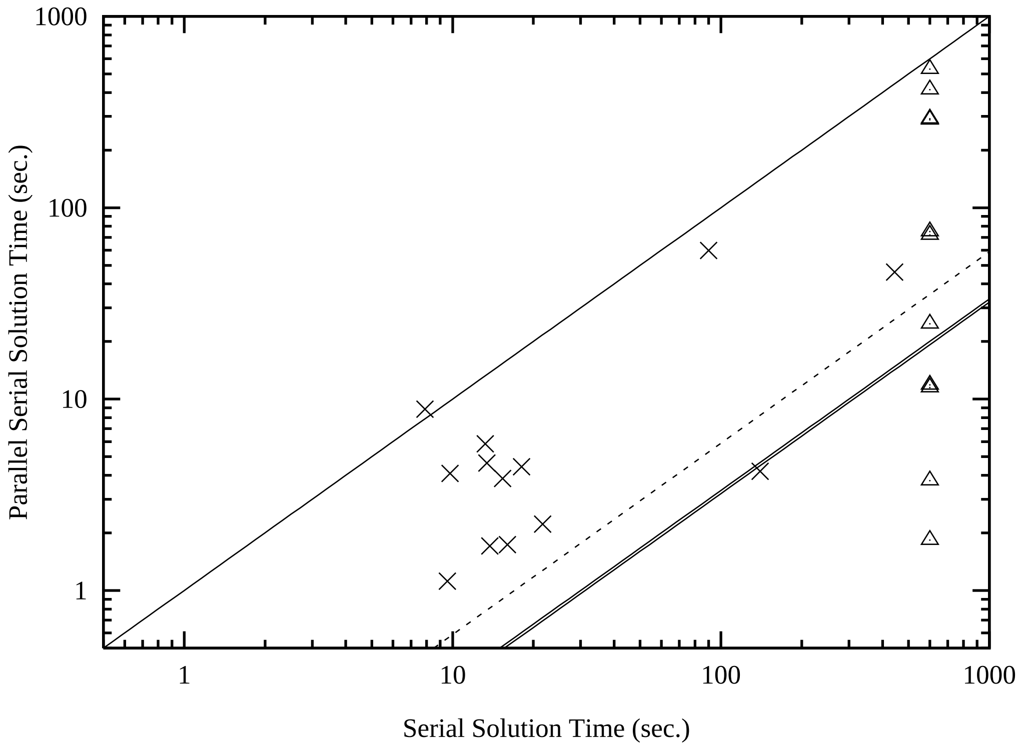


Figure 6.5: Master with 8 permutation and 8 abstraction naggers vs. naive serial search. The parallel system solved 11 more problems in the 10 minute time limit than the serial system.

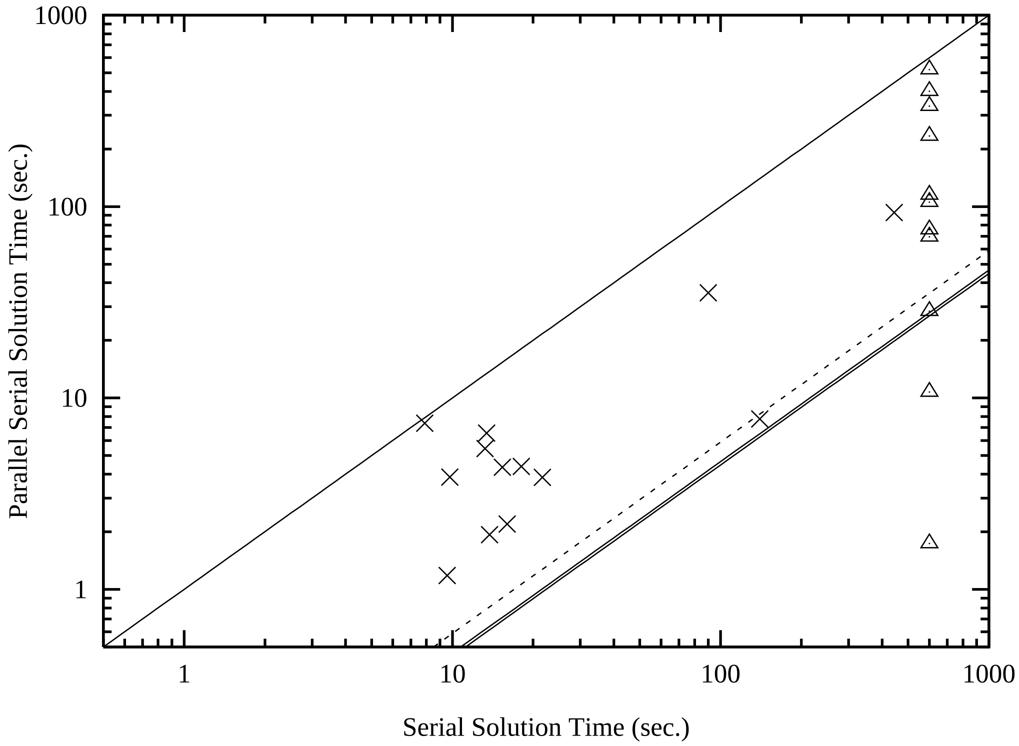


Figure 6.6: Master with 8 permutation and 8 abstraction naggers vs. naive serial search. Without the use of preemption, the parallel system still solved 11 more problems than the serial.

by the permutation only system, and all problems solved by the abstraction-only system. In general, this mixed-nagger system achieves an average speedup of 30.54 across all problems it solves. This is a significant improvement over the abstraction-only system but still a bit short of the permutation system.

As indicated at the start of this section, it is illuminating to consider the value of preemption as an element of the nagging protocol. This is particularly interesting because preemption seems to be much less vital than might be expected. Figure 6.6 plots the performance of a parallel nagging system without preemptive problem messages. The parallel system here is identical to the one in Figure 6.5, except that, once a nagger receives a search problem, it must complete it before receiving another. Even without the ability to interrupt naggers when their work becomes irrelevant, the parallel search is still able to solve the same set of problems. Preemption offers a clear performance advantage since average speedup without it is only 21.94 (as opposed to 30.54 with preemption). However, this advantage is much less pronounced than was originally anticipated.

6.2.2 Serial Search Refinements

Although appraisal of DALI's nagging component is the primary goal of this evaluation, DALI's serial search refinements represent significant contributions to both the complexity and the performance of the overall system. Including these refinements is vital to a fair evaluation of nagging. The preceding section shows that nagging can be effective at speeding the performance of a somewhat naive theorem prover. In practice, however, its contribution to the performance of a sophisticated search engine is a better indicator of its usefulness. Nontrivial search reduction in the context of a sophisticated serial search will be a much stronger result in favor of nagging and a much stronger indication that the search reduction achieved through nagging is "orthogonal" to that attained through common refinements of the underlying search procedure.

This section presents performance results for each of the major search refinements given in Chapter 5. Chapter 5 demonstrates that all of DALI's refinements, serial and parallel, may interact in non-obvious ways. In particular, it identifies many ways in which DALI's serial search refinements conflict with nagging. Some changes to the search procedure even cancel the search-ordering properties nagging would otherwise be guaranteed. Naturally, the assumption is that these disadvantages will be compensated by reduced search on typical problems. Isolating the contribution of each search refinement demonstrates that the performance advantage is usually worth the added complexity.

Figure 6.7 compares the performance of a serial search equipped with the intelligent backtracking scheme of Section 5.1 against a serial search that lacks the intelligent backtracking mechanism but is otherwise equivalent. As with the results given in the previous section, each point indicates solution times for a single problem in the 500-problem sample. The Y axis represents solution time with intelligent backtracking and the X axis gives solution time without it. As both systems are serial, Figure 6.7 includes only one diagonal line, discriminating problems solved more quickly by the refined search from those solved more slowly. Those points below the line represent problems solved in less time by the more sophisticated search procedure.

Many of the problems in Figure 6.7 are solved more quickly with intelligent backtracking. This is particularly evident in the larger problems. There are, however, a few problems that suffer under intelligent backtracking. This includes one (denoted by $\diamond$) that goes unsolved. This nonuniform performance is consistent with what was seen in the N-queens problems in Chapter 1. Search refinements are built around knowledge or assumptions about properties of typical problems. If these assump-

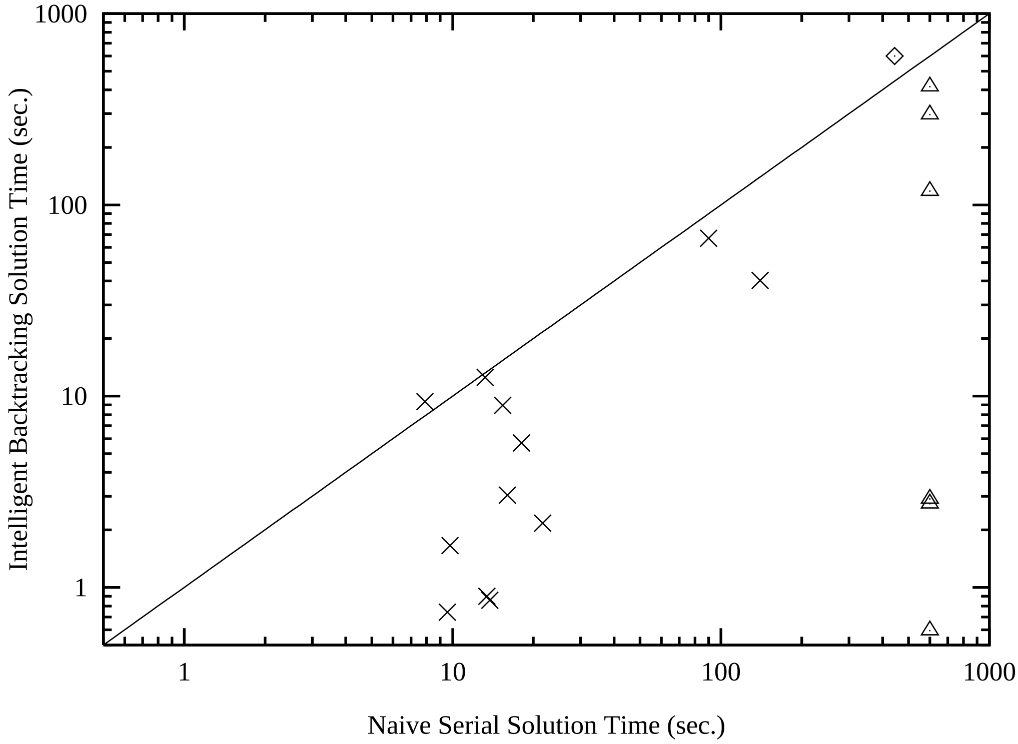


Figure 6.7: Intelligent backtracking serial search vs. naive serial search. Intelligent backtracking permitted 7 extra problems to be solved in the 10 minute time limit.

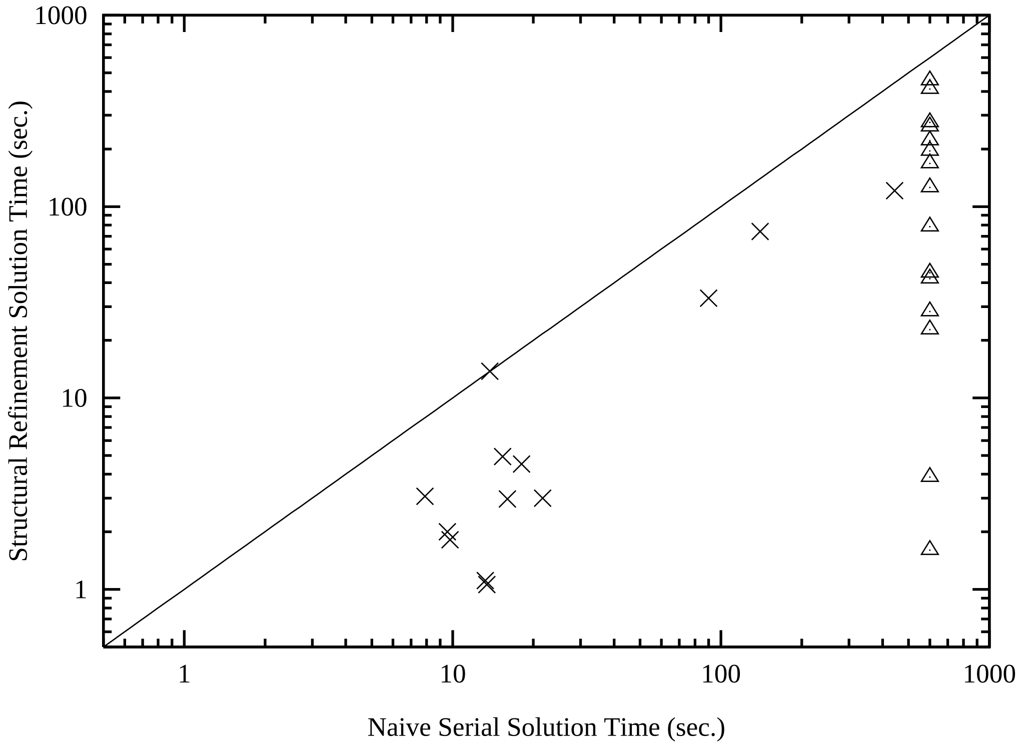


Figure 6.8: Serial search with structural refinement vs. naive serial search. Structural refinements permitted 15 extra problems to be solved in the 10 minute time limit.

tions are not uniformly valid, some problems will benefit from the refinement while others may suffer. Some of the points in Figure 6.7 reflect problems that seldom generate the repeated branch labelings that intelligent backtracking tries to exploit. Intelligent backtracking may be able to effect a small measure of pruning in these problems, but most of its influence is manifested as computational overhead.

Again, there are problems that were solved by the intelligent backtracking system within the 10 minute time limit that were not solved by the basic search within the same time limit. Here there are 7 extra problems solved by the refined serial search. Of these, 2 represent problems that were solved by none of the parallel nagging systems of the previous section. Thus, intelligent backtracking appears to capture some opportunities for search pruning that are not achieved through nagging alone.

Figure 6.8 compares the performance of a serial search equipped with the structural refinement of Section 5.2 against a serial search without this refinement. The structural refinement enforced here represents two cases of the identical-ancestor refinement, checked at the beginning and the end of each subproof construction. With this refinement, the serial search is able to solve 15 more problems in the 10 minute

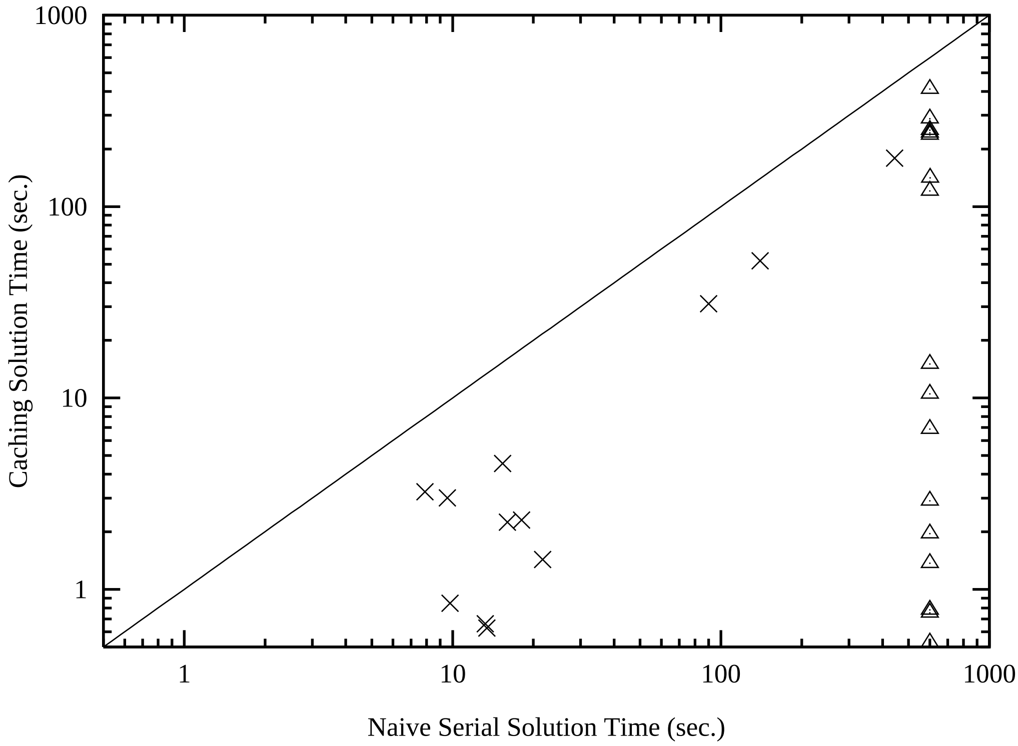


Figure 6.9: Serial search with a 200-element cache vs. naive serial search. The caching system was able to solve 17 additional problems in the 10 minute time limit.

time limit. Of these 15, 10 are problems that were not solved by the intelligent backtracking serial system and 5 are problems not solved by any of the parallel nagging systems of the previous section. Again, most problems benefit greatly from DALI's structural refinements.

Figure 6.9 compares the performance of a serial search equipped with the caching mechanism of Section 5.3 against a serial search without caching. The caching system uses a 200 element cache with a least-recently used cache replacement policy [SS93]. Both success caching and redundancy avoidance are used on all theories, and failure caching is used for Horn-clause theories. Here, 17 more problems were solved with the caching mechanism than were solved without it in the 10 minute time limit. Of these 17, the caching system was able to solve 3 problems that were solved by neither the intelligent backtracking nor the structural refinement system; six of these 17 problems were not solved by any of the parallel nagging systems.

Figure 6.10 compares the performance of a serial search equipped with intelligent backtracking, the structural refinement and caching against a serial search without any of these refinements. As in all experiments, the caching component enforced a 200-element bound on cache size. By necessity, Chapter 5 addressed the possibil-

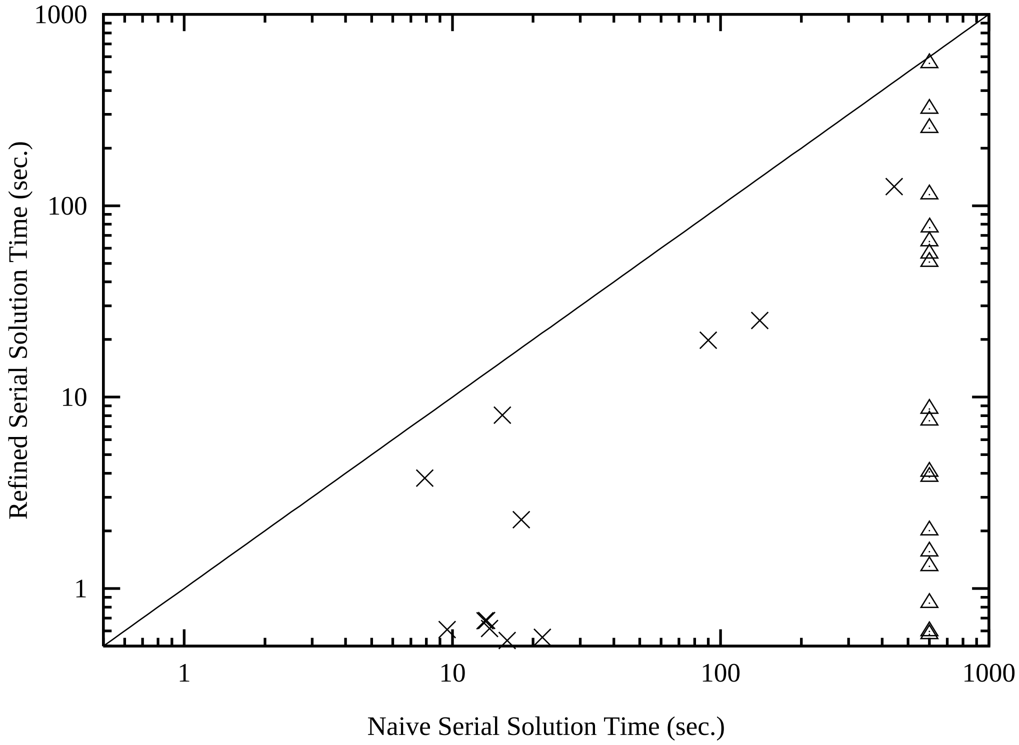


Figure 6.10: Serial search with intelligent backtracking, structural refinements and a 200-element cache vs. naive serial search. The refined serial system was able to solve 20 additional problems in the 10 minute time limit.

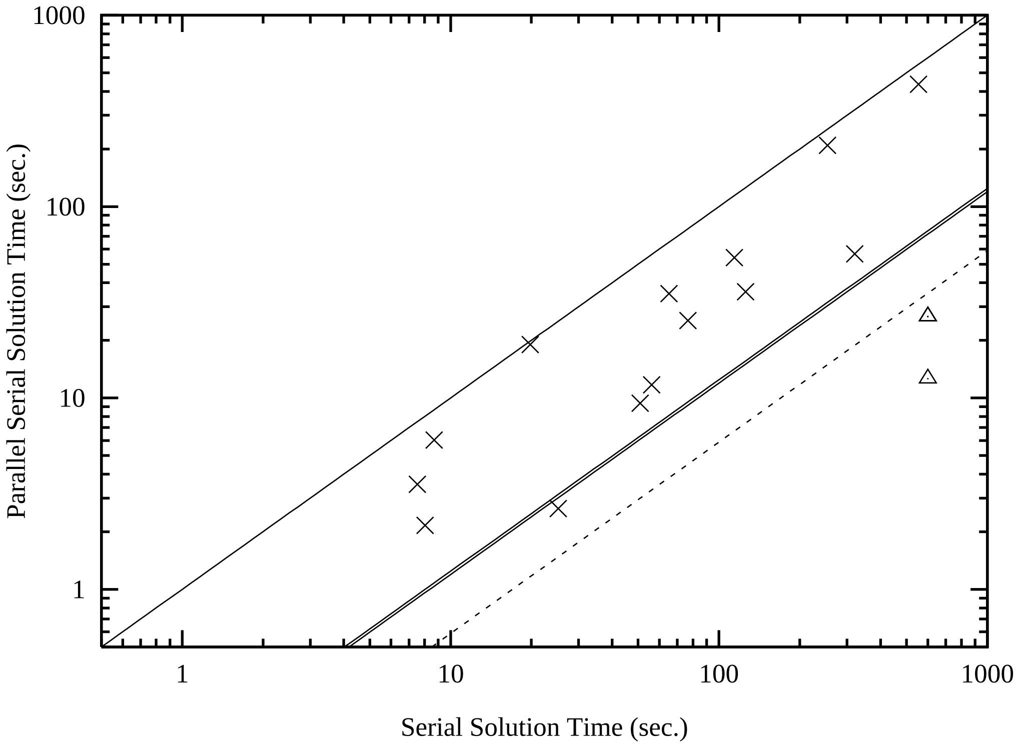


Figure 6.11: Parallel nagging search vs. serial search, both with all serial search refinements. Using 8 permutation and 8 abstraction naggers, the parallel system is able to solve 3 more problems than the serial system in the 10 minute time limit.

ity for interference between DALI's various search refinements. In all cases, search completeness can be maintained by requiring one mechanism to accommodate another, but some potential for adverse interaction remains. Consequently, there is the possibility that these refinements would function poorly in combination even though they work well individually. This experiment demonstrates that multiple, serial sources of search reduction can operate quite well in combination even when they don't operate in complete independence. Here, 20 problems were solved by the refined system that could not be solved by the naive serial search in the 10 minute time limit. In fact, the refined system was able to solve all problems solved by any of the serial systems that used only one search refinement. This most refined serial system solved 9 problems that none of the parallel nagging systems could solve and was even able to solve 2 new problems that were solved by none of the other serial searches.

Chapter 5 also negotiates some of the problems of combining nagging with DALI's serial search refinements. Here also, there is the possibility that nagging will interfere destructively with these refinements. Figure 6.11 compares the performance of a parallel nagging and serial serial search, both using all serial search refinements. The solution times plotted along the X axis here are the same as those given on the Y axis of Figure 6.10. In the parallel search, master and nagging processes use all of serial refinements, including a 200-element cache on each.

Figure 6.11 suggests that the potential for adverse interaction between nagging and DALI's serial search refinements is not realized in its fullest severity. Here, the parallel system is able to solve all problems solved by its serial counterpart, plus an additional 3 problems. Unfortunately, the advantages of naive nagging appear to be much less impressive in the presence of a more discriminating search procedure. Nagging still increases the number of solved problems but it attains an average speedup of only 8.199.

6.2.3 Extended Nagging Mechanisms

The two previous sections demonstrate that even a fairly simple nagging model can be effective at reducing search. Experiments in this section consider the value of the various extensions to this model introduced in Chapter 4. Each of these extensions is considered individually. Finally, a parallel search combining all refinements, parallel and serial, is evaluated.

Figure 6.12 compares the performance of a 17-process hierarchical nagging system with a serial search. For all experiments in this section, search engines use all refinements introduced in the previous section; both the master process and it's naggers maintain independent, 200-element caches. Thus, the serial solution times plotted on X axis are based on the same system as those in Figure 6.11, a refined serial search with 200-element cache.

Figure 6.13 illustrates the hierarchical arrangement of naggers used to generate the results given in Figure 6.12. This hierarchy includes 8 permutation and 8 abstraction naggers. Two of each work directly for the master process, and each of these has three recursive naggers, including at least one working under $\mathcal{P}$ and one under $\mathcal{A}$.

The parallel system in Figure 6.12 is able to solve only 5 more problems than the serial system. This includes all of the problems solved without recursive nagging in Figure 6.11. However, the average speedup of this system is only 7.943. Although the comparison is not completely meaningful, this is a value similar to the 8.199 attained when all naggers worked directly for the master. These similar performance characteristics are encouraging because, in the hierarchical experiment, only one fourth as many naggers have direct contact with the master. Any search reduction provided by the other 12 processes must be mediated by a nonterminal nagger.

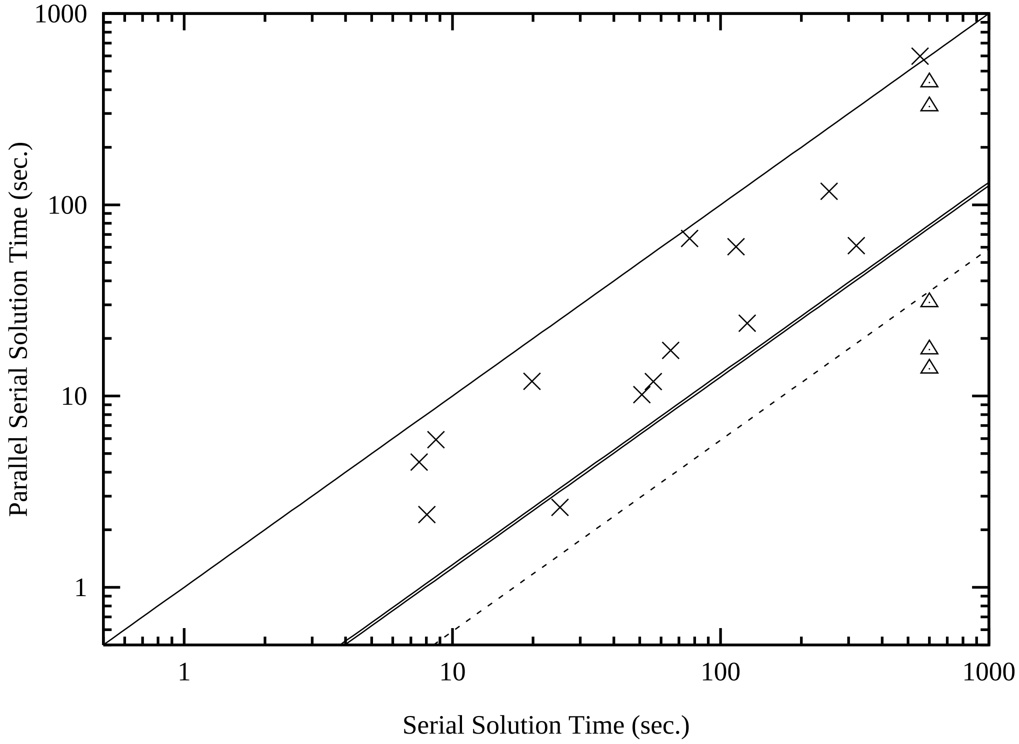


Figure 6.12: Performance of a 17-processor parallel system using recursive nagging vs. a serial system, both based on a refined serial search. The parallel system is able to solve 5 more problems than the serial system in the 10 minute time limit.

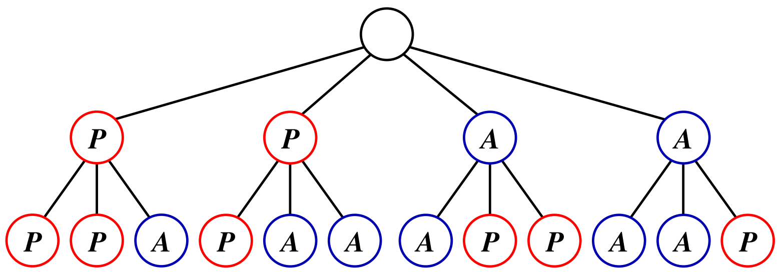


Figure 6.13: Hierarchical arrangement of 8 permutation and 8 abstraction naggers. The root node represents the top-level master process. Non-root nodes represent naggers and are marked according to the class of problem transformation used by the nagger.

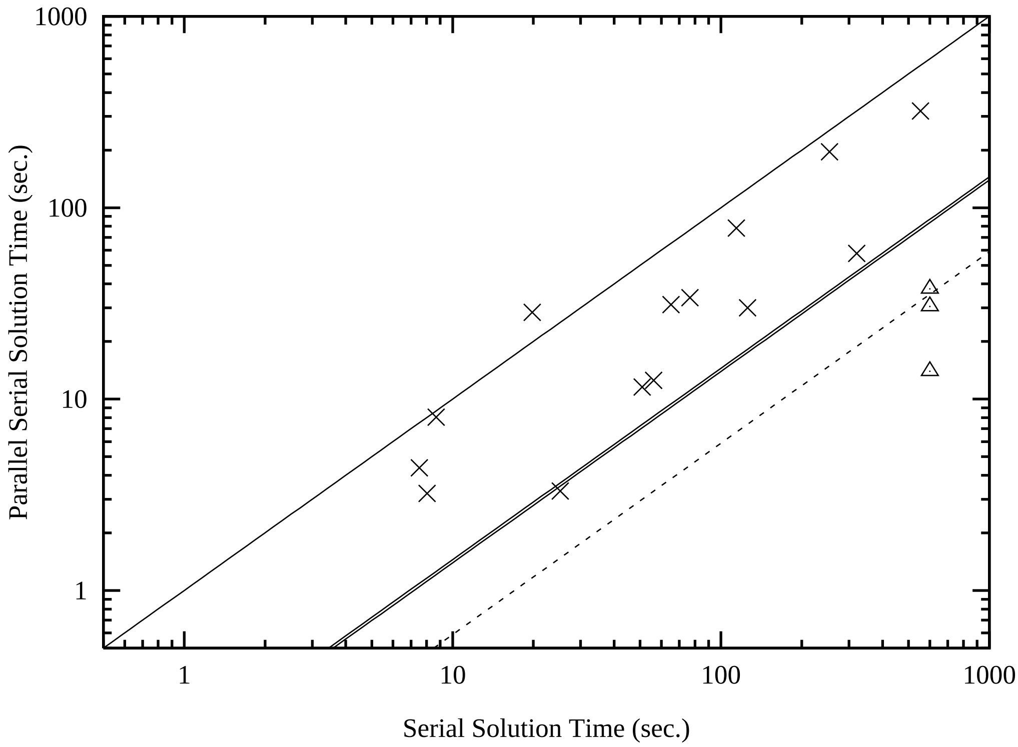


Figure 6.14: Performance of a 17-processor parallel system using the last-call rule vs. a serial system, both based on a refined serial search. The parallel system is able to solve 3 more problems than the serial system.

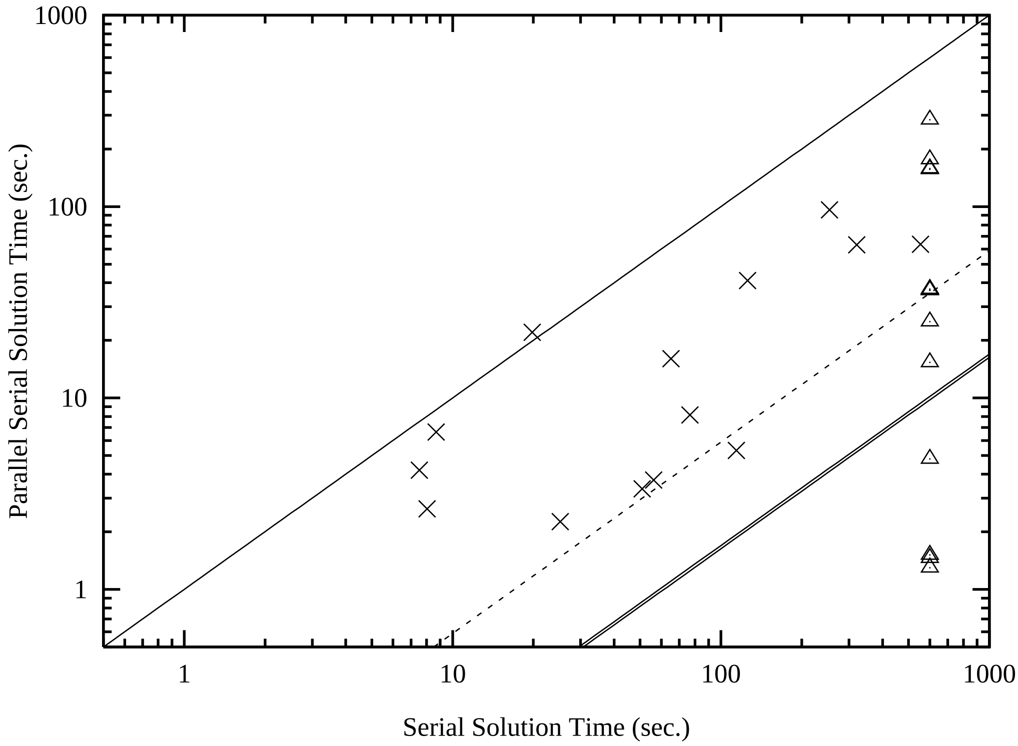


Figure 6.15: Comparison of the performance of a 17-processor parallel system utilizing nagger-discovered, weakly local subproofs and a serial system. The parallel system is able to solve 13 more problems than the serial system.

Figure 6.14 plots the performance of a 17-process nagging system with the last-call rule active. This experiment considers only the value of the last-call rule among nagging's extensions, so the 8 abstraction and 8 permutation naggers work directly under a single master. Results under this extension are somewhat disappointing. This system is able to solve the same 3 problems not solved by the serial system in Figure 6.11, but it's average speedup falls short. Speedup with the last-call rule is only 7.038, while it is 8.199 without it.

Figure 6.15 plots the performance of a 17-process nagging system in which the master is permitted to exploit nagger-discovered, weakly local subproofs. Using this extension of the nagging protocol, the parallel system is able to solve 13 more problems than the serial. With only one exception, this system solves all problems solved by any of the previous systems, serial or parallel. The exception is one problem solved by the hierarchical nagging system. This encouraging statistic in solved problems is supported by an average speedup of 60.16.

Figure 6.12 considers the value of eager search pruning within nagging. As in the other cases, the parallel system uses 16 nagging processes. Here, the nagging

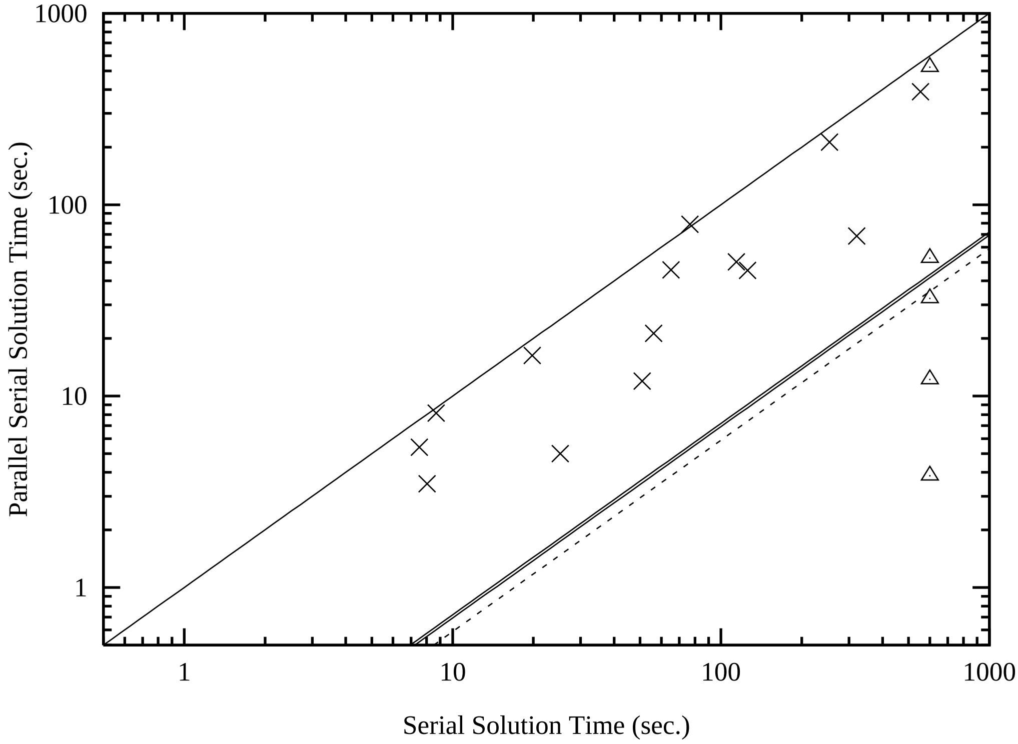


Figure 6.16: Performance of a 17-processor parallel system using eager pruning vs. a serial system, both based on a refined serial search. The parallel system is able to solve 5 more problem than the serial system.

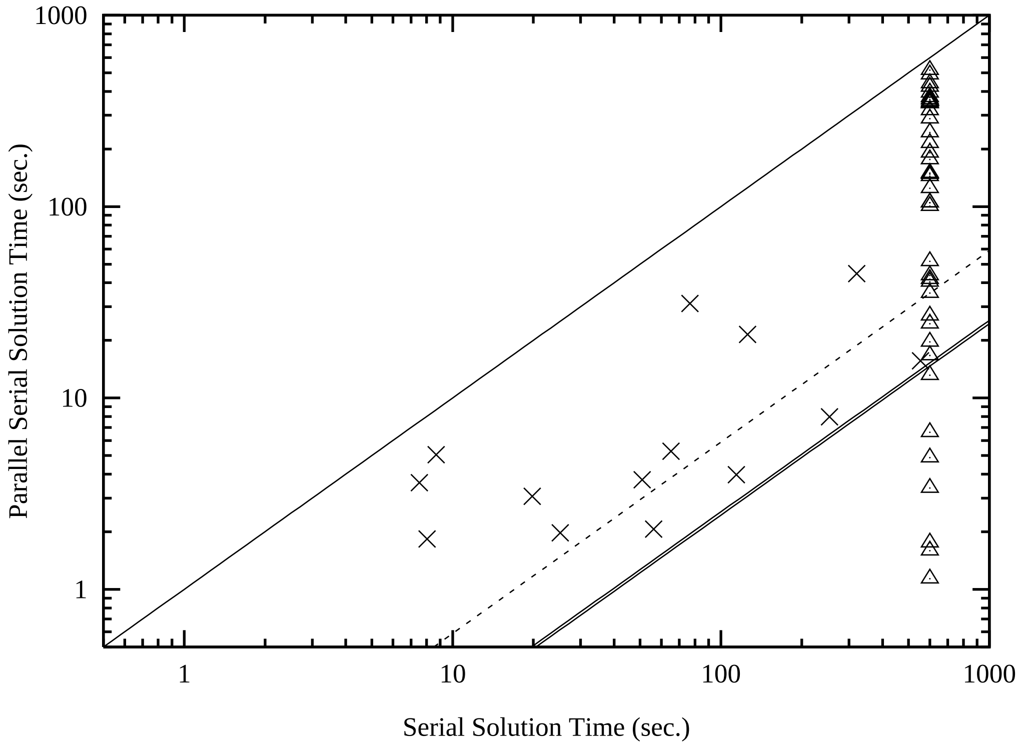


Figure 6.17: Performance of a 17-processor parallel system using all nagging extensions vs. a serial system, both based on a refined serial search. The parallel system is able to solve 39 more problem than the serial system in the 10 minute time limit.

protocol is extended to include infeasible-choice messages. The 5 additional problems solved by this system do not include any previously unsolved problems. However, the average speedup of 14.17 is an improvement over most other parallel systems in this section.

The parallel system of Figure 6.17 combines all nagging extensions introduced in Chapter 4. With all of these extensions, the parallel system is able to solve 39 more problems than its serial counterpart. This includes all problems solved by any of the previous systems, plus an additional 25 problems not solved by any other system. The average speedup of 40.13 is also impressive.

In terms of solved problems, this system with all nagging extensions in place vastly outperforms any of the previous systems. This is a bit surprising, because many of the nagging extensions that contributed to this result offered little or no performance improvement individually. This mirrors a result demonstrated with the serial refinements. Taken together, search refinements can facilitate greater performance improvement than their individual contributions would predict.

The TPTP provides a convenient basis for comparing the performance of DALI with that of similar search engines developed in other research efforts [SS95]. Generally, different theorem provers are designed to capitalize on different opportunities for search reduction; one system seldom demonstrates a uniform advantage over another. Consequently, even though DALI does not exhibit the greatest overall performance, it does manage to solve a number of problems that other tested systems do not. In particular, even on its subset of the TPTP, DALI solves more than 20 problems that the most effective tested system does not.

Although there are published results for a number of serial theorem proving systems against the TPTP, parallel systems are comparatively rare. Among parallel theorem provers DALI compares favorably to SPTHEO [SJ94]. SPTHEO is able to solve 39 percent of the TPTP problems while DALI solves 41 percent of the tested TPTP subset. Additionally, parallel DALI demonstrates a greater advantage over its serial version than does SPTHEO. While SPTHEO solves 20 percent more problems than its serial counterpart, SETHEO, the most sophisticated, parallel version of DALI solves 23 percent more problems than the most sophisticated serial version.

Chapter 7

Discussion

Because nagging is relevant to work in search pruning and in parallelism, it may be viewed in the context of several areas of research in search and automated deduction.

7.1 Parallel Inference

One of the most appealing qualities of a purely logical problem specification is that it describes computational content without placing unnecessary restrictions on how that computation should be carried out [Kow79]. As a rule, logical problem descriptions have very obvious parallel interpretations. Research in theorem proving and logic programming has identified many opportunities for exploiting this potential [Con87b,Kur90,FD90,dC84].

7.1.1 Low-Level Parallelization

Very often, a logical specification is open to parallelism at many levels of granularity. Correspondingly, target architectures for exploiting this parallelism range from specialized, tightly-coupled machines to networks of general-purpose workstations. Most fine-grained approaches concentrate on low-level operations that are ubiquitous and heavily used in deduction systems. The unification operation, for example, may account for as much as fifty percent of a logic program's total execution time [Sib90]. Speeding this operation alone can go a long way toward speeding search overall. To this end, various parallel implementations of unification have been proposed [Kac90]. Other research suggests that ordinary unification is a predominantly linear computation and that special variants such as associative, commutative unification are more amenable to parallelization [KW89].

Similar efforts have concentrated on other operations related to logical inference. For example, propositional assumption-based truth maintenance systems, commonly used to eliminate redundant avenues of search, are particularly suited to execution on the SIMD architecture of a connection machine [Dd88]. Likewise, Ibáñez presents a strategy for parallelizing component operations of a connection method theorem prover. This approach features parallel matching of complementary literals and parallel construction of all spanning sets [Ibá88].

These fine-grained techniques are particularly attractive because they do not fundamentally change search behavior. Furthermore, by parallelizing the basic operations used in every deduction problem, it may be possible to speed the solution of all such problems without requiring modification or annotation of their logical formulation. The hope is that fine-grained parallelism will offer substantial performance improvements that are uniformly beneficial to all problem instances. Unfortunately, even if uniform, this type of performance improvement may be inherently limited. If, for example, unification accounts for no more than fifty percent of a program's execution time, parallelizing unification can do no better than to double execution speed.

Of all parallelism schemes common in logical inference, these low-level approaches are, perhaps, the least similar to nagging. Most of the differences stem from differences in their intended granularity. There are, however, some common features.

- Exploiting fine-grained parallelism often requires specialized parallel hardware in which inter-process communication is comparatively quick. Nagging was designed to be feasible on a collection of general-purpose computers, where communication may be comparatively slow.

- The intent of many fine-grained parallelism techniques is to provide a uniform performance boost to all search problems. Nagging is built on the reasonable assumption that some problem formulations are better than others and its effectiveness is highly dependent on the quality of the default formulation. Thus, as Chapter 6 demonstrates, nagging provides a very nonuniform benefit across a population of search problems.

- Both nagging and these fine-grained techniques retain qualities of the underlying serial search, although the two techniques achieve this through different means. In nagging, similarity to the serial search is ultimately a consequence of the master process operating under the standard serial search procedure. In contrast, the influence of fine-grained parallelism techniques can be seen as independent of most notions of search order.

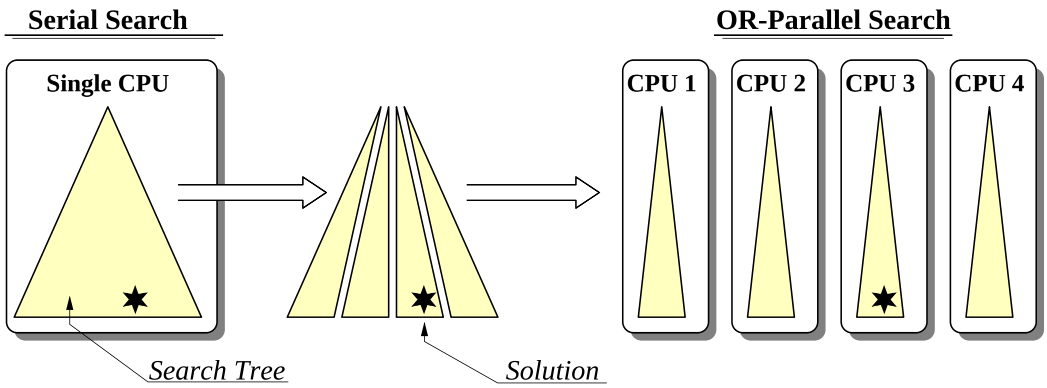


Figure 7.1: General model of OR-parallel search. Each participating process is charged with exploring a portion of the search tree.

7.1.2 OR Parallelism

OR parallelism is one of the most obvious and popular schemes of parallelizing search, and it is a technique that bears interesting relationships to nagging. Under OR parallelism, the work of exploring search tree T is divided among more than one search process. Each is given part of the search space in which to look for solutions. The intent is that, with several processes, each exploring different portions of T, one process will find a solution quickly.

Assigning portions of T to available processes depends on some mechanism for dividing the nodes of T into subsets $S_1, \ldots S_k$. Each of k available processes is assigned its own subset to explore. The mechanism by which subsets $S_1, \ldots S_k$ are selected distinguishes various, individual, OR-parallel techniques.

Most commonly, $S_1, \ldots S_k$ are chosen so that they are mutually disjoint. This ensures that OR-parallel processes do not duplicate search. Typical OR-parallel schemes partition T by dividing it at subtree boundaries (Figure 7.1) and letting each S_i be a resulting subtree. This is a convenient type of division since it guarantees a partition of T and permits a concise way of informing each process of its assigned subset of T. This arrangement is also conducive to efficient search of each S_i. Once given some $n_i \in T$, an OR-parallel process is free to explore its assigned search space, $T(n_i)$, without interruption. For depth-first search procedures (*e.g.* the WAM), the locality of nodes in each subtree is also consistent with the incremental maintenance of data structures.

This model of OR-parallelism is appealing because of its simplicity, but, alone, it is unrealistic for practical search problems. Breaking T at its subtrees affords a convenient and non-overlapping division of the search space but it does not guarantee uniformity. Without domain-specific knowledge, the division of a finite T is likely to produce subtrees of widely varying size. The bulk of the search space may be given to a small number of processes while others quickly complete their assigned subproblems and then sit idle. To avoid this, OR-parallelism schemes ordinarily provide some mechanism for redistributing the search as necessary. Many systems such as PARTHEO [SL90a,EKL+89], METEOR [AL91], and OPERA [BFGd92] employ a load-balancing mechanism that is driven by idle processes. When a process exhausts its search problem without finding a solution, it requests work from neighboring, busy processes. When requested, busy processes are obliged to share unexplored subtrees of their search with idle processes [DR92c]. Like nagging, this approach has the advantage of placing the responsibility for work distribution on idle rather than busy processes. A process that is busy does not have to divide its attention between performing search and managing parallelism.

Other implementations favor a more centralized approach to load balancing. Instead of assigning an entire subtree to each OR-parallel process, search is distributed only a few nodes at a time. OR-parallel processes obtain some number of nodes from a central facility, expand them and return the newly generated children to the central facility. These new nodes are then available for distribution to other processes. On a shared-memory architecture, each OR-parallel process is capable of directly accessing the pool of available work. Thus, each process may simply claim new search nodes as needed and insert generated nodes directly into the pool (*e.g.,* ROO [Lus90,SL90b], PARTHENON [ABCM88], ANL-WAM [DLO87], PIE [MOTA+84]). In a distributed implementation, a designated master process maintains an account of which parts of the search space have been explored. Master and slave exchange work assignments and results through message passing; the master simply tries to keep all slaves busy [LO88,Clo87]. Some interesting techniques have been developed to cope with the scalability problems inherent in this kind of centralized approach [BGR88]. In general, fine-grained OR-parallelism is most common where theorem proving is seen more as a process of generating T than of exploring it. On some systems, this approach has been very effectively combined with parallel mechanisms for detecting and filtering redundant portions of T [BFJO90].

Alternative OR-parallelism schemes completely avoid the need to redistribute work. In one such scheme, called competitive OR parallelism [Ert90], every process is permitted to explore the entire search tree T. Each explores T under a different search procedure or search order. Although this permits duplication of work, the intent is that each process will begin its search in a different part of T and, in so doing, one will find a solution quickly. Other systems, like SPTHEO [SJ94], try to insure that all processes stay busy by giving each a potentially infinite subtree to

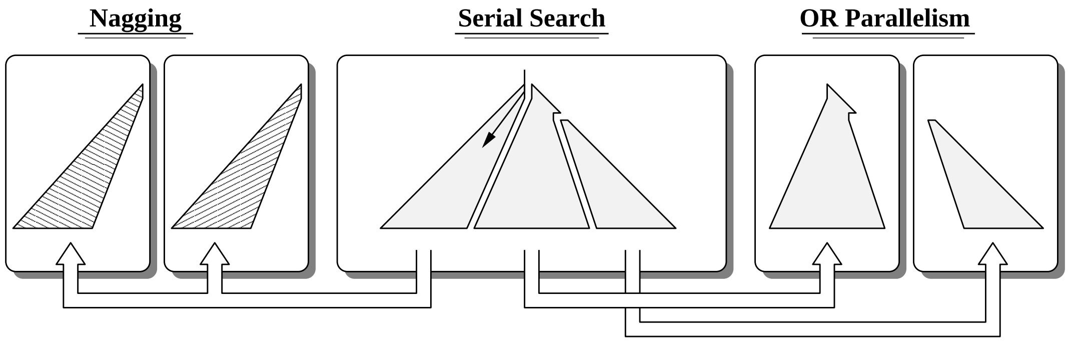


Figure 7.2: Comparison of OR parallelism and nagging. OR-parallel processes are permitted to skip ahead in the serial search order. Nagging attempts to cause the master itself to skip ahead in this order.

explore. For example, processes may explore their subtrees of T via independent iterative deepening disciplines. If each subtree is infinite, then some regions of T may be searched to a greater depth than others, but all processes will remain busy without the need to communicate with their peers.

OR parallelism features many desirable properties that make it quite popular in both theorem proving and logic programming. Unlike low-level parallelization schemes such as parallel unification, where performance improvement is inherently limited, the potential for OR parallelism grows with the search space. In the more coarse-grained models, processes are free to explore large components of T much as they would under a serial search. Thus, some implementations represent small changes to the underlying serial inference procedure and incur only a small amount of overhead. Shared memory architectures permit highly efficient implementations in which processes can share common data structures without mutual interference [DR92c,GJ90b,AK90,War87,LBD$^+$88,TL87,Cra85].

OR parallelism and nagging exhibit interesting similarities. Fundamentally, each is designed to contend with the nondeterministic nature of search. Faced with nondeterminism, any serial search procedure imposes a particular order on the way candidate solutions are generated and checked against the solution criteria. Of course, it is desirable that actual solution nodes occur early in this ordering. Heuristic guidance can help to insure that solutions are reached early in some cases, but, for legitimate search problems, there will always be situations in which the serial search is ordered poorly, such that many non-solution nodes will be examined before the first solution is reached. Both nagging and OR parallelism can be seen as measures against the possibility that the serial search is poorly ordered. Consider Figure 7.2. OR parallelism can be seen as an attempt to overcome a poor search order by taking regions of T that the serial search would not reach until much later and promoting them so that they are examined early by parallel processes. OR-parallel processes are permitted to skip ahead in the search order in the hope that they will skip past large, fruitless regions of T. Nagging can be seen an effort to permit the serial search itself to skip past futile regions of the search space. While the work of an OR-parallel process may be seen as an attempt to show that solutions lie elsewhere in the search, the work of a nagger may be seen as an attempt to quickly show that no solutions lie in the portion of T that is currently being explored.

Comparisons between nagging and competitive OR parallelism are particularly notable. Under both models, processes compete to explore related search spaces using different search orderings or search procedures. 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. Consequently, the focus of nagging is to use transformed search problems to quickly identify regions of the search space that are void of solutions, while competitive OR parallelism is concerned with using alternative search disciplines to quickly identify solutions.

Largely because of their similar intent, nagging and OR parallelism share many features.

- Since both schemes attempt to compensate for a suboptimal ordering of the serial search, they are less effective when the search is already well-ordered. In the extreme case, both techniques would be completely ineffective at speeding the execution of a deterministic logic program. Likewise, if the search space is rich in solutions, both techniques may be equally unproductive.

- Figure 7.2 suggests the similar granularity at which nagging and some forms of OR parallelism are intended to operate. Both are expected to use large subtrees of T as the basis for distributing work. Both schemes are designed to permit parallel processes to operate independently most of the time.

- Both nagging and OR parallelism permit load balancing to be performed by idle rather than busy processes.

Nagging and OR parallelism also differ in a number of respects. OR parallelism, for example, requires no domain-specific knowledge. Although heuristic information about the shape of T may be helpful in dividing the search evenly, such knowledge is not strictly necessary. In contrast, nagging requires a domain-specific problem transformation function. The fairly general function classes, $\mathcal{P}$ and $\mathcal{A}$, reduce this dependence somewhat since the expressive power of first-order logic makes them applicable to a large class of problems, but requirements such as the presence of AND branching needed by $\mathcal{P}$ demonstrate that some domain-specific dependencies remain.

This disadvantage is balanced with properties of nagging that are not shared by OR parallelism. Since OR parallelism lets some processes skip ahead to later portions of the search, it effectively changes the search order. The order in which solutions are discovered under OR parallelism may not match that of a serial search. Although this compromise of the serial search order may be of little importance when proving theorems, it is more inconvenient in logic programming where there may be an interest in retaining serial semantics under parallel execution. This is evident in the interaction between OR parallelism and Prolog's cut operator [Hau90,LBD+88, HCC88]. Also, OR-parallel computation models commonly lack the fault tolerant properties inherent in nagging. If T is partitioned among OR-parallel processes and one such process fails, solutions contained in the failed process' partition of T may not be discovered.

7.1.3 AND Parallelism

While OR parallelism involves working in more than one branch of the search space concurrently, a dual approach known as AND parallelism involves working in more than one branch of the tableau concurrently. Under OR parallelism, the nodes of the search tree are distributed among two or more processes; under AND parallelism, it is the open branches of the tableau that are distributed among participating processes. Figure 7.3 illustrates the operation of AND parallelism. For some partial proof P, each AND-parallel process is given a subset B_i of the open branches in $P\Delta_{\emptyset}$. Each process attempts to find subproofs for its assigned B_i, and, if all of them succeed, it may be possible to compose the subproofs they find into a single proof. Failure of any AND-parallel process to close its branches may be taken as a demonstration that P cannot be lengthened to exhibit a closed tableau. In the event of such a failure, all work by sibling AND-parallel processes may be abandoned, P may be discarded, and the search may continue with some other partial proof.

Under serial search, open branches must be closed one after another, and closing each branch may represent a significant amount of computation. AND-parallelism seeks to speed the construction of a proof by trying to close several tableau branches in parallel. While the potential for OR parallelism grows with the branching of the search space, available AND parallelism is proportional to the branching of the tableau. Because of this, it has attracted more attention in the context of logic programming than in automated deduction. Some of this bias stems from the dif-

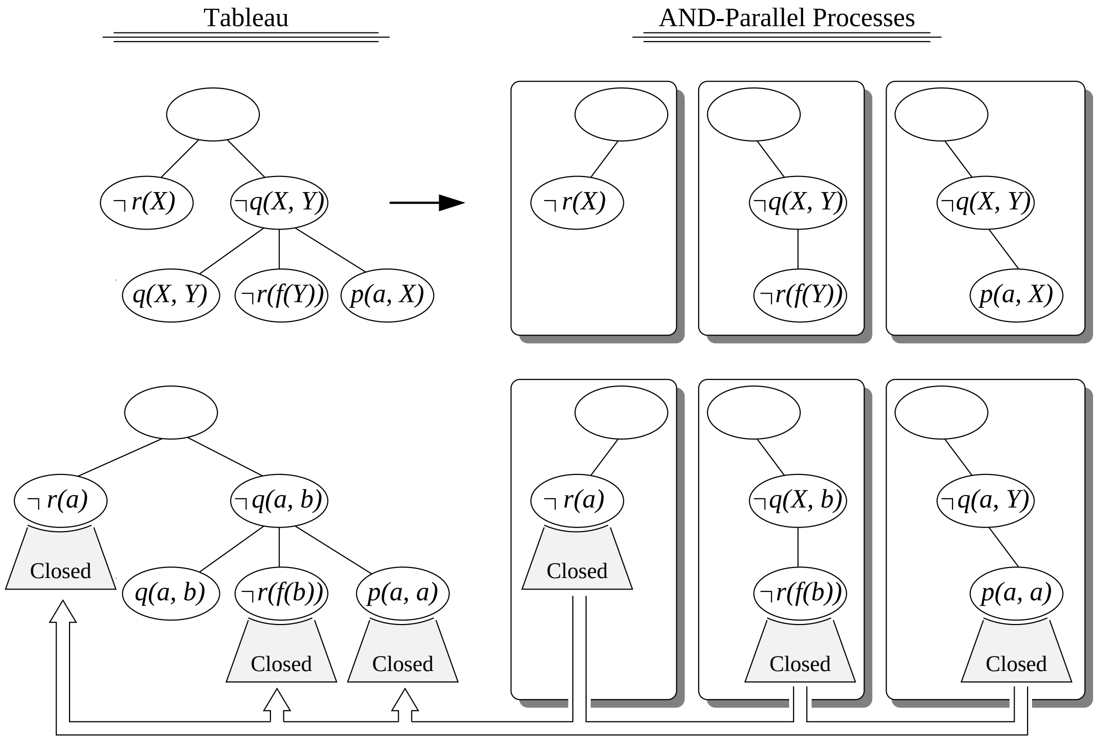


Figure 7.3: Model of AND-parallel search. Each AND-parallel process attempts to close a subset of the tableau branches. If all are successful, it may be possible to compose their solutions into a consistent proof.

ference in problems addressed by these two communities. Logic programming places emphasis on a domain theory's procedural interpretation. Each open branch can be seen as a procedure call, and the inference operations that close a branch represent the execution of a procedure. Thus, AND-parallelism is analogous to evaluating the steps of a computation in parallel, and is comparable to more conventional models of parallelism used in deterministic algorithms.

Since logic programming applications are most often concerned with the variable bindings made in a proof and not with the content of the proof itself, once all AND-parallel processes find their subproofs, composing them requires only the integration of their variable bindings and not the combination of their subproofs. This enables what is perhaps the most attractive feature of AND-parallelism; unlike OR parallelism, it can reduce execution time even in completely deterministic domains.

There are a number of other reasons AND parallelism is favored over OR parallelism for some applications. There is some evidence that OR-parallelism is not particularly effective when only one solution is desired [FD90]. Furthermore, some exclusively OR-parallel approaches have been criticized because they do not permit cooperation between parallel searches [GS88]. In principle, OR-parallel processes operate independently. Under AND parallelism, however, the subproblems held by different processes may be fundamentally interdependent.

It is precisely this interdependence that complicates AND parallelism. Even if all processes are able to close their assigned branches of the tableau, the subproofs they find will not necessarily be composable. Subproofs found by each AND-parallel process may bind variables in the tableau. In Figure 7.3, these bindings do not conflict; process one and three bind the variable X, but both bind it to the constant a. There is, of course, no guarantee the variable bindings made in each subproof will agree. Even if there exist subproofs for all open branches such that variables are bound consistently, these may not be the subproofs that are found first under AND-parallelism. To cope with these potential variable binding conflicts, various forms of AND-parallelism have been developed.

7.1.3.1 Restricted AND Parallelism

Under *Independent* or *Restricted AND Parallelism* (RAP), parallelism is only permitted when there is no risk of processes generating proofs with conflicting variable bindings. As a consequence, RAP is not capable of exploiting all of the potential AND parallelism.

RAP systems avoid binding conflicts by closing branches in parallel only when they share no unbound variables. The restriction to Horn-clause theories, common among AND-parallel systems, simplifies the condition for parallel execution. In tableaux built from Horn-clause theories, positive literals appear only as the leaves of closed branches. Since reduction requires both a positive and a negative literal in an open branch, reduction can never occur in such a tableau. A subproof for open branch β can only bind variables occurring in $\mu(\mathit{leaf}(\beta))$, and the condition for restricted AND-parallel execution requires only that the leaves of open branches be variable disjoint.

There are two fundamental approaches to enforcing this condition for restricted AND-parallel execution, dynamic and static. Under the dynamic approach, potential AND parallelism is identified during search [LK88,Con87b,Con87a]. Each open branch is assigned to its own process, but processes may not be permitted to start searching for subproofs immediately. If the leaves of two or more open branches share an unbound variable, only one of the corresponding processes is allowed to bind the variable. This process is called the *generator* of bindings for the variable. Other processes must receive a variable's bindings from that variable's generator and must wait for the appropriate generators to complete their subproofs before beginning search. If a generator binds each of its variables to a ground term, processes dependent on these bindings may be able to execute in parallel. Otherwise, a new generator must be appointed to supply values for remaining unbound variables in the generated term.

Because of their sensitivity to the run-time variable binding information, dynamic RAP approaches are capable of exploiting much of the available AND-parallelism. However, this attention to process interdependence incurs significant run-time overhead [FD90]. The static approach to RAP represents an effort to reduce this overhead, typically at the expense of a reduction in parallelism. Under the static approach, all opportunities for AND-parallelism are identified before search begins [Her86,HN86,CDD85]. Branches that are likely to be variable-disjoint are identified through compile-time analysis of the domain theory. At run time, only a simple check is necessary to verify independence. Communication overhead is also lower than in the dynamic case. Dynamic RAP requires variable bindings to be communicated from generator to consumer. Under the static approach, the leaves of AND-parallel branches share no unbound variables, so it is not necessary to exchange bindings between sibling AND-parallel processes.

Since the static approach exploits only the restricted AND-parallelism that can be detected at compile time, some available parallelism is typically not realized. Hybrid techniques have emerged as an attempt to capture the flexibility of the dynamic model while maintaining the efficiency of the static model [DeG84,DR92a]. Under the dynamic model, the AND-parallel execution strategy is continuously updated in response to the occurrences of unbound variables in open branches. Hybrid techniques identify a fixed number of feasible AND-parallel execution strategies through compile-time analysis. At run-time, one such strategy is selected based on a single inspection of the variable binding structure. In this way, it may be possible to approximate the dynamic AND-parallel execution while consulting run-time binding information no more often than in the static model.

As a side-effect, implementations of RAP often realize some component of intelligent backtracking [GBJ+90,LK88,Con87b,LKL86,VG92]. Although this effect is more evident in the dynamic models, it is present in most implementations, whether static or dynamic. Restricted AND parallelism requires that the interdependence of open branches be identified in the form of variables shared among those branches. It is often straightforward or even necessary to consult this interdependence information when backtracking. Backtracking may be permitted to effectively skip some choice points when their reconsideration would not avoid future failure. Figure 7.4 illustrates this phenomenon. Under a serial search discipline, open branches may be closed one after another. When the serial search encounters a branch that it cannot close, chronological backtracking demands reconsideration of the most recent choice point with untried alternatives. In Figure 7.4, for example, this means that, when no subproof can be found for the branch labeled $p(a,c)$, alternative subproofs for the branch $r(f(Z))$ are considered. Naturally, this is futile since subproofs for $r(f(Z))$ cannot change the labeling of any nodes on the failed branch, and, therefore, even if alternative subproofs for $r(f(Z))$ exist, the $p(a,c)$ branch will continue to fail. Under RAP, subproofs for $r(f(Z))$ and $p(a,c)$ may be sought in parallel because the closing of the leftmost branch leaves them variable disjoint. When *Process 2* determines that $p(a,c)$ cannot be closed, it prompts backtracking to consider alternative subproofs for $q(X,Y)$ and forces *Process 1* to abandon its search whether or not it has found a solution. Effectively, the failure to close $p(a,c)$ results backtracking past $r(f(Z))$, much like what is achieved through intelligent backtracking.

7.1.3.2 Pipeline AND Parallelism

In *pipeline AND parallelism*, branches of the tableau are closed one after another and in the same order as the underlying serial search [CFS89,Tan86,DeG84]. An open branch is given to each process. Initially, only the process holding the leftmost branch is permitted to search. If this process finds a subproof, it transmits any variable bindings made in that subproof to the process holding the next sequential branch. Upon receiving a set of variable bindings a process searches for a subproof of its branch that is consistent with those bindings. If successful, it forwards those bindings to the next process along with any new bindings made in its subproof. As shown in Figure 7.5, parallelism is achieved by permitting each process to continue

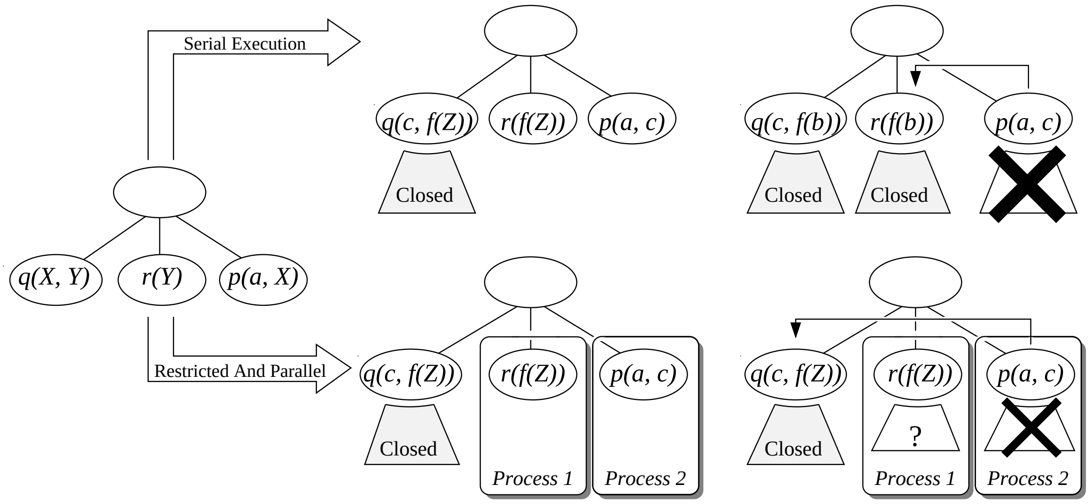


Figure 7.4: Intelligent backtracking effects of restricted AND parallelism. Failure of one and-parallel process to find a solution may prompt backtracking and cause other and-parallel processes to abandon their work. This is analogous to skipping choice points during backtracking.

search for alternative solutions immediately after an initial subproof has been found. Once a process finds all subproofs of its branch under a particular set of bindings, it awaits alternative bindings from its predecessor.

Pipeline AND parallelism avoids potential variable binding conflicts by imposing a strict order on processes. Each process is only permitted to construct subproofs that are composable with subproofs found by all of its predecessors; when the last process finds a solution, a consistent set of subproofs for each open branch will have been identified. Pipeline AND parallelism is designed around the assumption that search will frequently necessitate backtracking to consider alternative subproofs for tableau branches. Instead of waiting for failure before generating alternative solutions, this technique mandates generating alternative subproofs before it is certain that they will be needed. When a failure is encountered, construction of the next potential solution will have already begun.

Although pipeline AND parallelism involves concurrent attempts to close more than one branch of the tableau, it is otherwise more akin to OR- than AND-parallel techniques. Like OR parallelism, pipeline AND parallelism provides no performance improvement in deterministic domains or if backtracking never occurs [ACF92]. There is a direct correspondence between nodes expanded under pipeline AND parallelism and those expanded under the underlying serial search. As a result, pipeline AND parallelism is equivalent to an OR-parallel search in which search nodes are

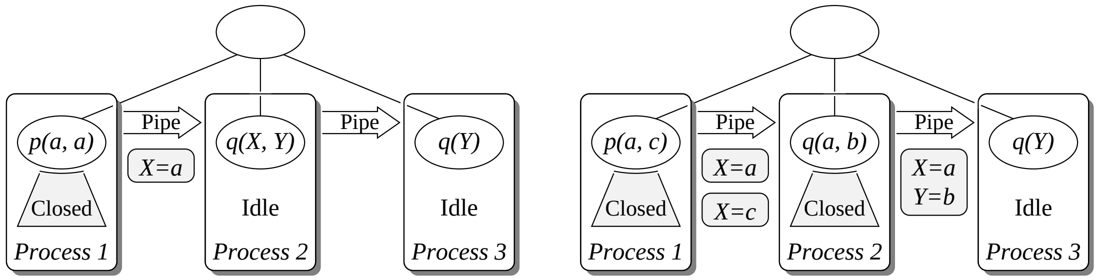


Figure 7.5: Pipeline AND-parallel search model. Each process enumerates subproofs for its assigned branch. It forwards the variable bindings made in each subproof to the next process in the pipe.

assigned to different processes based on the output of the branch selection function.

7.1.3.3 Stream AND Parallelism

For many models of AND parallelism, a variable shared among more than one open branch is problematic. *Stream AND parallelism* treats these shared variables, instead, as channels through which processes may communicate and cooperatively solve their respective search problems. Other AND-parallelism schemes exchange information about variable bindings only as complete subproofs are found [TF86]. Under stream AND parallelism, a process publicizes its variable bindings as soon as they are made. If a shared variable is bound to a non-ground term, then any subsequent bindings that instantiate that term are similarly broadcast to other processes sensitive to that variable binding. As in the dynamic forms of RAP, most stream AND-parallel models appoint one process as the generator of bindings for each shared variable. When an appointed process binds one of its shared variables, the binding is communicated to other processes holding branches featuring that variable. Processes that are not permitted to bind a shared variable may begin search immediately; they simply block when they require a binding that they are not permitted to make and wait for the generator to supply one.

Figure 7.6 demonstrates the operation of stream AND parallelism. *Process 1* is permitted to bind the variable X and *Process 2* is permitted to bind Y. When *Process 1* binds X to $f(Z)$, it communicates this binding to *Process 2* and *Process 3*. After *Process 2* receives the binding of X, it generates a binding of Y, which it shares with the third process. As the first process continues to instantiate X, it forwards these bindings to the other processes.

Although the operation of forward execution is straightforward, it is the response to failure and the backtracking policy that most significantly complicates stream

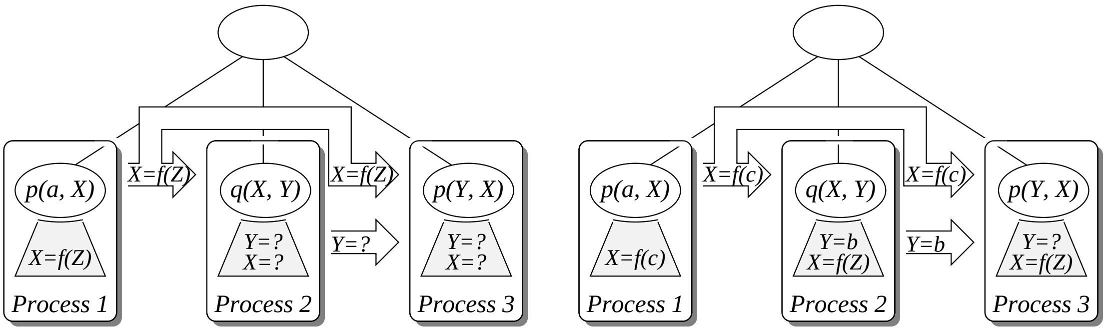


Figure 7.6: Stream AND-parallel computation model. Processes share variable binding information even before they have a complete subproof. As non-ground terms are instantiated, the additional bindings are communicated.

AND-parallelism [Nai88]. If one process encounters a failure, it may impact not only the process discovering the failure but also any other processes that share logic variables with it. This may transitively impact the search of still other processes. In Figure 7.6, if *Process 1* cannot complete its proof under the binding $X = f(c)$, it must backtrack and consider an alternative binding of X. This will cause the other processes to interrupt their search and return to a state before they first made use of the binding for X. Parallel backtracking in this framework may necessitate retraction of work on a large number of processors and may entail high communication overhead [SRV88,PMCA86].

As a result of its unwieldy backtracking requirements, stream AND parallelism is principally exploited in committed choice frameworks, where backtracking is not required and search completeness is not guaranteed [dC84,YA86,OM87]. A number of *concurrent logic languages* have emerged to take advantage of this computation model [CCC90,DR92b,TK86,Lev86,PN84,VY84,UC90,Ter88,SKL90,TF87]. Although they differ in terms of their details, most of these languages are superficially quite similar [Sha89,CG81,Sha87b,CG86,CG87,Ued87,Sha87a,Sar87,TF86, dP90]. They feature Prolog's logical orientation but not its control regimen. Typically, the programmer is responsible for identifying potential parallelism within the logic, indicating which processes may generate bindings for each variable [Sar88] and ensuring that all desired solutions are found within a limited backtracking framework.

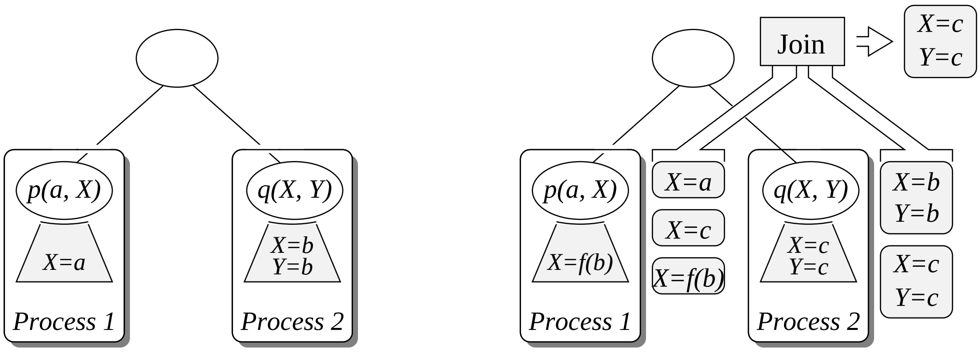


Figure 7.7: Parallel join search model. Processes enumerate all solutions for their branches and a relational join on these solutions filters inconsistent subproofs.

7.1.3.4 Parallel Join

Most approaches to AND parallelism adopt an eager policy of ensuring that AND-parallel solutions are composable. When there is a possibility for the generation of conflicting bindings, most policies either permit processes to communicate so that their bindings agree or forego parallel execution. In contrast, *parallel join* enforces a lazy policy of resolving variable binding conflicts. Open branches are distributed among available processes. Each process is permitted to bind variables as necessary without attention to the possibility that other processes may make incompatible bindings. As in pipeline AND parallelism, processes continue to search for subproofs even after an initial subproof has been found. As subproofs are found, a relational join is computed on their variable bindings. Inconsistent bindings are filtered automatically through the join operation [KR90]. Figure 7.7 illustrates the operation of the parallel join.

The parallel join permits processes to operate in complete independence. As in OR parallelism, this is conducive to efficient implementation of the search engine. It can also avoid some pathological instances of backtracking and redundant computation common among sequential search procedures. As a consequence, however, it has the potential to dramatically increase the size of the search space in some cases [DeG84]. In Figure 7.7, one process is assigned to find subproofs for the $q(X, Y)$ branch. Under a serial search $q(X, Y)$ would not be attempted until a subproof for $p(a, X)$ had been found. Since subproofs for $p(a, X)$ may bind the variable X, the serial search may only have to prove $q(X, Y)$ for particular values of X. Search under parallel join may attempt to solve $q(X, Y)$ for values of X that would never be considered under the serial search. To avoid this, many implementations apply parallel join only where restricted AND parallelism would be applicable [DR92a, KR90,YCS90].

7.1.3.5 Relationship with Nagging

Nagging was originally developed as a variant of AND-parallel techniques, and, although its most general formulation is probably more similar to OR parallelism, nagging under $\mathcal{P}$ is easily compared to AND-parallel techniques. The most apparent similarity is that both require nontrivial branching of the tableau. Like AND-parallel processes, naggers working under $\mathcal{P}$ consider subsets of the open branches. Failure of any one AND-parallel process to find a subproof is sufficient to justify abandoning the current search path, just as failure by any nagger is sufficient to justify backtracking on the master process. The principal differences between nagging and AND-parallelism stem from a difference in intent. Whereas the goal of AND parallelism is to find a consistent set of subproofs for all open branches, the goal of nagging is to demonstrate that no such subproof exists. Thus, while AND parallelism must negotiate the difficulties of insuring that parallel-derived subproofs are consistent, nagging deliberately permits nagging processes to operate independently and to disagree in their binding of shared variables. When nagging processes fail to close their branches, these failures are not qualified with respect to variable bindings made by sibling naggers, and the master's search can be pruned immediately.

One of most obvious ways in which nagging under $\mathcal{P}$ differs from AND parallelism is in the division of tableau branches among available processors. While AND parallelism typically partitions the set of open branches among participating processors, the subsets of branches examined by naggers may overlap with each other and will always overlap with the master. Because of this overlap, even when we permit the use of nagger-derived, local subproofs, nagging retains its fault tolerant properties. This type of fault tolerance is more difficult to achieve in conventional AND-parallel techniques, where the loss of a processor may mean that parts of the tableau will not be closed.

As illustrated in Figure 4.7 of Chapter 4, exploiting local subproofs found by a nagging process actually affords a component of AND parallelism. If nagging processes are able to identify local subproofs, then it is guaranteed that they can be composed. However, the circumstances under which nagging will compose subproofs differ greatly from conventional AND-parallel techniques. In particular, nagging only composes parallel-derived subproofs that satisfy the locality conditions. As a result, nagging exploits only a restricted form of AND parallelism but a form that differs from true RAP. The conservative nature of RAP requires that AND parallelism be attempted only when the resulting subproofs are certain to be weakly local. Nagging, in contrast, employs much weaker preconditions for the parallelization of tableau branches. If weakly local subproofs are found they can be exploited, but DALI makes no deliberate attempt to encourage them. This distinction between *a priori* and *a posteriori* assurance of weak locality is reflective of a difference in the intended domains of these two techniques. Most of the work in RAP is targeted toward the evaluation of Horn-clause theories, where all potential variable binding conflicts can be detected by examining the leaf node of each branch. This makes it both possible to identify independent branches at the clause level through compile-time theory analysis and reasonably efficient to identify them during search. When using model elimination on first-order theories, however, shared variables may occur anywhere in the branch, making the possibility of variable-disjoint subproblems more remote and making tests for branch independence more expensive.

Naturally, this difference in policy has consequences for the particular situations in which nagging and RAP can exploit AND parallelism. Figure 7.8 illustrates this, showing a particular tableau along with two of its possible derivatives. If the leftmost tableau branch is closed with a subproof that binds the variable Y to $f(X)$, then the two remaining branches are not candidates for Restricted AND Parallelism; they share the variable X. Independently derived subproofs for these two branches are not guaranteed to be composable. However, a nagger working under $\mathcal{P}$ would be permitted to search for subproofs for $p(c, f(X))$ while the master attempts to close the $q(X, b)$ branch. If the nagger discovers a subproof for $p(c, f(X))$ that does not bind X, then it has found a local subproof, effectively permitting AND parallelism. In this case, the particular subproof found by the nagger is composable with any subproofs found by the master, even though the variable sharing structure of the rest of the tableau did not guarantee that it would be.

Of course, there are situations in which RAP will extract AND parallelism that nagging will overlook. If the leftmost tableau branch is closed with a subproof that binds Y to the term a, then the two remaining branches are made variable disjoint and are candidates for RAP. However, under the prototype permutation transformation given in Chapter 6, a nagger would not be permitted to try to close $p(c, Y)$ while the master works on $p(c, a)$. In this case, the *Failure Potential Rule* forbids naggers from considering the $p(c, Y)$ branch alone since its label is unchanged by the closing of the leftmost branch.

7.1.4 Hybrid Systems

Exploiting one source of parallelism does not necessarily preclude the use of some other parallelism scheme. Many implementations take advantage of this by employing more than one source of parallelism in cooperation. The model elimination and

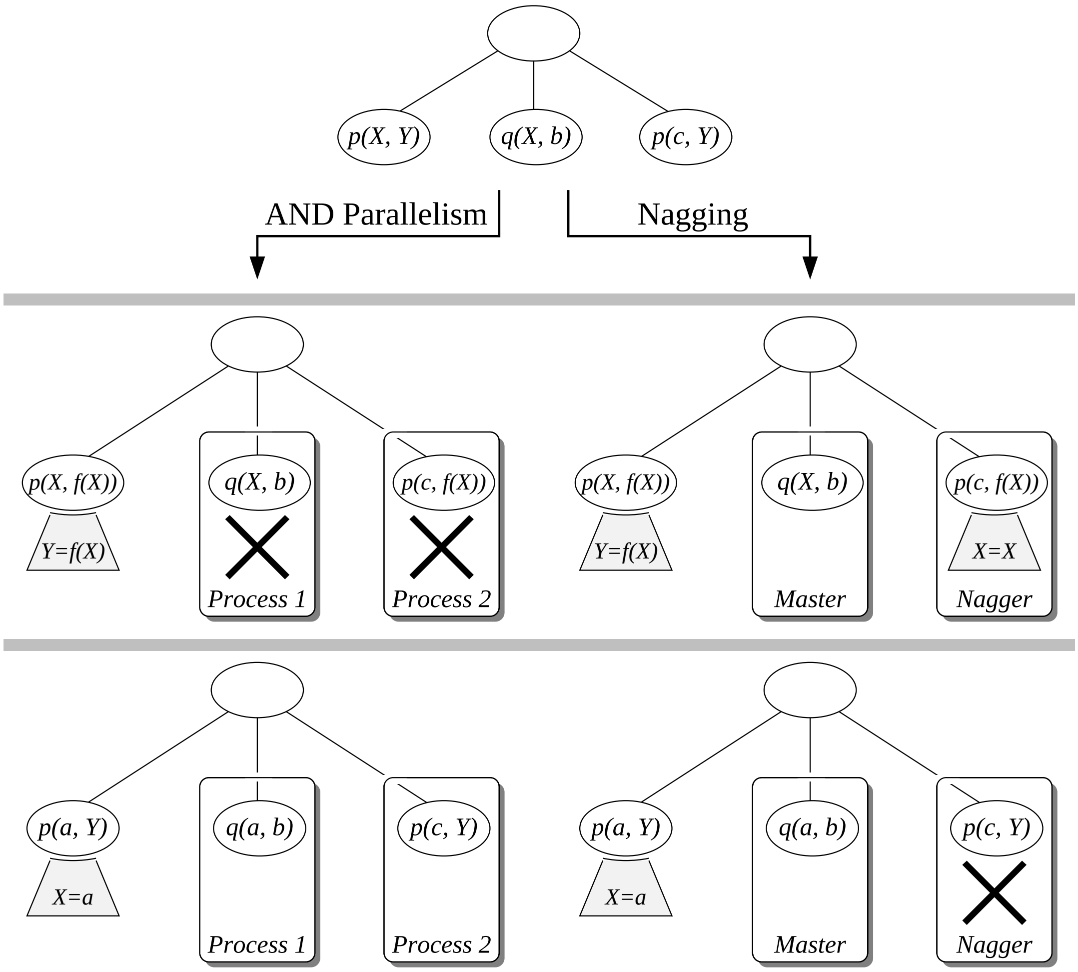


Figure 7.8: Differing potential for AND parallelism in RAP and nagging. This figure illustrates two possible derivatives of a single tableau shown at the top. The upper derivative forbids RAP but permits AND parallelism via nagging. The lower permits RAP but not AND parallelism as a side effect of nagging.

SLD resolution search trees have a natural parallel interpretation at both the AND and OR branches. Many approaches to parallel inference exploit both of these by combining AND- and OR-parallel components. Conery describes a natural mapping between the AND/OR search tree and a tree of communicating processes [Con87b, Con92]. This mapping applies AND and OR parallelism alternately at each ply of the AND/OR tree and has been used in a number of systems as the basis for combining AND- and OR-parallel techniques. In principle, many concurrent logic languages provide for OR parallelism in addition to their more prevalent stream AND parallelism [TF86]. Other systems have have coupled OR parallelism with other major forms of AND parallelism, with parallel join and pipeline AND parallelism being particularly popular [Kac92,CFS89,ACF92,PSS92,BdH+88,GBJ+90, GJ90a,HA84,BT88,Kal87,KR92]. This dual approach makes it possible to extract a large amount of the parallelism inherent in the logic and provides some insulation against problems that are not amenable to a particular variety of parallelism [YCS90].

In some approaches, OR parallelism is seen as a means of imparting completeness to an otherwise incomplete AND-parallel search strategy [HN86,Bre88]. The definition of a correct backward execution policy is sometimes subtle and elusive. Exploring alternative search branches through OR parallelism rather than backtracking obviates the need for a backward execution mechanism.

Implementation of either AND or OR parallelism entails some overhead on the search engine. To balance this overhead, some implementations include mechanisms for fine-grained parallelism along with the more coarse-grained AND or OR parallelism. By employing a collection of specialized inference processors, it may be possible to overlap the local operations necessitated by a particular brand of AND or OR parallelism. This can greatly reduce the parallelization overhead on each processor and can facilitate inference speeds comparable or superior to those of an efficient serial implementation [HG88,GSC+90,SYH+87,ALS90,AS88,Tan86,MKO84,TM87].

7.2 Serial Search Reduction

Nagging may also be compared to a number of serial search reduction techniques that direct search through some notion of problem transformation. Like nagging, many of these techniques attempt to focus search in the original domain by considering, instead, related search problems in simpler domains. These alternative problems have some relevance to the original domain but typically have more favorable computational characteristics.

Many *abstraction* and *hierarchical problem-solving* techniques use the transformed problem as a guide for solving the original [Pla81,Kno94,BY94,SC95,GW92]. Before attempting to solve a given problem, they generate an alternative problem that factors out some of the details of the original formulation. The intent is that search on behalf of this alternative problem will be more effective or even tractable. If a solution to the alternative problem can be found, it may be possible to use it as a template for constructing a solution in the original domain. This provision for coercing solutions of the alternative problem into solutions to the original is the most significant feature distinguishing nagging from these other techniques. Nagging attempts, primarily, to capitalize on failures in the alternative search space, not on successes. While much of the work on these techniques concentrates on making guarantees about how solutions of the transformed problem relate to solutions of the original, nagging requires no such guarantees. In spite of this difference, there are similarities between these two techniques. Sections 4.2.1 and 4.2.2 identify relationships within original and abstracted search spaces and mappings between solutions in these spaces. Thus, nagging can sometimes use solutions of the transformed problem to constrain or solve the original. Furthermore, both techniques can benefit from recursive application. The abstraction mechanism used to govern search in the original problem can naturally be used to constrain search in the abstracted problems themselves.

Other approaches focus on problem abstractions that can be used as computationally efficient approximations of the original [Brü94,SK91,KS94,KKS95]. Typically, the approximation is consulted first and, if it is sufficient to solve the problems, actual search can be avoided. Since the abstracted problem is guaranteed to be polynomially computable, it makes sense to consult it before resorting to exponential search. In contrast, the problem transformations used in nagging make no such guarantees about computational efficiency. Although Figure 3.5 demonstrates that nagging can eliminate exponential search, it is not generally expected to do so. Thus, it would not make sense to use a nagging-generated transformation as a predictor for the original since the transformed problem might be more difficult to solve than the original.

7.3 Conclusion

Chapter 6 demonstrates that nagging, even in a fairly naive form, can substantially reduce search on a number of first-order logic problems. As anticipated, its effectiveness varies from one problem instance to the next. However, even though performance improvement is not guaranteed by its formal properties, nagging does succeed in reducing search time on most nontrivial problems. This is apparent in a visual inspection of the plots in Chapter 6 and is supported by the simple analysis used to compare different nagging systems.

Previous sections explicate the similarities between nagging and other search-reduction mechanisms, both serial and parallel. Chapter 5 highlights the many potential conflicts between nagging and DALI's serial search refinements. In spite of the possibility of overlap and interference, however, nagging greatly enhances the performance of even a sophisticated theorem prover. It appears to capture different opportunities for search reduction than DALI's serial search refinements. Of course, this is exactly the type of performance improvement that one would want from a parallelism scheme. If the intent is to solve larger, more difficult problems, parallel techniques that duplicate search reduction and performance improvement available through known serial techniques are of dubious value.

Chapters 1 and 2 introduce nagging and the assumptions on which it is based in the context of a fairly general notion of search. The nagging properties and many of the principals of its implementation in DALI are clearly applicable to other search problems. Empirical results suggest that nagging can be effective even when a number of powerful serial search reduction mechanisms are in place. If these results on the first-order nagging are representative, then this technique has the potential to reduce search and increase the horizon of feasibly solvable problems in a number of interesting domains.

Appendix A

Proofs

Theorem 2.2 During a search of T, if $n \in T$ has already been expanded, then deleting all nodes in $T(n)$ from the fringe prevents further exploration of $T(n)$.

Proof: It can be shown that any node $n' \in T$ has at most one representative $p^j(n')$ on the fringe at any point in the search. Search begins with exactly one node on the fringe. During search, the insertion of any $p^i(n'), i \geq 0$ into the fringe is accompanied by the removal of $p^{i+1}(n')$. Thus, any node n' can have no more than one $p^j(n')$ on the fringe at any time. Removing the nodes in $T(n)$ from the fringe eliminates this single representative for each $n' \in T(n)$.

It can be shown that, if neither n' nor any of its ancestors are on the fringe, then n' cannot subsequently be expanded during the search. Assume that neither n' nor any of its ancestors are on the fringe after $i \geq 0$ node expansions but that some $p^j(n'), j \geq 0$ appears on the fringe after the $i+1^{st}$ node expansion. Since $p^j(n')$, was not on the fringe after the i^{th} node was expanded, it must have been generated in the $i+1^{st}$ node expansion. Expansion of a node generates children of that node only, so $p^{j+1}(n')$ must have been on the fringe after the i^{th} node expansion. This contradicts the assumption that no such node was on the fringe at that point in the search. Therefore, if neither n' nor any of its ancestors are on the fringe at some point in the search, none of these nodes can enter the fringe or be expanded later in the search. $\square$

Lemma 2.4 For any search tree T and any serial and nagged exploration of T under a myopic search procedure, there exists a nondecreasing function $g : \mathcal{N} \rightarrow \mathcal{N}$ that satisfies the following conditions:

1. Let F be the set of nodes on the fringe after i nodes are expanded under the serial search. If F' is the set of nodes on the fringe after the master expands $g(i)$ nodes under the nagged search then $F' \subseteq F$.

2. If node n is expanded both by the serial search and by the master under the nagged search, then $\exists i \in \mathcal{N}$ such that n is the i^{th} node expanded in the serial search and the $g(i)^{th}$ node expanded under the nagged search.
3. $g(i) \leq i$.

Proof: Proof is by induction on the number of nodes expanded under a serial search. Before any nodes have been expanded under the serial search, then the fringe contains only the root of T. Initially, the master's fringe under a nagging search also contains only the root node. Thus, $g(0) = 0$.

Let F_i be the fringe after the i^{th} node expansion under the serial search and, for the nagging search, let $F'_{g(i)}$ be the fringe on the master process after it has expanded $g(i)$ nodes and processed all pending messages. Assume that $F'_{g(i)} \subseteq F_i$ and $g(i) \leq i$, and let F_{i+1} be the fringe reached by the serial search after $i+1$ nodes are expanded. Let n_{i+1} be the $i+1^{st}$ node expanded by the serial search. Expansion removes n_{i+1} from the fringe, possibly replacing it by some of its newly generated children. Search reduction mechanisms may allow some of $c(n_{i+1})$ to be omitted when expanding n_{i+1} just as, in Chapter 1, it was permissible to skip some infeasible board configurations when exploring the N-queens search tree. Thus $F_{i+1} = F_i - \{n_{i+1}\} + C$ where $C \subseteq c(n_{i+1})$. In identifying $g(i+1)$ two cases must be considered:

- If $n_{i+1} \notin F'_{g(i)}$ then, $F'_{g(i)} \subseteq F_i - \{n_{i+1}\}$. Since $F_{i+1} = (F_i - \{n_{i+1}\}) + C$, $F'_{g(i)} \subseteq F_{i+1}$. Thus, $g(i+1) = g(i)$, and, since $g(i) \leq i$, $g(i+1) \leq i+1$.
- If $n_{i+1} \in F'_{g(i)}$, then the master must select n_{i+1} as the $g(i)+1^{st}$ node to be expanded under the nagging search. This is because the master's search procedure is myopic, $F'_{g(i)} \subseteq F_i$ and $n_{i+1} \in F'_{g(i)}$. Thus $F'_{g(i)+1} = F'_{g(i)} - \{n_{i+1}\} + C'$ where C' is also a subset of $c(n_{i+1})$. Myopia guarantees that C and C' are identical, so $F'_{g(i)+1} = F'_{g(i)} - \{n_{i+1}\} + C \subseteq F_i - \{n_{i+1}\} + C = F_{i+1}$. Therefore, $g(i+1) = g(i)+1$, and, since $g(i) \leq i$, $g(i)+1 \leq i+1$.

The function g constructed in this manner is nondecreasing since, for any i, either $g(i+1) = g(i)$ or $g(i+1) = g(i)+1$. If n is the j^{th} node expanded by the master in the presence of nagging, this inductive argument identifies some $i \in \mathcal{N}$ such that $g(i) = j$ and n is the i^{th} node expanded under the serial search. All that remains is to show that, if n is the i^{th} node expanded under the serial search, then n is either the $g(i)^{th}$ node expanded under the nagging search, or n is not expanded under the nagging search. If n is not the $g(i)^{th}$ node expanded by the master under the nagged search, there are two possibilities; either n is expanded before $g(i)$ or it is expanded after $g(i)$. Node n cannot be expanded before $g(i)$ because the inductive construction of g maps some other node expanded before i in the serial to each node expanded before $g(i)$ in the nagged search. If n is expanded after $g(i)$, then, as in the proof of Theorem 2.2, $F'_{g(i)}$ must contain either n or one of its ancestors. The expansion of n in the serial search presupposes the expansion of all ancestors of n. Since n is the i^{th} node expanded, F_{i+1} contains neither n nor any of it's ancestors. Since $F'_{g(i+1)} \subseteq F_{i+1}$, $F'_{g(i+1)}$ also cannot contain n or its ancestors. As $g(i) \leq g(i+1) \leq g(i)+1$, expansion of n cannot come after $g(i)$. □

Theorem 2.3 (Solution Ordering) If the master's search procedure is myopic, the master will find all solutions with nagging that it would without nagging, and it will find them in the same order.

Proof: Let $\langle n_i \rangle$ be the sequence of nodes expanded by a serial search using the master's search procedure and $\langle \hat{n}_i \rangle$ be the subsequence of $\langle n_i \rangle$ containing only the solution nodes. Similarly, let $\langle n'_i \rangle$ be the sequence of nodes expanded by the master under some nagged search and $\langle \hat{n}'_i \rangle$ be the sequence of solutions in $\langle n'_i \rangle$. The theorem is proven if $\langle \hat{n}_i \rangle$ is a subsequence of $\langle \hat{n}'_i \rangle$. First, it is shown that any element of $\langle \hat{n}_i \rangle$ must also be in $\langle \hat{n}'_i \rangle$ and then it is shown that the ordering of nodes in $\langle \hat{n}_i \rangle$ is preserved in $\langle \hat{n}'_i \rangle$.

Assume that n is some element of $\langle \hat{n}_i \rangle$ that does not appear in $\langle n'_i \rangle$. Let j be the maximum integer for which $p^j(n)$ does not appear in $\langle n'_i \rangle$. Node $p^j(n)$ cannot be the root of the search tree, since the root is always placed on the fringe at the beginning of search. Otherwise, myopia dictates that, since $p^j(n)$ is generated when $p^{j+1}(n)$ is expanded in $\langle n_i \rangle$, $p^j(n)$ must also be placed on the fringe when $p^{j+1}(n)$ is expanded in $\langle n'_i \rangle$. The proof focuses on what happens to $p^j(n)$ once it is placed on the fringe in the nagging search. There are three relevant cases:

1. Ordinarily, $p^j(n)$ could be removed from the fringe by expansion. This cannot be the case here since $p^j(n)$ is defined as a node absent from $\langle n'_i \rangle$.
2. Ordinarily, $p^j(n)$ could be removed from the fringe by a prune message. This cannot occur in this case because $T(p^j(n))$ contains the solution n. Theorem 2.1 guarantees that a prune message is transmitted only when its targeted subtree contains no solutions.
3. Ordinarily, $p^j(n)$ might remain on the fringe forever without being selected for expansion. Let F_k be the fringe of the serial search immediately after the expansion of $p^j(n)$. Lemma 2.4 identifies a function $g : \mathcal{N} \rightarrow \mathcal{N}$ such that, if $F'_{g(k)}$ is the fringe reached by the master process after $g(k)$ node expansions, then $F'_{g(k)} \subseteq F_k$. Since F_k follows the expansion of $p^j(n)$, F_k contains neither $p^j(n)$ nor any of its ancestors. The fringe $F'_{g(k)}$, being a subset of F_k, must also contain neither $p^j(n)$ nor any of its ancestors. The proof of Theorem 2.2

shows that no fringe following $F'_{g(k)}$ in the nagged search can contain $p^j(n)$, and, therefore, $p^j(n)$ cannot remain on the fringe indefinitely.

This is a contradiction since there are no possible outcomes once $p^j(n)$ is placed on the fringe. Thus, the assumption that n appears in $\langle n_i \rangle$ but not in $\langle n'_i \rangle$ must be false.

A simple inductive argument demonstrates that $\langle \hat{n}_i \rangle$ is a subsequence of $\langle \hat{n}'_i \rangle$. If $\langle \hat{n}_i \rangle$ contains only one element, then $\langle \hat{n}_i \rangle$ must be a subsequence of $\langle \hat{n}'_i \rangle$. Otherwise, let n_j and n_k be two nodes in $\langle \hat{n}_i \rangle$, with $j < k$. Let g be the function given by Lemma 2.4. The first part of this proof shows that both n_j and n_k must be expanded in the nagged search. Since n_j and n_k are also expanded under the serial search, Lemma 2.4 requires that they appear at indices $g(j)$ and $g(k)$ respectively in $\langle \hat{n}'_i \rangle$. Since $j < k$ and g is nondecreasing, n_j must precede n_k in $\langle n'_i \rangle$. Thus, $\langle \hat{n}'_i \rangle$ must be a subsequence of $\langle \hat{n}_i \rangle$. $\square$

Theorem 2.5 (Non-Increasing Search) As long as the master's search procedure is myopic, then, for any solution n, if n is the i^{th} node expanded without nagging then n will be found within j node expansions with nagging where $j \leq i$.

Proof: As shown in Theorem 2.3, node n must be found in the nagged search. For g defined as in Lemma 2.4, if n is the i^{th} node expanded in the serial search, then it will be the $g(i)^{th}$ node expanded in the nagged search. Lemma 2.4 insures that $g(i) \leq i$. $\square$

Theorem 2.6 (Completeness for Finite T) Given a finite search tree T and a punctilious search procedure that discovers all solutions in T, a nagged search based on the same search procedure will also discover all solutions in T.

Proof: Assume that n is a solution found in the serial search but not under nagging. Let $j > 0$ be the minimum integer such that $p^j(n)$ is expanded under both the serial and nagged searches. The underlying punctilious search procedure must place $p^{j-1}(n)$ on the fringe with the expansion of $p^j(n)$ since the solution n is in $T\,(p^{j-1}(n))$. Once placed on the fringe under a nagged search, there are ordinarily three possible fates for a node. In the case of $p^{j-1}(n)$, none of these are possible.

1. If $p^{j-1}(n)$ is removed from the fringe by expansion, then j is not the minimal integer for which $p^j(n)$ is expanded under both serial and nagged searches.
2. Node $p^{j-1}(n)$ could not be removed from the fringe by a nagging prune message since solution n is in $T\,(p^{j-1}(n))$.

3. Node $p^{j-1}(n)$ may not remain on the fringe without eventually being selected for expansion. Since T is finite, the expansion of n cannot be deferred indefinitely.

Therefore, the assumption that n is discovered only under the serial search must be false. □

Theorem 2.7 (Fault Tolerance) Theorems 2.3 and 2.6 apply even if messages under the nagging protocol are lost or delayed.

Proof: The completeness theorems for nagging depend only on the master's receipt of prune messages. This proof relates the loss or delay of arbitrary nagging messages to the eventual loss or delay of prune messages.

First, consider the consequences of the loss or delay of an idle message. Since the master has no agenda for the receipt of idle messages, there will be no direct effect on the master's search. The absence of such a message will, however, suppress the reciprocal problem message.

After transmitting an idle message, a nagging process waits for the assignment of a new problem. The loss or delay of a problem message will simply cause the nagger to continue waiting. Since the basic nagging protocol makes no provision for reminding the master that a nagger is idle, lost idle messages may cause idle naggers to remain idle indefinitely. Here also, there is direct effect on the master's search, but there is the possibility that subsequent prune messages will be postponed or eliminated.

Thus, with respect to the master's search, the only tangible consequences of lost or delayed messages are manifested in the prune message. As with the idle message, the master does not anticipate the arrival of prune messages; it simply continues exploring the search tree and processes them as they are received. The proofs of Theorems 2.3 and 2.6 make no assumptions about the number or schedule of prune messages received by the master. Even if failures in communication result in the loss of prune messages, the completeness proofs in these theorems are still valid. If a prune message is simply delayed until the master has completed exploration of the related subtree, the message will cause no changes to the fringe. This is because once some $T(n)$ has been completely explored, all of its nodes have been expanded and, therefore, are no longer on the fringe. □

Lemma 3.2 (Operator Transposition) Let Δ be some tableau containing two different branches β_1 and β_2. Let op_1 and op_2 be two model elimination inferences that operate on β_1 and β_2 respectively such that $op_2 op_1 \Delta$ is defined. The tableau $op_1 op_2 \Delta$ is defined and is a syntactic variant of $op_2 op_1 \Delta$.

Proof: Let $\Delta = \langle\tau, \mu\rangle$. Let θ_1 be the substitution applied when op_1 is applied to Δ and let θ_2 be the substitutions applied when op_2 is applied to $op_1 \langle\tau, \mu\rangle$. The substitutions applied by op_1 and op_2 may be different when these operations are transposed, but they must still satisfy the definitions of extension and reduction.

- If op_2 is of the form $Red_{\beta_2,n}$ then $op_2 \langle\tau, \mu\rangle$ is defined. Branch β_2 is present in $\langle\tau, \mu\rangle$, and, since the application of op_2 to $op_1 \langle\tau, \mu\rangle$ requires that $\mu(n)\ \theta_1\ \theta_2$ and $\mu(\mathit{leaf}(\beta_2))\ \theta_1\ \theta_2$ be complementary, $\neg\mu(n)$ and $\mu(\mathit{leaf}(\beta_2))$ have $\theta_1 \circ \theta_2$ as a unifier and must therefore have a most general unifier. For the remainder of this proof, let θ'_2 be the substitution imposed on the tableau when op_2 is applied to $\langle\tau, \mu\rangle$, and let l_1 and l_2 stand for $\mu(\mathit{leaf}(\beta_2))$ and $\mu(n)$ respectively.
- If op_2 is some $Ext_{\beta_2,C,l}$ then $op_2 \langle\tau, \mu\rangle$ is defined. As required by the statement of the lemma, β_2 is a branch of $\langle\tau, \mu\rangle$. Substitution $\theta_1 \circ \theta_2$ is a witness to the unifiability of $\mu(\mathit{leaf}(\beta_2))$ and $\neg l$. For the remainder of this proof, let θ'_2 be the substitution imposed on the tableau when op_2 is applied to $\langle\tau, \mu\rangle$, and let l_1 and l_2 stand for $\mu(\mathit{leaf}(\beta_2))$ and l respectively.

The operation of op_1 on $op_2 \langle\tau, \mu\rangle$ depends on op_1 and the state of the tableau after op_2 is applied:

- If op_1 is of the form $Red_{\beta_1,n}$, then $op_1 op_2 \langle\tau, \mu\rangle$ is defined. Since op_1 and op_2 operate on different branches, the only change op_2 can effect on β_1 is a change in labeling. Substitution $\theta_1 \circ \theta_2$ unifies l_1 and l_2 as evidenced by the applicability of op_2 to $op_1 \langle\tau, \mu\rangle$. There exists a substitution θ' such, for all $\hat{n} \in \tau$, $\mu(\hat{n})\ \theta'_2\ \theta' = \mu(\hat{n})\ \theta_1\ \theta_2$. This θ' exists whether op_2 is a reduction or an expansion:

  1. If op_2 is a reduction, then the substitution θ'_2 is the most general unifier of l_1 and l_2. Since $\theta_1 \circ \theta_2$ is also a unifier of these literals, the definition of a most general unifier specifies that there exists θ' such that $\theta'_2 \circ \theta' = \theta_1 \circ \theta_2$.
  2. If op_2 is an extension, then the substitution θ'_2 is the composition of a most general unifier of l_1 and l_2 and a second substitution that renames the variables of the resulting tableau. Let θ_2^u represent the most general unifier applied in θ'_2 and let θ_2^s be the substitution that then standardizes the tableau apart from S. Since $\theta_1 \circ \theta_2$ is also a unifier of l_1 and l_2, while θ_2^u is their most general unifier, there exists θ'' such that $\theta_2^u \circ \theta'' = \theta_1 \circ \theta_2$. The labeling of the tableau after applying substitution $\theta_2^u \circ \theta_2^s$ is a syntactic variant of its labeling after applying only θ_2^u. Consequently, there must

exist a substitution θ_2^r such that the labeling of the tableau after applying $\theta_2^u \circ \theta_2^s \circ \theta_2^r$ is identical to its labeling after applying only θ_2^u. The substitution θ' is therefore defined as $\theta_2^r \circ \theta''$.

Since $\theta_1 \circ \theta_2$ is a unifier of $\mu(leaf(\beta_1))$ and $\neg\mu(n)$, θ' is a unifier of $\mu(leaf(\beta_1))\ \theta_2'$ and $\neg\mu(n)\ \theta_2'$. The substitution applied when op_1 operates on $op_2\langle\tau,\mu\rangle$ is the analogous most general unifier. As a result, $op_2 op_1\langle\tau,\mu\rangle$ must be an instance of $op_1 op_2\langle\tau,\mu\rangle$ when op_1 is a reduction.

- If op_1 is some $Ext_{\beta_1,C,l}$, then $op_1 op_2\langle\tau,\mu\rangle$ is defined. As in the former case, there exists a θ' such that, for all $\hat{n} \in \tau$, $\mu(\hat{n})\ \theta_2'\ \theta' = \mu(\hat{n})\ \theta_1\ \theta_2$. Since $\theta_1 \circ \theta_2$ is a unifier of $\neg l$ and $\mu(leaf(\beta_1))$, and the literal l does not change with the application of op_2, this θ' is a unifier of $\neg l$ and $\mu(leaf(\beta_1))\ \theta_2'$. The substitution applied when op_1 operates on $op_2\langle\tau,\mu\rangle$ consists of the corresponding most general unifier followed by a standardizing apart of the tableau from S. Since standardizing apart results in only syntactic variation in the tableau, $op_2 op_1\langle\tau,\mu\rangle$ must again be an instance of $op_1 op_2\langle\tau,\mu\rangle$ when op_1 is an extension.

This construction ensures that $op_2 op_1\langle\tau,\mu\rangle$ is an instance of $op_1 op_2\langle\tau,\mu\rangle$. A similar argument constructing $op_2 op_1\langle\tau,\mu\rangle$ from $op_1 op_2\langle\tau,\mu\rangle$ would demonstrate that $op_1 op_2\langle\tau,\mu\rangle$ is an instance of $op_2 op_1\langle\tau,\mu\rangle$. Thus, provided they operate on branches that exist contemporaneously, it is possible to exchange adjacent inference operations without changing the resulting tableau apart from syntactic variation. □

Theorem 3.1 $\mathcal{P} \subseteq \mathcal{F}$

Proof of Theorem 3.1: Let f be some function in $\mathcal{P}$, and let T be some model elimination search tree. Assume that $T(\Delta)$ contains a closed tableau Δ_s of height k, for some $\Delta \in T$. It must be shown that $f(\Delta, T)$ also contains a closed tableau of height no more than k. Let $\left\langle op_1, \ldots op_j \right\rangle$ be a bloated sequence of model elimination operations such that $op_j \ldots op_1\Delta = \Delta_s$, and let Δ' be the permuted, truncated tableau at the root of $f(\Delta, T)$. It is possible to reorder operations in $\left\langle op_1, \ldots op_j \right\rangle$ into a new, bloated sequence $\left\langle op_{1,m}, \ldots op_{j,m} \right\rangle$ such that $op_{j',m} \ldots op_{1,m}\Delta'$ is closed for some $j' \leq j$.

Closing of Δ' may differ from the closing of Δ in two respects. Primarily, there may be branches in Δ that do not appear in Δ'. Additionally, the nagger's search may select open branches from Δ' in a different order from the master's selection of branches in Δ. However, the definitions of the model elimination inference operations make reference only to the labeling of a particular branch. The deletion of leaves from a set of open branches does not directly affect the state of branches outside this set or the inference operations that can be applied to them.

The sequence $\left\langle op_{1,m}, \ldots op_{j,m} \right\rangle$ is constructed inductively from $\left\langle op_1, \ldots op_j \right\rangle$ by repeated local transpositions permitted under Lemma 3.2. Let the initial sequence $\left\langle op_{1,0}, \ldots op_{j,0} \right\rangle = \left\langle op_1, \ldots op_j \right\rangle$. Assume that the sequence $\left\langle op_{1,i}, \ldots op_{j,i} \right\rangle$ satisfies the following for some $i \geq 0$:

- $\left\langle op_{1,i}, \ldots op_{j,i} \right\rangle$ is bloated for Δ
- $op_{j,i} \ldots op_{1,i}\Delta$ is closed
- $op_{i,i} \ldots op_{1,i}\Delta'$ is defined
- if β is a branch of $op_{i,i} \ldots op_{1,i}\Delta'$ then β is also a branch of $op_{i,i} \ldots op_{1,i}\Delta$ and it has the same labeling in both tableaux

Clearly these conditions are satisfied for $i = 0$. If $op_{i,i} \ldots op_{1,i}\Delta'$ is closed, then $j' = m = i$. Otherwise, the next sequence $\left\langle op_{1,i+1}, \ldots op_{j,i+1} \right\rangle$ is defined as follows. Let β be the branch chosen by the nagger's branch selection function on the tableau $op_{i,i} \ldots op_{1,i}\Delta'$. Let o be the index of the single inference in $\left\langle op_{1,i}, \ldots op_{j,i} \right\rangle$ pertaining to branch β. Since $\left\langle op_{1,i}, \ldots op_{j,i} \right\rangle$ is bloated, this o is unique. Define $\left\langle op_{1,i+1}, \ldots op_{j,i+1} \right\rangle$ as:

$$\left\langle op_{1,i+1}, \ldots op_{j,i+1} \right\rangle = \left\langle op_{1,i}, \ldots op_{i,i}, op_{o,i}, op_{i+1,i}, \ldots op_{o-1,i}, op_{o+1,i}, \ldots op_{j,i} \right\rangle$$

This promotion of inference operation $op_{o,i}$ is permissible since it corresponds to repeated transpositions of $op_{o,i}$ with its predecessor. Since β is an open branch in $op_{i,i} \ldots op_{1,i}\Delta'$, the inductive assumption requires that it must also be an open branch in $op_{i,i} \ldots op_{1,i}\Delta$ and must remain an open branch until $op_{o,i}$ is applied. Thus, all of these transpositions meet the preconditions for Lemma 3.2, and $op_{j,i+1} \ldots op_{j,i+1}\Delta$, like $op_{j,i} \ldots op_{j,i}\Delta$ is closed. Similarly, since $\left\langle op_{1,i+1}, \ldots op_{j,i+1} \right\rangle$ contains the same inference operations as $\left\langle op_{1,i}, \ldots op_{j,i} \right\rangle$, it is also bloated for Δ. Finally, $op_{i+1,i+1} \ldots op_{1,i+1}\Delta'$ is defined and features a subset of the branches in $op_{i+1,i+1} \ldots op_{1,i+1}\Delta$. Since β occurs in both $op_{i,i+1} \ldots op_{1,i+1}\Delta'$ and $op_{i,i+1} \ldots op_{1,i+1}\Delta$ with the same labeling, applying $op_{i+1,i+1}$ to either tableau imposes the same substitution and has the same effect on branches.

This construction must terminate after a finite number of iterations. The branches in the transformed tableau are a always a subset of the branches in the original tableau and branches occurring in both are labeled identically. Since all branches in $op_{j,i} \ldots op_{1,i}\Delta$ are closed for every i, m is bounded above by j.

It is easy to show that tableau $op_{j',m} \ldots op_{1,m}\Delta'$ must be of height no greater than k. Since $op_{j,m} \ldots op_{1,m}\Delta$ differs from $op_j \ldots op_1\Delta$ only in syntactic variation, the heights of these tableaux must be the same. Neither extension nor reduction are capable of reducing the height of the tableau. Thus $op_{j',m} \ldots op_{1,m}\Delta$ must be of height k or less. The inductive hypothesis guarantees that all branches of $op_{j',m} \ldots op_{1,m}\Delta'$ appear in $op_{j',m} \ldots op_{1,m}\Delta$, so $op_{j',m} \ldots op_{1,m}\Delta'$ must also be of height no greater than k. $\square$

Theorem 3.3 $\mathcal{A} \subseteq \mathcal{F}$

Proof: Let T be a model elimination search tree with corresponding theory S. Assume that $T(\Delta)$ contains a solution of height m for some tableau $\Delta \in T$. Let $\left\langle op_1, \ldots op_j \right\rangle$ be a bloated sequence of inference operations such that $op_j \ldots op_1\Delta$ is closed. Let g_{abs} be an abstraction mapping for the symbols in S and let f be a transformation based on g_{abs}. Let S' be the abstracted theory associated with $f(\Delta, T)$, and let Δ' be the abstracted tableau at the root of $f(\Delta, T)$. The sequence $\left\langle op_1, \ldots op_j \right\rangle$ can be used to construct a sequence of inference operations $\left\langle op'_1, \ldots op'_j \right\rangle$ in S' such that $op'_j \ldots op'_1\Delta'$ is closed and of height m or less.

As in Definition 3.19, let $=_{abs}$ be the equivalence relation on which g_{abs} is based. Define the trivial abstraction mapping $\bar{g}_{abs}$ such that, for any formula F, $\bar{g}_{abs}(F)$ is the member of $g_{abs}(F)$ for which:

- Each occurrence of constant c in F is replaced by $f_{[c]}(c, V_1, \ldots V_n)$ where n is the maximum arity of symbols in $[c]$, $V_1, \ldots V_n$ are new, unique variables and $f_{[c]}$ is a function symbol specific to the equivalence class of c that does not appear in S.
- Each occurrence of function symbol h in F of the form $h(t_1, \ldots t_k)$ is replaced by $f_{[h]}(h, t_1, \ldots t_k, V_{k+1}, \ldots V_n)$ where n is the maximum arity of symbols in $[h]$, $V_{k+1}, \ldots V_n$ are new variables and $f_{[h]}$ is a function symbol specific to the equivalence class of h that does not appear in S.

The proof depends on showing that, for any $1 \leq i \leq j$, $\bar{g}_{abs}(op_i \ldots op_1\Delta)$ and $op'_i \ldots op'_1\Delta'$ have a common instance with no more open branches than $op_i \ldots op_1\Delta$. The base case is obvious. Since $\bar{g}_{abs}(\Delta)$ differs from Δ' only in that $\bar{g}_{abs}(\Delta)$ will contain constant or function symbols where Δ' may contain unique variables, it is possible to construct a substitution replacing these variables in Δ' with their representatives in $\bar{g}_{abs}(\Delta)$. Therefore, $\bar{g}_{abs}(\Delta)$ is an instance of Δ'.

For notational convenience, let Δ_i be shorthand for $op_i \ldots op_1\Delta$. Similarly, let Δ'_i stand for $op'_i \ldots op'_1\Delta'$. Let $\bar{\Delta}_i$ be the, inductively constructed, common instance

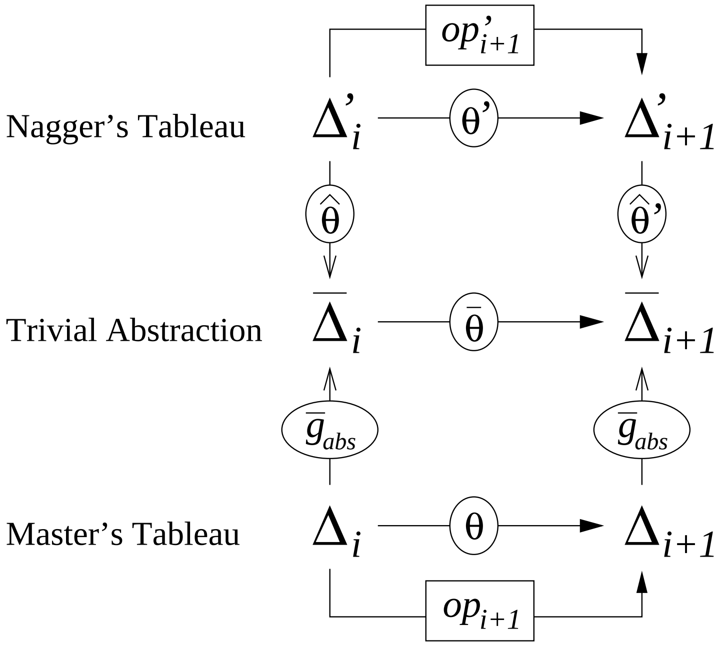


Figure A.1: Master's, nagger's and trivially abstracted tableaux.

of $\bar{g}_{abs}(\Delta_i)$ and Δ'_i. In particular, $\bar{\Delta}_i$ can be constructed from $\bar{g}_{abs}(\Delta_i)$. Branches that are closed in Δ_i may not be closed in $\bar{g}_{abs}(\Delta_i)$. If the equivalence relation $=_{abs}$ includes symbols of unlike arity any a single equivalence class, $\bar{g}_{abs}$ may introduce new, unique variables when applied to Δ_i. Since each such variable is unique, literals that are complementary in Δ_i may be replaced by abstracted literals that are not complementary but are complementary unifiable. Again, since each occurrence of new variables introduced by $\bar{g}_{abs}$ is unique, there exists a single substitution θ such that any closed branch in Δ_i is also closed in $\bar{g}_{abs}(\Delta_i)\ \theta$. With $\bar{\Delta}_i$ defined as $\bar{g}_{abs}(\Delta_i)\ \theta$, $\bar{\Delta}_i$ is automatically an instance of $\bar{g}_{abs}(\Delta_i)$. Proof depends on showing that $\bar{\Delta}_i$ is an instance of Δ'_i.

Assume that $\bar{\Delta}_i$ is an instance of Δ'_i for some $i \in \{0, \ldots j-1\}$. Figure A.1 outlines the structure of this inductive argument. It must be shown that there is a substitution $\hat{\theta}'$ such that $\Delta'_{i+1}\ \hat{\theta}' = \bar{\Delta}_{i+1}$. Let $\hat{\theta}$ be the substitution that makes Δ'_i identical to $\bar{\Delta}_i$. Consider operation op_{i+1} in $\left\langle op_1, \ldots op_j \right\rangle$. Assume that op_{i+1} operates on branch β and let $\theta = \{t_1/v_1, \ldots t_j/v_j\}$ denote the substitution applied to the tableau by op_{i+1}. Let $\bar{\theta} = \{\bar{g}_{abs}(t_1)/v_1, \ldots \bar{g}_{abs}(t_j)/v_j\}$. The substitution $\bar{\theta}$ captures the changes in tableau labels from $\bar{\Delta}_i$ to $\bar{\Delta}_{i+1}$. Consider some label l_i on a node in Δ_i. The trivial abstraction simply replaces non-variable symbols with ground terms and possibly introduces a number of new variables. Since substitution only affects variable symbols and the variables introduced by cannot occur anywhere in l_i, $\bar{g}_{abs}(l_i\ \theta)$ and $\bar{g}_{abs}(l_i)\ \bar{\theta}$ must be syntactic variants.

The form of op'_{i+1} depends on the form of op_{i+1}. If op_{i+1} is some reduction $Red_{\beta,n}$, then $op'_{i+1} = Red_{\beta,n}$. The substitution θ makes the labels of $leaf(\beta)$ and n complementary in Δ_{i+1}. Since the trivial abstraction replaces constant symbols uniformly, the labels on $leaf(\beta)$ and n in $\bar{\Delta}_{i+1}$ are also complementary. The labels on $leaf(\beta)$ and n in $\bar{\Delta}_i$ are the same as the labels of $leaf(\beta)$ and n in $\Delta'_i\ \hat{\theta}$. Thus, $\hat{\theta} \circ \bar{\theta}$ is a witness to the unifiability of $leaf(\beta)$ and n in Δ'_i, and $Red_{\beta,n}\Delta'_i$ is defined. Let θ' be the substitution imposed when $Red_{\beta,n}$ is applied to Δ'_i. Since θ' is a most general unifier, there must be some $\hat{\theta}'$ such that $\theta' \circ \hat{\theta}' = \hat{\theta} \circ \bar{\theta}$. Thus, $(Red_{\beta,n}\Delta'_i)\ \hat{\theta}' = \bar{g}_{abs}(Red_{\beta,n}\Delta_i)$ and therefore $\bar{\Delta}_{i+1}$ is an instance of Δ'_{i+1}.

If op_{i+1} is of the form $Ext_{\beta,C,l}$ then it not only applies some substitution θ to the tableau labeling but also adds one or more children to $leaf(\beta)$. In this case, op'_{i+1} is also an extension. Relating Δ_{i+1} to $\bar{\Delta}_{i+1}$ is a bit more complicated in the extension case because $\hat{\theta}$ relates the tableaux Δ_i to $\bar{\Delta}_i$ but not their respective theories. It is necessary to extend the substitution $\hat{\theta}$ to account for instantiation of both the tableau and the relevant clause C. Let C' be the member of $g_{abs}(C)$ chosen by the transformation f and let l' be the member of C' that corresponds to l. With g_{tr} defined as the abstraction trail from S to S', let $\langle C'', l''\rangle = g_{tr}(C, l)$. If $C'' \neq C'$, then Definition 3.21 requires that there must be a substitution θ_C such that $C''\ \theta_C = C'$. If $C'' = C'$, then simply let θ_C be the empty set (the identity substitution). Given these definitions, $op'_{g(i)+1} = Ext_{\beta,C'',l''}$. The abstractions represented by C' and $\bar{g}_{abs}(C)$ differ only in that $\bar{g}_{abs}(C)$ will contain constant or function symbols in places where C' may contain unique variables. As with $\bar{g}_{abs}(\Delta)$ and Δ' it is possible to construct a substitution replacing these variables in C' with their representatives in $\bar{g}_{abs}(C)$. Call this substitution $\hat{\theta}_C$. Composing θ_C and $\hat{\theta}_C$, $C''\ \theta_C\ \hat{\theta}_C = \bar{g}_{abs}(C)$. Because S' and Δ'_i are variable disjoint as are $\bar{g}_{abs}(S)$ and $\bar{\Delta}_i$, $\hat{\theta} \circ (\theta_C \circ \hat{\theta}_C) = \hat{\theta} \cup (\theta_C \circ \hat{\theta}_C)$. Thus $\Delta'_i\ \hat{\theta}\ \theta_C\ \hat{\theta}_C = \bar{\Delta}_i$ and $C''\ \hat{\theta}\ \theta_C\ \hat{\theta}_C = \bar{g}_{abs}(C)$. Again, since the trivial abstraction replaces symbols uniformly with ground terms, the label of $leaf(\beta)$ in $\bar{\Delta}_i$ and $\bar{g}_{abs}(C)$ must be complementary unifiable literals with $\bar{\theta}$ as a unifier. Consequently, $\hat{\theta} \circ \theta_C \circ \hat{\theta}_C \circ \bar{\theta}$ is a unifier of l'' and $leaf(\beta)$ in Δ'_i and $Ext_{\beta,C'',l''}\Delta'_{g(i)}$ is defined. Let θ' be the substitution imposed when $Ext_{\beta,C'',l''}$ is applied to Δ'_i and let θ_{mgu} be the most general unifier of l'' and $leaf(\beta)$ in Δ'_i. Since θ_{mgu} is a most general unifier, there must be some $\hat{\theta}'_a$ such that $\theta_{mgu} \circ \hat{\theta}'_a = \hat{\theta} \circ \theta_C \circ \hat{\theta}_C \circ \bar{\theta}$. Of course, θ' is just the composition of this most general unifier θ_{mgu} and second substitution that standardizes the tableau apart from S'. Consequently, there must be a substitution $\hat{\theta}'_b$ such that $\theta' \circ \hat{\theta}'_b = \theta_{mgu}$. Let $\hat{\theta}'$ be defined as $\hat{\theta}'_b \circ \hat{\theta}'_a$. As before, $\hat{\theta}'$ demonstrates that $\bar{\Delta}_{i+1}$ is an instance of Δ'_{i+1}. For node n in Δ_i, $\bar{g}_{abs}(\mu(n))\ \bar{\theta} = \mu'(n)\ \theta'\ \hat{\theta}'$. For tableau nodes added in the $i+1^{st}$ inference, any literals $l_C \in C$ and $l_{C''} \in C''$, satisfy

$l_C\ \bar{\theta} = l_{C''}\ \theta'\ \hat{\theta}'$.

It is easy to show that $op'_j \ldots op'_1\Delta'$ is closed and of height k. The definition of $\mathcal{A}$ stipulates that tableau Δ' must not contain open branches that are not open in Δ. Each extension in $\left\langle op'_1, \ldots op'_j \right\rangle$ introduces the same number of new branches into the transformed tableau as its corresponding extension in $\left\langle op_1, \ldots op_j \right\rangle$ introduces into the original tableau. Since partial proof $\left\langle op_1, \ldots op_j \right\rangle$ is bloated, it contains a reduction for every branch that it closes. Since all tableau branches are closed in $op_j \ldots op_1\Delta$ and $\bar{\Delta}_j$ all branches must also be closed in $op'_j \ldots op'_1\Delta'$. □

Theorem 4.1 Let Δ be an open tableau with at least two open branches. Let B be the set of open branches in Δ and let $\hat{B}$ be a nonempty, proper subset of B. If $\langle op_1, \ldots op_k\rangle$ is a weakly local subproof for $\hat{B}$ in Δ and op' is an inference operation for a branch in $B - \hat{B}$, then the following are true:

1. if op' is not applicable to Δ then op' is also not applicable to $op_k \ldots op_1\Delta$.
2. if op' is applicable to Δ then op' is also applicable to $op_k \ldots op_1\Delta$.
3. if op' is applicable to Δ then $\langle op_1, \ldots op_k\rangle$ is still a weakly local subproof for $\hat{B}$ in $op'\Delta$.

Proof: To verify the first property, assume that op' is applicable to $op_k \ldots op_1\Delta$ but not to Δ. Since the branch pertaining to op' is open in Δ, repeated application of the Operation Transposition Lemma of Chapter 3 demonstrates that $op' op_k \ldots op_1\Delta$ and $op_k \ldots op_1 op'\Delta$ are syntactic variants. This implies that, contrary to the assumption, $op'\Delta$ is defined.

To verify the second property, assume that op' is applicable to Δ. It must be shown that op' is also applicable to $op_k \ldots op_1\Delta$. Operation op' relates to some open branch $\beta \in (B - \hat{B})$. Let θ be the substitution imposed on the tableau when op' is applied to Δ, let θ_a be the composition of all substitutions imposed on the tableau when $\langle op_1, \ldots op_k\rangle$ is applied to Δ and let μ be the usual labeling function of Δ. Whether an extension or a reduction, the applicability of op' depends only on the condition of β. The status of $\langle op_1, \ldots op_k\rangle$ as a local subproof implies that, when applied to Δ, the only changes it can effect on β represent syntactic variation. Consequently, there exists some substitution θ_r such that, for all $n \in \beta$, $\mu(n)\ \theta_a\ \theta_r = \mu(n)$. Two cases must be addressed to show that op' is applicable to $op_k \ldots op_1\Delta$:

- If op' is of the form $Red_{\beta,n}$ then $\theta_r \circ \theta$ is a witness to the unifiability of $\mu(n)\ \theta_a$ and $\neg\mu(\mathit{leaf}(\beta))\ \theta_a$. Thus op' is applicable to $op_k \ldots op_1\Delta$.

- If op' is of the form $Ext_{\beta,C,l}$ then $\mu(leaf(\beta))$ must be complementary and unifiable with some literal l from the theory. Since the theory does not change when $\langle op_1, \ldots op_k\rangle$ is applied to Δ and $op_k, \ldots op_1, \Delta$ is variable-disjoint from S, $\theta_r \circ \theta$ is, again, a witness to the unifiability of l and $\neg\mu(leaf(\beta))\,\theta_a$. Thus op' is applicable to $op_k \ldots op_1\Delta$.

The final property requires that all three conditions of weak locality be satisfied. These are demonstrated in the order they are given in the weak locality definition. Assume that op' is applicable to Δ.

First, it must be shown that, when applied to $op'\Delta$, $\langle op_1, \ldots op_k\rangle$ still closes all branches in $\hat{B}$. Proof of the second property demonstrated that $op'op_k \ldots op_1\Delta$ is defined. Repeated application of the Operation Transposition Lemma of Chapter 3 demonstrates that $op'op_k \ldots op_1\Delta$ and $op_k \ldots op_1 op'\Delta$ are syntactic variants. Assume that there is some $\beta \in \hat{B}$ such that $T(leaf(\beta))$ is closed in $op_k, \ldots op_1\Delta$ but not in $op_k \ldots op_1 op'\Delta$. Since $T(leaf(\beta))$ is closed in $op_k \ldots op_1\Delta$, it must also be closed in $op'op_k, \ldots op_1\Delta$. When applied to $op_k \ldots op_1\Delta$, the only changes op' can effect on $T(leaf(\beta))$ consist of substitution-induced changes to node labeling. In $op_k, \ldots op_1\Delta$, each branch containing $leaf(\beta)$ exhibits a pair of complementary literals; these literals, featuring identical atoms, will still be complementary after op' has applied its substitution. Since $op'op_k \ldots op_1\Delta$ and $op_k \ldots op_1 op'\Delta$ are syntactic variants, $op_k \ldots op_1 op'\Delta$ is an instance of $op'op_k \ldots op_1\Delta$. Let θ' be the substitution such that $op_k \ldots op_1 op'\Delta = (op'op_k \ldots op_1\Delta)\,\theta'$. Again, all branches that are closed in $op'op_k \ldots op_1\Delta$ must also be closed in $(op'op_k \ldots op_1\Delta)\,\theta' = op_k \ldots op_1 op'\Delta$. Thus, contrary to the assumption, $T(leaf(\beta))$ must be closed in $op'op_k \ldots op_1\Delta$.

Let B' be the set of open branches in the tableau $op'\Delta$, and let β be some branch in $B' - \hat{B}$. Assume that $\langle op_1, \ldots op_k\rangle$ adds a child to β when applied to $op'\Delta$. Let i be the smallest integer such that a child is added to $leaf(\beta)$ when op_i is applied to $op_{i-1} \ldots op_1 op'\Delta$. Operation op_i must be an extension of β. Let β' be the subset of β that is present in Δ. Two cases must be addressed:

$\beta' = \beta$ Because $\langle op_1, \ldots op_k\rangle$ is weakly local and β is open but not in $\hat{B}$, subproof $\langle op_1, \ldots op_k\rangle$ is not permitted to add children to β and thus cannot contain extensions of β.

$\beta' \subset \beta$ Since β is present in $op'\Delta$, operation op' must be the extension of β' creating it. The theorem statement requires that $\beta' \in (B - \hat{B})$. Subproof $\langle op_1, \ldots op_k\rangle$ being weakly local cannot, when applied to Δ, extend β' to create β. As a result, $\langle op_1, \ldots op_k\rangle$ cannot contain extensions of β as $op_j \ldots op_1\Delta$ is defined, but β does not exist in $op_j \ldots op_1\Delta$ for any $j \in \{0, \ldots k\}$.

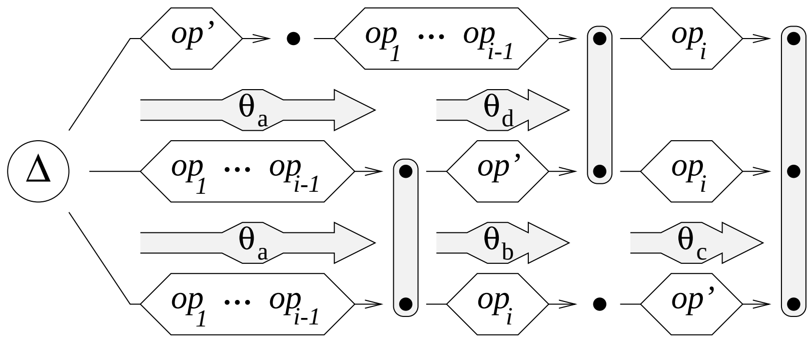


Figure A.2: Relationships among different derivation paths.

Let $N = \left\{n \mid n \in \beta, \beta \in (B' - \hat{B})\right\}$. All that remains is to show that the labeling of N in $op_k, \ldots op_1 op' \Delta$ is a syntactic variant of its labeling in $op'\Delta$. Assume that, when $\langle op_1, \ldots op_k \rangle$ is applied to $op'\Delta$, the labeling of N is changed in some way not constituting syntactic variation. Let i be the smallest integer such that the labeling of N in $op_i \ldots op_1 op' \Delta$ is not a syntactic variant of its labeling in $op_{i-1} \ldots op_1 op' \Delta$.

Figure A.2 outlines the remainder of this proof. It illustrates the relationships between tableaux derived under different orderings of the inference operations. Repeated application of the Operation Transposition Lemma shows that the tableaux $op_i \ldots op_1 op' \Delta$ and $op_i op' op_{i-1} \ldots op_1 \Delta$ are syntactic variants and that $op' op_i op_{i-1} \ldots op_1 \Delta$ is a syntactic variant of them both. For convenience, names are given to some of the substitutions applied under these derivations. Let θ_a be the substitution that results when $\left\langle op_1, \ldots op_{i-1} \right\rangle$ is applied to Δ. Let θ_b be the substitution imposed by the application of op_i to $op_{i-1} \ldots op_1 \Delta$, and let θ_d be the substitution associated with the application of op' to $op_{i-1} \ldots op_1 \Delta$. Finally, let θ_c be the substitution applied when op' operates on $op_i \ldots op_1 \Delta$.

Let N' be the subset of N that is present in Δ and let μ again denote the labeling function of Δ. The weak locality of $\langle op_1 \ldots op_k \rangle$ ensures that the labeling of N' in $op_{i-1} \ldots op_1 \Delta$ is a syntactic variant of its labeling in $op_i \ldots op_1 \Delta$. Accordingly, there must be some θ_r and θ_s such that, for all $n \in N'$, $\mu(n)\ \theta_a\ \theta_b\ \theta_r = \mu(n)\ \theta_a$ and $\mu(n)\ \theta_a\ \theta_s = \mu(n)\ \theta_a\ \theta_b$. Applying θ_d, $\mu(n)\ \theta_a\ \theta_b\ \theta_r\ \theta_d = \mu(n)\ \theta_a\ \theta_d$ for all $n \in N'$. Since op' operates on the same branch in the same way no matter when it is applied, $\theta_r \circ \theta_d$ unifies the same pair of literals in $op_i \ldots op_1 \Delta$ as does θ_c. The substitution θ_c is either a most general unifier of these literals or a most general unifier composed with a substitution that standardizes the tableau apart from S. In either case, there must be some θ' such that $\theta_c \circ \theta' = \theta_r \circ \theta_d$. Thus, the labeling of N' in $op' op_{i-1} \ldots op_1 \Delta$ is an instance of its labeling in $op' op_i \ldots op_1 \Delta$. Similarly, $\mu(n)\ \theta_a\ \theta_s\ \theta_c = \mu(n)\ \theta_a\ \theta_b\ \theta_c$. Again, because of the way θ_d is chosen, there must be some θ'' such that $\theta_d \circ \theta'' = \theta_s \circ \theta_c$. The labeling of N' in $op' op_i \ldots op_1 \Delta$ must also be an instance of its labeling in $op' op_{i-1} \ldots op_1 \Delta$.

This result can be easily extended from N' to the entire set N. If $N' \neq N$, then op' must be an extension. In this case, op' adds tableau nodes with labels taken from some clause $C \in S$. Since clause C is unchanged no matter when op' is applied and the tableau is always variable-disjoint from C, $l\ \theta_r\ \theta_d = l\ \theta_d$ for all $l \in C$. As before, this demonstrates that N in $op' op_{i-1} \ldots op_1 \Delta$ is an instance of its labeling in $op' op_i \ldots op_1 \Delta$. Likewise, $l\ \theta_s\ \theta_c = l\ \theta_c$ and N in $op' op_i \ldots op_1 \Delta$ is an instance of N in $op' op_{i-1} \ldots op_1 \Delta$

Thus, the labeling of N in $op' op_i \ldots op_1 \Delta$ and $op' op_{i-1} \ldots op_1 \Delta$ must be syntactic variants. But, the Operation Transposition Lemma guarantees that $op' op_i \ldots op_1 \Delta$ and $op_i \ldots op_1 op' \Delta$ are syntactic variants, while $op' op_{i-1} \ldots op_1 \Delta$ and $op_{i-1} \ldots op_1 op' \Delta$ are also syntactic variants. Consequently, the labeling of N in $op_i \ldots op_1 op' \Delta$ must be a syntactic variant of its labeling in $op_{i-1} \ldots op_1 op' \Delta$. This contradicts the assumption that i is smallest integer for which the labeling of N in $op_i \ldots op_1 op' \Delta$ was not a syntactic variant of its labeling in $op_{i-1} \ldots op_1 op' \Delta$. Consequently, none of the operations in $\langle op_1 \ldots op_k \rangle$ change the labeling of N beyond syntactic variation. $\Box$

Theorem 4.2 For problem transformation function $f \in \mathcal{P}$ and tableau $\Delta \in T$, let Δ' be the root of $f(\Delta, T)$, β be an open branch in Δ' and op be an inference operation applicable to β in Δ'. If $T\,(op\ \Delta')$ contains no solutions of height j or less, then $T\,(\Delta)$ contains no solutions of height j or less that include op.

Proof: Assume that $T\,(op\Delta')$ contains no solutions and that there is a bloated sequence of inference operations $\langle op_1, \ldots op_k \rangle$ containing op such that $op_k \ldots op_1 \Delta$ is closed. The mapping from solutions in $T\,(\Delta)$ to solutions in $f(\Delta, T)$ given in the proof of Theorem 3.1 exhibits a new, bloated sequence $\langle op'_1, \ldots op'_{k'} \rangle$ consisting of a reordered subset of $\langle op_1, \ldots op_k \rangle$ such that $op'_{k'} \ldots op'_1 \Delta'$ is closed. Since $\langle op'_1, \ldots op'_{k'} \rangle$ is bloated and contains no operations outside $\langle op_1, \ldots op_k \rangle$, $\langle op'_1, \ldots op'_{k'} \rangle$ must also contain op as it is the only element of $\langle op_1, \ldots op_k \rangle$ that operates on β. Since β is a branch of Δ', permissible local transpositions make it possible to reorder $\langle op'_1, \ldots op'_{k'} \rangle$ so that op is its first operation. This reordered sequence represents a solution in $T\,(op\Delta')$, which is assumed not to exist. Therefore, either $T\,(op\Delta')$ contains a solution $\langle op_1, \ldots op_k \rangle$ does not contain op, or $op_k \ldots op_1 \Delta$ is not closed. $\Box$

Theorem 4.3 Let f be a transformation in $\mathcal{P}$, and let tableau Δ_1 be a node in T and Δ_2 a node in $T\,(\Delta_1)$. If $T\,(op\ \Delta_1)$ contains no solutions of height j or less, then either $f(\Delta_2, T)$ contains solutions of height j or less that do not include op or $T(\Delta_2)$ contains no solutions of height j or less.

Proof: Assume that the theorem is false; assume that $T(op\Delta_1)$ contains no solutions, there is a solution in $f(\Delta_2, T)$ of height j or less, that all such solutions include op and that $T(\Delta_2)$ contains at least one solution of height j or less. Let Δ_2' be the root of $f(\Delta_2, T)$. If β is the branch on which op operates, then the tableau Δ_2' must contain β. Let $\langle op_1, \ldots op_k \rangle$ be a partial proof such that $op_k \ldots op_1 \Delta_2$ is closed and of height at most j. As in the proof of Theorem 4.2, it is possible to construct from $\langle op_1, \ldots op_k \rangle$ a sequence $\langle op_1', \ldots op_{k'}' \rangle$ so that $op_k \ldots op_1 \Delta_2'$ is closed and all elements of $\langle op_1', \ldots op_{k'}' \rangle$ appear in $\langle op_1, \ldots op_k \rangle$. Since it is assumed that all partial proofs closing Δ_2' must contain op, $\langle op_1, \ldots op_k \rangle$, containing a superset of the operations in $\langle op_1', \ldots op_{k'}' \rangle$, must also include some $op_i = op$. Let $\langle op_1^2, \ldots op_m^2 \rangle$ be the partial proof deriving Δ_2 from Δ_1. Thus, $op_k \ldots op_1 op_m^2 \ldots op_1^2 \Delta_1$ is closed. Since $op\Delta_1$ is defined, β must be a branch of Δ_1, and $op_i = op$ can be repeatedly transposed with its predecessor until it is the first operation. Consequently, $op_k \ldots op_{i+1} op_{i-1} \ldots op_1 op_m^2 \ldots op_1^2 op \Delta_1$ is still closed. This violates the requirement that $T(op\ \Delta_1)$ contains no solutions. $\square$

Theorem 4.4 For problem transformation function $f \in \mathcal{A}$ and tableau $\Delta \in T$, let Δ' be the root of $f(\Delta, T)$, β be an open branch in Δ' and op' be an inference operation applicable to β in Δ'. Let S be the theory corresponding to T, S' be the abstracted theory constructed under f and g_{tr} be the abstraction trail from S to S'. If $T(op'\Delta')$ contains no solutions of height j or less, then $T(\Delta)$ contains no solutions of height j or less that include op where op satisfies the following:

- If op' is of the form $Red_{\beta,n}$ then $op = Red_{\beta,n}$.
- If op' is of the form $Ext_{\beta,C',l'}$ then $op = Ext_{\beta,C,l}$ where $g_{tr}(C, l) = \langle C', l' \rangle$.

Proof: Assume that $T(op'\Delta')$ contains no solutions, but there is a sequence of inference operations $\langle op_1, \ldots op_k \rangle$ containing op, and $op_k \ldots op_1 \Delta$ is closed. Since β is an open branch in Δ', β must be a branch in Δ. Through repeated transposition of operators, it is possible to construct from $\langle op_1, \ldots op_k \rangle$ a sequence $\left\langle op_1^T, \ldots op_k^T \right\rangle$ for which $op_1^T = op'$ and $op_k^T, \ldots op_1^T \Delta$ is closed. The mapping from solutions in $T(\Delta)$ to solutions in $f(\Delta, T)$ given in the proof of Theorem 3.3 constructs from $\left\langle op_1^T, \ldots op_k^T \right\rangle$ a new sequence $\langle op_1', \ldots op_k' \rangle$ such that $op_k' \ldots op_1' \Delta_1'$ is closed and of height at most j. If op' and op are reductions, the construction of $\langle op_1', \ldots op_k' \rangle$ makes $op_1' = Red_{\beta,n}$. If op' and op are extensions, the construction makes $op_1' = Ext_{\beta,C',l'}$. In either case, $op_k' \ldots op_2 op' \Delta$ is a closed tableau, contradiction the assumption that $T\,(op'\Delta)$ contains no solutions. Thus, either $T\,(op'\Delta')$ contains no solutions, $\langle op_1, \ldots op_k\rangle$ does not contain op or $op_k \ldots op_1\Delta$ is not closed. $\square$

Theorem 4.5 Let f be a function in $\mathcal{A}$, and let $\Delta_1 \in T$ and $\Delta_2 \in T\,(\Delta_1)$. For theory S, let S' be the abstraction of S on which $f\,(\Delta_2, T)$ is based. Let op' be an inference operation under S' and let O be a set of inference operations under S that satisfy the following:

- If op' is of the form $Red_{\beta,n}$ then $O = \{Red_{\beta,n}\}$.
- If op' is of the form $Ext_{\beta,C',l'}$ and g_{tr} is the abstraction trail from S to S', then $O = \{Ext_{\beta,C,l} \mid g_{tr}(C,l) = \langle C', l'\rangle\}$.

If $T\,(op\ \Delta_1)$ contains no solutions of height j or less for any $op \in O$, then either $f\,(\Delta_2, T)$ contains solutions of height j or less that do not include op' or $T\,(\Delta_2)$ contains no solutions of height j or less.

Proof: Assume that the theorem is false; assume that $T\,(op\Delta_1)$ contains no solutions, but there is a solution in $f(\Delta_2, T)$ of height j or less, all such solutions include op', and $T\,(\Delta_2)$ contains a solution of height j or less. Let Δ_2' be the tableau at the root of $f(\Delta_2, T)$. Since all solutions in $f(\Delta_2, T)$ contain the operator op', Δ_2' must contain β as a branch. Let $\langle op_1, \ldots op_k\rangle$ be a partial proof such that $op_k \ldots op_1\Delta_2$ is closed and of height j or less. Again, the proof of Theorem 3.3 shows how to construct from $\langle op_1, \ldots op_k\rangle$ a sequence $\langle op_1', \ldots op_{k'}'\rangle$ so that $op_k \ldots op_1\Delta_2'$ is closed. Since all solutions in $f(\Delta_2, T)$ must contain op', operation op must occur in $\langle op_1, \ldots op_{k'}\rangle$:

- If op' is of the form $Red_{\beta,n}$, then under the the construction of Theorem 3.3, $\langle op_1, \ldots op_k\rangle$ must contain some $op_i = Red_{\beta,n}$.
- If op' is of the form $Ext_{\beta,C',l'}$, then under the construction of Theorem 3.3, $\langle op_1, \ldots op_k\rangle$ must contain some $op_i = Ext_{\beta,C,l}$ for some clause $C \in S$ and $l \in C$ such that $g_{tr}(C,l) = \langle C', l'\rangle$.

As in the proof of Theorem 4.3, let $\langle op_1^2, \ldots op_m^2\rangle$ be the partial proof deriving Δ_2 from Δ_1. The tableau $op_k \ldots op_1 op_m^2 \ldots op_1^2\Delta_1$ must be closed. Since tableau $op\Delta_1$ is defined, β must be a branch of Δ_1, and op_i can be repeatedly transposed with its predecessor as long β exists. Consequently, $op_k \ldots op_{i+1} op_{i-1} \ldots op_1 op_m^2 \ldots op_1^2 op\Delta_1$ is defined and closed. This violates the requirement that $T\,(op\ \Delta_1)$ contains no solutions. $\square$

Theorem 5.1 For $f \in \mathcal{P}$ or $f \in \mathcal{A}$, the ancestors of Δ marked while exploring $f\,(\Delta, T)$ include an element of $\psi_{\Downarrow}\,(\Delta)$.

Proof: Let m be the smallest integer for which $p^m(\Delta)$ contains solutions. If no such m exists, $\psi_{\Downarrow}(\Delta) = \emptyset$. Let Δ' be some node of $f(\Delta, T)$, and let $p^{m'}(\Delta')$ be the root of $f(\Delta, T)$. Define $\langle \Delta_1, \ldots \Delta_k \rangle$ as $\left\langle p^m(\Delta), \ldots p^0(\Delta), p^{m'}(\Delta'), \ldots p^0(\Delta') \right\rangle$. This sequence records the derivation of Δ' from $p^m(\Delta)$ via inference operations and transformation. As with the marking procedure, it is possible to extend the notion of sufficient marks across transformation. This requires some means of comparing the presence of solutions in transformed search trees reached via different inference paths. A new transformation f' can be formulated to permit this kind of comparison. The function f' is a variant of f that agrees with f on Δ and approximates the behavior of $f(\Delta, T)$ on tableaux that share particular features with Δ. The definition of f' depends on the membership of f in $\mathcal{P}$ or $\mathcal{A}$:

- For $f \in \mathcal{P}$ and any tableau Δ_x derivable from Δ_1, f' is in $\mathcal{P}$. When f' is applied to Δ_x, it discards branch β if and only if β occurs in Δ with the same labeling and β is discarded when f is applied to Δ. Function f' preserves the branch ordering used in $T(\Delta_x)$.
- For $f \in \mathcal{A}$ and any tableau Δ_x derivable from Δ_1, f' is in $\mathcal{A}$ and the abstracted theory employed in $f'(\Delta_x, T)$ is identical to the abstracted theory used in $f(\Delta, T)$. Whenever Δ and Δ_x both contain an identically labeled branch β, the abstracted labeling of β in the root of $f'(\Delta_x, T)$ must be identical to its labeling in the root of $f(\Delta, T)$. Any node not on a branch such as β is labeled with its trivial abstraction as defined in the proof of Theorem 3.3.

If $\Delta_{i+1} \in T$ or if $\Delta_i \in f(\Delta, T)$, let op_i be the inference operation from Δ_i to Δ_{i+1}. For $1 \le j < k$, node Δ_j is classified as *dirty* for Δ_k under the following circumstances:

1. Let $\langle i_1, \ldots i_{j'} \rangle$ be the subsequence of $\langle 1, \ldots j-1 \rangle$ that includes i if and only if Δ_i is not dirty for Δ_k. Node Δ_j is dirty for Δ_k if either of the following are true:
   - $\Delta_{i_{j'}+1}$ is in T, and $T\left(op_{i_{j'}} \ldots op_{i_1} \Delta_1\right)$ contains one or more solutions, but $T\left(op_j op_{i_{j'}} \ldots op_{i_1} \Delta_1\right)$ contains none.
   - If $\Delta_{i_{j'}}$ is in $f(\Delta, T)$, let w be the largest integer for which Δ_{i_w} is in T and let Δ_r be the root of $f'(op_{i_w} \ldots op_{i_1} \Delta_1, T)$. Subtree $T\left(op_{i_{j'}} \ldots op_{i_{w+1}} \Delta_r\right)$ contains solutions, but $T\left(op_j op_{i_{j'}} \ldots op_{i_{w+1}} \Delta_r\right)$ contains none.
2. Node Δ_j is also dirty for Δ_k if Δ_a is dirty for Δ_j and $\Delta_a \in \psi'(\Delta_j, \beta)$ where $a < j$ and β is the selected branch from Δ_j.

Any node that is not identified as dirty for Δ_k is said to be *clean* for Δ_k. Observe that node Δ itself is always clean for Δ_k. Under this definition, the dirty nodes for Δ are the same as the sufficient marks for $T(\Delta)$. The theorem is proven if it can be shown that some dirty node must be marked when the search of $f(\Delta, T)$ is completed. The dirty nodes, being a generalization of the sufficient marks to the transformed search space, permit an induction from the leaves of $f(\Delta, T)$ to its root.

Assume that Δ' is a leaf of $f(\Delta, T)$ with β' as its selected branch and that $\psi'(\Delta', \beta')$ contains no nodes that are dirty for Δ'. Let $\langle i_1, \ldots i_{k'}\rangle$ be the subsequence of $\langle 1, \ldots k-1\rangle$ such that $\{\Delta_{i_a} \mid a = 1, \ldots k'\}$ is the subset of nodes in $\{\Delta_1, \ldots \Delta_{k-1}\}$ that are clean for Δ'. Let w be the largest integer for which $\Delta_{i_w} \in T$ and let Δ_r be the root of $f'(op_{i_w} \ldots op_{i_1}\Delta_1, T)$. The classification of $\Delta_{i_{k'}}$ as clean implies that there is a closed tableau in $T\left(op_{i_{k'}} \ldots op_{i_{w+1}}\Delta_r\right)$. This is shown inductively. In the same induction, it can be demonstrated that, β' must occur in both Δ' and $op_{i_{k'}} \ldots op_{i_{w+1}}\Delta_r$ with the same labeling. By construction, $T(\Delta_1)$ contains solutions. For $j \in \{1 \ldots k-1\}$ let the set B_j contain all target branches of operations $op_{i_{j'}} \ldots op_{i_{k'}}$ for $i_{j'} \geq j$. There are five inductive cases, segregated by the status of Δ_j as clean or dirty and by the transformation function.

1. If $\Delta_{j+1} \in T$ and Δ_j is clean, then let j' be the integer for which $i_{j'} = j$. Assume that all members of B_j that are branches of Δ_j also occur in $op_{i_{j'-1}} \ldots op_{i_1}\Delta_1$ with the same labeling and that $T\left(op_{i_{j'-1}} \ldots op_{i_1}\Delta_1\right)$ contains a solution. Since node Δ_j is not dirty and $i_{j'} = j$, there must also be a solution in $T\left(op_{i_{j'}} \ldots op_{i_1}\Delta_1\right)$. According to the definition of B_j, op_j operates on a member of B_j. As a result, the substitution imposed when op_j is applied to Δ_j is the same as when $op_{i_{j'}} = op_j$ is applied to $op_{i_{j'-1}} \ldots op_{i_1}\Delta_1$. The labelings of nodes in β' and B_j must still be the same in Δ_{j+1} and $op_{i_{j'}} \ldots op_{i_1}\Delta_1$.

2. If $\Delta_{j+1} \in T$ and Δ_j is dirty, then let j' be the greatest integer for which $i_{j'} < j$, and let $\Delta_c = op_{i_{j'}} \ldots op_{i_1}\Delta_1$. If no such j' exists, let $j' = 0$ and $\Delta_c = \Delta_1$. Assume that all members of B_j that are branches of Δ_j occur in Δ_c with the same labeling. Operation op_j cannot create or change the labeling on a branch in B_j. If it did, the second condition in the definition of dirty nodes would require that Δ_j not be dirty since it would be among the tableaux marked on behalf of one of the clean nodes in $\Delta_{i_{j'+1}} \ldots \Delta_{i_{k'}}$. For similar reasons, branch β' cannot be modified by op_j. If it was, then the dirty node Δ_j would be $in\psi'(\Delta', \beta')$. Thus, the labelings of nodes in β' and B_j must still be the same in Δ_{j+1} and Δ_c.

3. If $\Delta_j = \Delta$, the status of f' as a legitimate problem transformation function guarantees that $f'(op_{i_w} \ldots op_{i_1}\Delta_1, T)$ contains a solution whenever

$T\left(op_{i_w} \ldots op_{i_1}\Delta_1\right)$ does. Furthermore, if all branches of B_j that appear in Δ also appear in $op_{i_w} \ldots op_{i_1}\Delta_1$ with the same labeling, the construction of f' insures that they also occur in Δ_r and the root of $f(\Delta, T)$ with identical labelings.

4. If $\Delta_j \in f(\Delta, T)$ and Δ_j is clean, let j' be the integer for which $i_{j'} = j$. Assume that all branches of B_j that are in Δ_j are also present in $op_{i_{j'-1}} \ldots op_{i_{w+1}}\Delta_r$ and have the same labeling in both tableaux. Also assume that $T\left(op_{i_{j'-1}} \ldots op_{i_{w+1}}\Delta_r\right)$ contains solutions. As in Case 1, $T\left(op_{i_{j'}} \ldots op_{i_{w+1}}\Delta_r\right)$ must contain a solution since Δ_j is not dirty. Likewise, operation $op_j = op_{i_{j'}}$ must maintain the identical labelings of nodes in β' and B_j between Δ_{j+1} and $op_{i_{j'}} \ldots op_{i_{w+1}}\Delta_r$.

5. If $\Delta_j \in f(\Delta, T)$ and Δ_j is dirty, then let j' be the greatest integer for which $i_w < i_{j'} < j$ and let $\Delta_c = op_{i_{j'}} \ldots op_{i_{w+1}}\Delta_r$. If no such j' exists, let $j' = i_w$ and $\Delta_c = \Delta_r$. Assume that all members of B_j that are branches of Δ_j occur in Δ_c with the same labeling. As in the second inductive case, operation op_j can neither create nor change the labeling of β' and the branches in B_j.

Inductively, $T\left(op_{i_{k'}} \ldots op_{i_{w+1}}\Delta_r\right)$ must contain a solution, and branch β' must occur in both Δ' and $op_{i_{k'}} \ldots op_{i_{w+1}}\Delta_r$ with the same labeling. But, because Δ' is a leaf node and β' is the selected branch from Δ', there can be no subproof for β' from Δ' within the effective depth bound. Without a subproof for β' the supposed solution in $T\left(op_{i_{k'}} \ldots op_{i_{w+1}}\Delta_r\right)$ cannot exist. Thus, the assumption that no member of $\psi'(\Delta', \beta')$ is dirty must be false.

If Δ' is not a leaf of $f(\Delta, T)$, assume that search of $T(\Delta'')$ marks some dirty node for each $\Delta'' \in c(\Delta')$. If any subtree $T(\Delta'')$ does not mark Δ', then, since it must leave a dirty ancestor marked, it must mark some dirty tableau in $\{\Delta_1, \ldots \Delta_{k-1}\}$. Any ancestor of Δ' that is dirty for some $\Delta'' \in \Delta'$ must also be dirty for Δ'. If $T(\Delta'')$ does not mark a node that is dirty for Δ', it must mark Δ' itself.

If no children of Δ' leave a dirty ancestor of Δ' marked, they must all mark Δ'. So long as $\psi'(\Delta', \beta')$ contains a dirty node, exploring $T(\Delta')$ will leave a dirty node marked. Assume that $\psi'(\Delta', \beta')$ contains no dirty nodes. Since $\psi'(\Delta', \beta')$ includes no dirty nodes, Δ' must qualify as dirty for its children under the first condition of the definition of dirty nodes. Consequently, there must be a subsequence $\langle i_1, \ldots i_{k'}\rangle$ of $\langle 1, \ldots k-1\rangle$ such that, for w and Δ_r defined as above, $T\left(op_{i_{k'}} \ldots op_{i_{w+1}}\Delta_r\right)$ contains solutions. Although $T\left(op_{i_{k'}} \ldots op_{i_{w+1}}\Delta_r\right)$ contains solutions, node $op_{i_{k'}} \ldots op_{i_{w+1}}\Delta_r$ cannot, itself, be a solution. If it was, it would be possible to reorder the derivation of Δ from Δ_1 so that operations in $\left\langle op_{i_1}, \ldots op_{i_{k'}}\right\rangle$ appear first. If $op_{i_{k'}} \ldots op_{i_{w+1}}\Delta_r$

is closed, then permissible operator transpositions can be used to exhibit a closed tableau in $f(\Delta, T)$. If the nagger discovers such a closed tableau, it leads to the marking of all ancestors of Δ.

If $T\left(op_{i_{k'}} \ldots op_{i_{w+1}} \Delta_r\right)$ contains a solution, but $op_{i_{k'}} \ldots op_{i_{w+1}} \Delta_r$ is not, itself, a solution, there must be some op' such that $T\left(op' op_{i_{k'}} \ldots op_{i_{w+1}} \Delta_r\right)$ contains a solution. In fact, the operator transposition lemma implies that op' can be chosen so that it operates on any open branch in $op_{i_{k'}} \ldots op_{i_{w+1}} \Delta_r$. Choose op' so that it operates on β'. As in the case where Δ' is a leaf, β' must have the same labeling in $op_{i_{k'}} \ldots op_{i_{w+1}} \Delta_r$ as in Δ'. Consequently, $op'\Delta'$ is one of the children of Δ'. But, since both $T\left(op_{i_{k'}} \ldots op_{i_{w+1}} \Delta_r\right)$ and $T\left(op' op_{i_{k'}} \ldots op_{i_{w+1}} \Delta_r\right)$ contain solutions, Δ' cannot qualify as dirty for its child $op'\Delta'$ under the first condition of the definition of dirty. A search of $T(op'\Delta')$ must, therefore, leave some dirty ancestor of Δ' marked.

Every subtree in $f(\Delta, T)$ must leave some dirty node marked. Since the dirty nodes in $\Delta_1, \ldots p(\Delta)$ are exactly the sufficient marks for $T(\Delta)$, exploration of $f(\Delta, T)$ must leave some node in $\psi_{\Downarrow}(\Delta)$ marked. □

Theorem 5.2 Let $f \in \mathcal{P} \cup \mathcal{A}$ and let Δ_r be the root of some $f(\Delta, T)$. Let op' be an inference operation applicable to Δ_r. If a nagger exhausts $T(op'\Delta_r)$ and generates an infeasible-choice message implicating operation op in the original search space, then, for any $\bar{\Delta} \in T(\Delta)$, if $\psi_{\Downarrow}\left(op\bar{\Delta}\right)$ is defined and nonempty, it must contain one of the following:

- One of the ancestors of Δ marked during the search of $T(op'\Delta_r)$
- Δ
- $\bar{\Delta}$

Proof: First, it can be shown that, either Δ or one of the nodes marked during the nagger's search of $T(op'\Delta_r)$ must be in $\psi_{\Downarrow}(op\Delta)$. This follows from the fact that, if $op\Delta$ is defined, it is possible to construct a transformation $f' \in \mathcal{P} \cup \mathcal{A}$ such that $f'(op\Delta, T) = T(op'\Delta_r)$. Theorem 5.1 demonstrates that exploring $f'(op\Delta, T)$ must leave some member of $\psi_{\Downarrow}(op\Delta)$ marked. Tableau Δ is the only node that can be marked while exploring $f'(op\Delta, T)$ but not while exploring $T(op'\Delta_r)$ in $f(\Delta, T)$.

Now it can be shown that for any node $\Delta' \neq \Delta$, if $\Delta' \in \psi_{\Downarrow}(op\Delta)$, Δ' must also be a member of $\psi_{\Downarrow}\left(op\bar{\Delta}\right)$. Since Theorems 4.2 and 4.4 guarantee that $T(op\Delta)$ contains no solutions, membership of Δ' in $\psi_{\Downarrow}(op\Delta)$ depends only on the inference operations applied at Δ' and its ancestors. Theorems 4.2 and 4.4 demonstrate that $T\left(op\bar{\Delta}\right)$ must also contain no solutions. Consequently, the membership of Δ' in

$\psi_{\Downarrow}\left(op\bar{\Delta}\right)$ depends only on the inference operations applied at Δ' and its ancestors. If $\Delta' \in \psi_{\Downarrow}(op\Delta)$, then $\Delta' \in \psi_{\Downarrow}\left(op\bar{\Delta}\right)$.

Finally, it can be shown that, if Δ is in $\psi_{\Downarrow}(op\Delta)$, either Δ or $\bar{\Delta}$ must be in $\psi_{\Downarrow}\left(op\bar{\Delta}\right)$. If Δ is a sufficient mark for $T(op\Delta)$ then it must qualify for membership in $\psi_{\Downarrow}(op\Delta)$ under one of the enumerated cases in Definition 5.2:

- Assume that Δ satisfies the first condition for membership in $\psi_{\Downarrow}(op\Delta)$. Let $\langle op_1, \ldots op_j\rangle$ be the sequence of inference operations deriving $op\bar{\Delta}$ from $\Delta_{\emptyset}$. Let $\langle i_1, \ldots i_k\rangle$ be the subsequence of $\langle 1, \ldots j\rangle$ such that $i \in \langle i_1, \ldots i_k\rangle$ if and only if $op_{i-1}\ldots op_1\Delta_{\emptyset} \notin \psi_{\Downarrow}\left(op\bar{\Delta}\right)$. Assume that $\Delta \notin \psi_{\Downarrow}\left(op\bar{\Delta}\right)$. Choose $k' \leq k$ as the maximum integer such that $op_{i_{k'}}op_{i_{k'}-1}\ldots op_1\Delta_{\emptyset}$ is an ancestor or Δ. The relationship demonstrated above between $\psi_{\Downarrow}(op\Delta)$ and $\psi_{\Downarrow}\left(op\bar{\Delta}\right)$ is reciprocal. Any ancestor of Δ appearing in $\psi_{\Downarrow}\left(op\bar{\Delta}\right)$ must also be a member of $\psi_{\Downarrow}(op\Delta)$. Thus, $i \in \langle i_1, \ldots i_{k'}\rangle$ if and only if $op_{i-1}\ldots op_1\Delta_{\emptyset} \notin (\psi_{\Downarrow}(op\Delta) - \{\Delta\})$. Since Δ achieves membership in $\psi_{\Downarrow}(op\Delta)$ via the first condition of Definition 5.2, $T\left(op_{i_{k'}}\ldots op_{i_1}\Delta_{\emptyset}\right)$ must contain a solution and $T\left(op\, op_{i_{k'}}\ldots op_{i_1}\Delta_{\emptyset}\right)$ must contain no solutions. If $\bar{\Delta} \notin \psi_{\Downarrow}\left(op\bar{\Delta}\right)$, then $op_{i_k} = op$ and $i_{k'}$ must be strictly less than i_k. Furthermore, the induction in the proof of Theorem 5.2 demonstrates that there must be solutions in $T\left(op\, op_{i_{k-1}}\ldots op_{i_1}\Delta_{\emptyset}\right)$. But, if there are no solutions in $T\left(op\, op_{i_{k'}}\ldots op_{i_1}\Delta_{\emptyset}\right)$, operator transposition demonstrates that there can be no solutions in $T\left(op\, op_{i_{k-1}}\ldots op_{i_1}\Delta_{\emptyset}\right)$. Thus, $\bar{\Delta}$ must be in $\psi_{\Downarrow}\left(op\bar{\Delta}\right)$.
- Assume that Δ satisfies the second condition for membership in $\psi_{\Downarrow}(op\Delta)$ and let β be the target branch for operation op. There must be some node $p^m(\Delta) \in \psi_{\Downarrow}(op\Delta)$ such that $m > 0$ and $p^m(\Delta) \in \psi(\Delta, \beta)$. Since $op\bar{\Delta}$ is defined, β must also be a branch in $\bar{\Delta}$, and node $p^m(\Delta)$ must also be in $\psi\left(\bar{\Delta}, \beta\right)$. As before, since neither $T(op\Delta)$ nor $T\left(op\bar{\Delta}\right)$ contain solutions, membership of $p^m(\Delta)$ in $\psi_{\Downarrow}(op\Delta)$ depends only on $p^m(\Delta)$ and the inference operations deriving it. Thus, $p^m(\Delta) \in \psi_{\Downarrow}\left(op\bar{\Delta}\right)$. Since op, operates on β when applied to either Δ or $\bar{\Delta}$, node $\bar{\Delta}$ qualifies for membership in $\psi_{\Downarrow}\left(op\bar{\Delta}\right)$ under the second case of Definition 5.2. □

Theorem 5.3 Let $f \in \mathcal{P}$ and let Δ be a node of T. If $T(\Delta)$ contains a solution that is derivable in the presence of the identical-ancestor refinement, then $f(\Delta, T)$ also contains a solution that survives the pruning of the identical-ancestor refinement.

Proof: Let Δ_s be a solution in $T(\Delta)$ of height k. Let Δ' be the tableau at the root of $f(\Delta, T)$. For Δ_s the proof of Theorem 3.1 exhibits a sequence $\left\langle op'_1, \ldots op'_j\right\rangle$ such that $op'_j \ldots op'_1 \Delta'$ is closed and of height k or less and any branch in $op'_j \ldots op'_1 \Delta'$ also appears in Δ_s with the same labeling. Since no branch in Δ_s contains two nodes with the same label, no branch in Δ'_s can contain two nodes with the same label.

Assume that, for some $i \in \{0, \ldots j-1\}$, two different nodes, n and $p^m(n)$ have identical labels in $op'_i \ldots op'_1 \Delta'$. When applied, operations in $\left\langle op_{i+1} \ldots op_j \right\rangle$ can change labels on existing nodes only through variable substitutions. Since the labels of n and $p^m(n)$ are identical $op_i \ldots op_1 \Delta'$, they must still be identical in $op_j \ldots op_1 \Delta'$. Since Δ'_s has no branches with two identically labeled nodes, all tableaux on the derivation path from Δ' to Δ'_s must survive the identical ancestor constraint. $\square$

Theorem 5.4 Let $f \in \mathcal{A}$ and let Δ be a node of T. If $T(\Delta)$ contains a solution that is derivable in the presence of the identical-ancestor refinement, then $f(\Delta, T)$ also contains a solution that is derivable in the presence of the identical-ancestor refinement.

Proof: Let Δ_s be a solution in $T(\Delta)$ of height k, and let Δ' be the root of $f(\Delta, T)$. The proof of Theorem 3.3 exhibits a closed tableau Δ'_s corresponding to Δ_s such that Δ'_s is of height k or less and derivable from Δ'. Furthermore, the trivial abstraction of Δ_s is an instance of Δ'_s. It must be shown that Δ'_s does not have a branch with two identically labeled nodes. Let $\bar{\Delta}_s$ be the trivial abstraction of Δ_s and assume that some branch β in Δ'_s contains two nodes labeled with literal l. Since $\bar{\Delta}_s$ is an instance of Δ'_s, β in $\bar{\Delta}_s$ must contain two nodes labeled $l\,\hat{\theta}$ for some substitution $\hat{\theta}$. But, the trivial abstraction is simply a uniform replacement of symbols with ground terms. Under the trivial abstraction, each distinct symbol is replaced with a distinct term. Thus, if two nodes of branch β are labeled identically in $\bar{\Delta}_s$, they must also be labeled identically in Δ_s. Since the identical-ancestor refinement forbs such branches in Δ_s, β in Δ'_s must not contain two nodes labeled l. As in Theorem 5.3, no tableau on the derivation path from Δ' to Δ'_s can have a branch with identically labeled nodes. $\square$

Theorem 5.5 (Representative Completeness) Let Δ be a node in T such that there are no associations between the branches of Δ and the cache when the search of $T(\Delta)$ begins. If $T(\Delta)$ contains a solution Δ_s that is pruned by the redundancy-avoidance cache, then $T(\Delta)$ must contain a solution Δ'_s that is not pruned by the cache and is reached before Δ_s. Furthermore, Δ_s and Δ'_s must have the same number of first-level nodes with the same labels.

Proof: The theorem is proven inductively. If Δ_i is a solution and $p^j(\Delta_i)$ is rejected by the redundancy-avoidance cache, then there exists some solution Δ_{i+1} such that, either Δ_{i+1} is found or some node $p^{j'}(\Delta_{i+1})$ is rejected by the redundancy-avoidance cache for some $j' \leq j$.

If node $p^j(\Delta_i)$ is rejected by the cache, let β be the branch of $p^j(\Delta_i)$ for which a repeated subproof is detected. Since no associations exist between branches and cache entries when the search of $T(\Delta)$ begins, the association with β responsible for rejecting $p^j(\Delta_i)$ must have been created at some earlier point in the search of $T(\Delta)$. Let Δ' be the node at which this association is created. If Δ_i is pruned because of β, the labels of $\mathit{leaf}(\beta)$ in Δ' and $p^j(\Delta_i)$ must be identical. Let Δ_a be the common ancestor of $p^j(\Delta_i)$ and Δ' with the greatest depth. Since the association persists from Δ' to $p^j(\Delta_i)$, the inference operations deriving Δ' and $p^j(\Delta_i)$ from Δ_a must operate only on branches containing β and must not include reductions with any ancestors of $\mathit{leaf}(\beta)$. Any substitutions applied in deriving either Δ' or $p^j(\Delta_i)$ from Δ_a must be the most general unifier for labels in $T(\mathit{leaf}(\beta))$ and literals of the theory. If any variable other than those in $T(\mathit{leaf}(\beta))$ is bound by such a substitution, then it is not a most general unifier. Thus, the inference from Δ_a to $p^j(\Delta_i)$ and Δ' must bind only those variables of Δ_a that appear in the label of $\mathit{leaf}(\beta)$. Since the labels of $\mathit{leaf}(\beta)$ are identical in $p^j(\Delta_i)$ and Δ', all labels of nodes outside $T(\mathit{leaf}(\beta))$ must be the same in $p^j(\Delta_i)$ and Δ'. Branch β must contain at least two nodes, and, therefore, all the first-level nodes of Δ' and $p^j(\Delta_i)$ must be labeled the same.

Since β is closed in Δ' and $p^j(\Delta_i)$ and all branches that are not a superset of β are identical in Δ' and $p^j(\Delta_i)$, the sequence of operations deriving a closed tableau from $p^j(\Delta_i)$ will also derive a closed tableau from Δ'. Call this solution Δ_{i+1}. Clearly, $p^j(\Delta_{i+1}) = \Delta'$ and Δ_{i+1} is labeled the same as Δ_i everywhere except $T(\mathit{leaf}(\beta)) - \{\mathit{leaf}(\beta)\}$. Since an association between β and a cache entry is established in Δ', Δ' must be visited during the search. Therefore, if any ancestor of Δ_{i+1} is discarded by the redundancy-avoidance cache it must be Δ' or a node derivable from Δ'.

This completes the inductive argument. For any solution Δ_i in $T(\Delta)$ that is barred by the redundancy-avoidance cache, there must be a previous solution Δ_{i+1} such that, either Δ_{i+1} is found or the search comes at least as close to Δ_{i+1} as it does to Δ_i. Since $T(\Delta)$ is finite in a depth-bounded search, for any solution $\Delta_0 = \Delta_s \in T(\Delta)$ that is not found because of the redundancy-avoidance cache, this induction must terminate with a solution that is discovered. Furthermore, across all solutions Δ_i there must be the same number of first-level nodes with the same labeling. $\square$

Bibliography

[ABCM88] P. E. Allen, S. Bose, Edmund M. Clarke, and S. Michaylov. Parthenon: A parallel theorem prover for non-horn clauses. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 764–765. Springer-Verlag, May 1988.

[ACF92] M. Avvenuti, P. Corsini, and G. Frosini. A distributed interpreter for inherent and/or parallelism. In P. Kacsuk and M. J. Wise, editors, *Implementations of Distributed Prolog*, pages 65–87. John Wiley & Sons, 1992.

[AK90] K. A. M. Ali and R. Karlsson. The Muse Or-parallel Prolog model and its performance. In S. Debray and M. Hermenegildo, editors, *1990 North American Logic Programming Conference*, pages 757–776. MIT Press, November 1990.

[AK91] H. Aït-Kaci. *Warren's Abstract Machine.* MIT Press, 1991.

[AL91] 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.

[ALS90] L. Alkalaj, T. Lang, and E. Shapiro. Fcp sequential abstract machine characteristics for the systems development workload. In S. Debray and M. Hermenegildo, editors, *1990 North American Logic Programming Conference*, pages 321–339. MIT Press, November 1990.

[AS88] L. Alkalaj and E. Shapiro. An architectural model for a flat concurrent Prolog processor. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming. Proceedings of the Fifth International Conference and Symposium*, pages 1277–1297. MIT Press, 1988.

[AS92] O. L. Astrachan and M. E. Stickel. Caching and lemmaizing in model elimination theorem provers. In D. Kapur, editor, *11th International*

Conference on Automated Deduction, pages 224–238. Springer-Verlag, June 1992.

[BdH+88] U. Baron, J. C. de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, Jean-Claude Syre, and H. Westphal. The parallel ecrc Prolog system pepsys: An overview and evaluation results. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 3, pages 841–850, 1988.

[BFGd92] J. Briat, M. Favre, C. Geyer, and J. Chassin de Kergommeaux. Opera: Or-parallel Prolog system on supernode. In P. Kacsuk and M. J. Wise, editors, *Implementations of Distributed Prolog*, pages 45–63. John Wiley & Sons, 1992.

[BFJO90] R. Butler, I. Foster, A. Jindal, and R. Overbeek. A high-performance parallel theorem prover. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 649–650. Springer-Verlag, July 1990.

[BGR88] B. Bergsten and R. Gonzalez-Rubio. A database accelerator and its languages. In *CONPAR 88*. Cambridge Unversity Press, September 1988.

[Bre88] R. Brena. Parallelizing a natural deduction system. In T. O'Shea and V. Sgurev, editors, *Artificial Intelligence III, Methodology, Systems, Applications*, pages 83–92. North-Holland, 1988.

[Bru78] 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.

[Brü94] S. Brüning. Detecting non-provable goals. In Alan Bundy, editor, *12th International Conference on Automated Deduction*, pages 222–236. Springer-Verlag, June/July 1994.

[BT88] P. Biswas and C.C. Tseng. Logdf: A data-driven abstract machine model for parallel execution of logic programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 3, pages 1059–1070, 1988.

[Bun83] A. Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1983.

[BY94] F. Bacchus and Q. Yang. Downward refinement and the efficiency of hierarchical problem-solving. *Artificial Intelligence*, 71:43–100, 1994.

[CA88] W. F. Clocksin and H. Alshawi. A method for efficiently executing Horn clause programs using multiple processors. *New Generation Computing*, 5:361–376, 1988.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252. Association for Computing Machinery, January 1977.

[CCC90] C. Codognet, P. Codognet, and M.M. Corsini. Abstract interpretation for concurrent logic languages. In S. Debray and M. Hermenegildo, editors, *1990 North American Logic Programming Conference*, pages 215–232. MIT Press, November 1990.

[CD85] 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.

[CDD85] J. H. Chang, A. M. Despain, and D. DeGroot. And-parallelism of logic programs based on static data dependency analysis. In *Digest of Papers of COMPCON, Spring '85*, pages 218–225, 1985.

[CFS89] P. Corsini, G. Frosini, and G. Speranza. The parallel interpretation of logic programs in distributed architectures. *Computer Journal*, 32(1):29–35, 1989.

[CG81] K. Clark and S. Gregory. A relational language for parallel programming. In *ACM Conference on Functional Languages and Computer Architecture*, pages 171–178, 1981.

[CG86] K. Clark and S. Gregory. Parlog: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, 1986.

[CG87] K. Clark and S. Gregory. Parlog: Parallel programming in logic. In E. Shapiro, editor, *Concurrent Prolog, Collected Papers*, volume 1, pages 84–139. MIT Press, 1987.

[CL73] C-L. Chang and R. C-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[Clo87] W. F. Clocksin. Principles of the DelPhi parallel inference machine. *Computer Journal*, pages 386–392, 1987.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[Con87a] J. Conery. Implementing backward execution in nondeterministic and-parallel systems. In J. L. Lassez, editor, *Logic Programming. Proceedings of the Fourth International Conference and Symposium*, volume 2, pages 633–653. MIT Press, 1987.

[Con87b] J. S. Conery. *Parallel Execution of Logic Programs*. Kluwer, 1987.

[Con92] J. S. Conery. The opal machine. In P. Kacsuk and M. J. Wise, editors, *Implementations of Distributed Prolog*, pages 159–185. John Wiley & Sons, 1992.

[Cra85] J. Crammond. A comparative study of unification algorithms for or-parallel execution of logic languages. *IEEE Transactions on Computers*, 34(10), 1985.

[dC84] J. C. de Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, 1984.

[Dd88] M. Dixon and J. de Kleer. Massively parallel assumption-based truth maintenance. In *Proceedings of the AAAI-88*, volume 1, pages 199–204. Morgan Kaufmann, August 1988.

[DeG84] D. DeGroot. Restricted and-parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 471–478. North-Holland, November 1984.

[DLO87] T. Disz, E. Lusk, and R. Overbeek. Experiments with or-parallel logic programs. In J. L. Lassez, editor, *Logic Programming. Proceedings of the Fourth International Conference and Symposium*, volume 2, pages 576–600. MIT Press, 1987.

[DM86] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[dP90] F. S. de Boer and C. Palamidessi. Concurrent logic programming: Asynchronism and language comparison. In S. Debray and M. Hermenegildo, editors, *1990 North American Logic Programming Conference*, pages 175–194. MIT Press, November 1990.

[DR92a] S. A. Delgado-Rannauro. Restricted and- and and/or-parallel logic computational models. In P. Kacsuk and M. J. Wise, editors, *Implementations of Distributed Prolog*, pages 121–141. John Wiley & Sons, 1992.

[DR92b] 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.

[DR92c] 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.

[EKL+89] W. Ertel, F. Kurfeß, R. Letz, X. Pandolfi, and J. Schumann. Partheo: A parallel inference machine. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89, Parallel Architectures and Languages Europe*, volume 1, pages 459–476. Springer-Verlag, June 1989.

[Ert90] 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.

[FD90] B. Fagin and M. Despain. The performance of parallel Prolog programs. *IEEE Transactions on Computers*, 39(12):1434–1445, 1990.

[FLSY74] S. Fleisig, D. Loveland, A. Smiley, and D. Yarmash. An implementation of the model elimination proof procedure. *Journal of the Association for Computing Machinery*, 21:124–139, January 1974.

[GBJ+90] A. Gupta, A. Banerjea, V. Jha, V. Bafna, and P. Bhatt. Parallel implementation of logic languages. In *CONPAR 90*. Springer-Verlag, September 1990.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[GJ90a] G. Gupta and Jayaraman. Optimizing and-or parallel implementations. In S. Debray and M. Hermenegildo, editors, *1990 North American Logic Programming Conference*, pages 605–623. MIT Press, November 1990.

[GJ90b] G. Gupta and B. Jayaraman. On criteria for or-parallel execution models of logic programs. In S. Debray and M. Hermenegildo, editors, *1990 North American Logic Programming Conference*, pages 737–756. MIT Press, November 1990.

[GS88] G. Giandonato and G. Sofi. Parallelizing logic programming based inference engines. In L. P. Kartashev and S. I. Kartashev, editors, *Proceedings, Third International Conference on Supercomputing*, pages 282–287. International Supercomputing Institute, 1988.

[GSC+90] A. Goto, T. Shinogi, T. Chikayama, K. Kumon, and A. Hattori. Processor element architecture for a parallel inference machine, PIM/p. *Journal of Information Processing*, 13(2):174–182, 1990.

[GW92] F. Giunchiglia and T. Walsh. A theory of abstraction. *Artificial Intelligence*, 57:323–389, 1992.

[HA84] R. Hasegawa and M. Amamiya. Parallel execution of logic programs based on dataflow concept. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 507–516. North-Holland, November 1984.

[Hau90] B. Hausman. Handling of speculative work in or-parallel Prolog: Evaluation results. In S. Debray and M. Hermenegildo, editors, *1990 North American Logic Programming Conference*, pages 721–736. MIT Press, November 1990.

[HCC88] B. Hausman, A. Ciepielewski, and A. Calderwood. Cut and side-effects in or-parallel Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 3, pages 831–840, 1988.

[Her86] 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-Verlag, July 1986.

[HG88] A. Harsat and R. Ginosar. Carmel-2: A second generation vlsi architecture for flat concurrent Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 3, pages 962–969, 1988.

[HN86] M. V. Hermenegildo and R. I. Nasr. Efficient management of backtracking in and-parallelism. In E. Shapiro, editor, *Third International Conference on Logic Programming*, pages 40–54. Springer-Verlag, July 1986.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Ibá88] M. B. Ibáñez. Parallel inferencing in first order logic based on the connection method. In T. O'Shea and V. Sgurev, editors, *Artificial Intelligence III, Methodology, Systems, Applications*, pages 149–157. North-Holland, 1988.

[Joh90] R. Johnsonbaugh. *Discrete Mathematics.* Macmillan, second edition, 1990.

[Kac90] P. Kacsuk. Prolog implementations on parallel computers. In *CONPAR 90*. Springer-Verlag, September 1990.

[Kac92] P. Kacsuk. Distributed data driven Prolog abstract machine. In P. Kacsuk and M. J. Wise, editors, *Implementations of Distributed Prolog*, pages 89–118. John Wiley & Sons, 1992.

[Kal87] L. Kalé. The reduce-or process model for parallel evaluation of logic programs. In J. L. Lassez, editor, *Logic Programming. Proceedings of the Fourth International Conference and Symposium*, volume 2, pages 616–632. MIT Press, 1987.

[KKS95] H. Kautz, M. Kerns, and B. Selman. Horn approximations of empirical-data. *Artificial Intelligence*, 74:129–145, 1995.

[KL86] V. Kumar and Y-J. Lin. A framework for intelligent backtracking in logic programs. In Kesav V. Nori, editor, *Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, pages 108–123. Springer-Verlag, December 1986.

[KL87] 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 1987.

[Kno94] C. A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68:243–302, 1994.

[Kor85] R. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Kow79] R. A. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, 1979.

[KR90] L. Kalé and B. Ramkumar. Joining and parallel solutions in and/or parallel systems. In S. Debray and M. Hermenegildo, editors, *1990 North American Logic Programming Conference*, pages 624–641. MIT Press, November 1990.

[KR92] L. Kalè and B. Ramkumar. The reduce-or process model for parallel logic programming on non-shared memory machines. In P. Kacsuk and M. J. Wise, editors, *Implementations of Distributed Prolog*, pages 187–212. John Wiley & Sons, 1992.

[KS94] H. Kautz and B. Selman. An empirical evaluation of knowledge compilation by theory approximation. In *Proceedings of the AAAI-94*, volume 1, pages 155–161. MIT Press, August 1994.

[Kur90] F. Kurfeß. Potentiality of parallelism in logic. In *Parallelization in Inference Systems*, pages 3–25. Springer-Verlag, December 1990.

[KW89] D. J. Kleikamp and R. W. Wilkerson. Associative-commutative unification in a multicomputer environment. In *The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, volume 2, pages 901–908, March 1989.

[LBD⁺88] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The aurora or-parallel Prolog system. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 3, pages 819–830, 1988.

[Lev86] J. Levy. A GHC abstract machine and instruction set. In E. Shapiro, editor, *Third International Conference on Logic Programming*, pages 157–171. Springer-Verlag, July 1986.

[LK88] Y-J. Lin and V. Kumar. Performance of and-parallel execution of logic programs on a shared-memory multiprocessor. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 3, pages 851–860, 1988.

[LKL86] Y-J. Lin, V. Kumar, and C. Leung. An intelligent backtracking algorithm for parallel execution of logic programs. In E. Shapiro, editor, *Third International Conference on Logic Programming*, pages 55–68. Springer-Verlag, July 1986.

[LMG92] 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.

[LO88] E. Lusk and R. Overbeek. Exploitation of parallelism in prototypical deduction problems. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 333–343. Springer-Verlag, May 1988.

[Lov72] D. W. Loveland. A unifying view of some linear herbrand procedures. *Journal of the Association for Computing Machinery*, 19:366–384, 1972.

[Lov78] D. W. Loveland. *Automated Theorem Proving: a Logical Basis.* North-Holland, 1978.

[LSBB94] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. Setheo: A high-performance theorem prover. *Journal of Automated Reasoning*, 13(3):183–212, 1994.

[Lus90] E. Lusk. Experiments with roo, a parallel automated deduction system. In *Parallelization in Inference Systems*, pages 139–162. Springer-Verlag, December 1990.

[Mas87] S. Y. Maslov. *Theory of Deductive Systems and Its Applications.* MIT Press, 1987.

[Mel86] C. S. Mellish. Abstract interpretation of Prolog programs. In E. Shapiro, editor, *Third International Conference on Logic Programming*, pages 463–474. Springer-Verlag, July 1986.

[MKKC86] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[MKO84] K. Murakami, T. Kakuta, and R. Onai. Architectures and hardware systems: Parallel inference machine and knowledge base machine. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 18–36. North-Holland, November 1984.

[MOTA+84] T. Moto-Oka, H. Tanaka, H. Aida, K. Hirata, and T. Maruyama. The architecture of a parallel inference engine - pie -. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 479–488. North-Holland, November 1984.

[Nai88] L. Naish. Parallelizing nu-Prolog. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming. Proceedings of the Fifth International Conference and Symposium*, pages 1546–1564. MIT Press, 1988.

[NS93] A. Nerode and R. A. Shore. *Logic for Applications.* Springer-Verlag, 1993.

[OM87] H. Ohwada and F. Mizoguchi. Managing search in parallel logic programmin. In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Logic Programming '87, Proceedings of the 6th Conference*, pages 148–177. Springer-Verlag, June 1987.

[PK95] I. Pramanick and J. G. Kuhl. An inherently parallel method for heuristic problem-solving. i. general framework. *IEEE Transactions on Parallel and Distributed Systems*, 6(10):1006–1015, 1995.

[Pla81] D. A. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16:47–108, 1981.

[Pla84] D. A. Plaisted. The occur-check problem in Prolog. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 272–280. IEEE Computer Society, February 1984.

[PMCA86] L. M. Pereira, L. Monteiro, J. Cunha, and J. N. Aparı'cio. Delta Prolog: A distributed backtracking extension with events. In E. Shapiro, editor, *Third International Conference on Logic Programming*, pages 69–83. Springer-Verlag, July 1986.

[PN84] L. M. Pereira and R. Nasr. Delta-Prolog: A distributed logic programming language. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 283–291. North-Holland, November 1984.

[Pos43] E. L. Post. Formal reduction of the general combinatorial decision problem. *American Journal of Mathematics*, 65(2):197–215, 1943?

[PSS92] C. Percebois, W. Signès, and L. Selle. An actor-oriented computer for logic and its applications. In P. Kacsuk and M. J. Wise, editors, *Implementations of Distributed Prolog*, pages 213–235. John Wiley & Sons, 1992.

[RK91] E. Rich and K. Knight. *Artificial Intelligence.* McGraw-Hill, second edition, 1991.

[Sar87] V. A. Saraswat. The concurrent logic programming language CP: Definition and operational semantics. In *13th ACM Symposium on Principles of Programming Languages*, pages 49–62. Association for Computing Machinery, 1987.

[Sar88] V. A. Saraswat. A somewhat logical formulation of clp synchronisation primitives. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming. Proceedings of the Fifth International Conference and Symposium*, pages 1298–1314. MIT Press, 1988.

[SC95] M. Schaerf and M. Cadoli. Tractable reasoning via approximation. *Artificial Intelligence*, 74:249–310, 1995.

[SE94] A. M. Segre and C. Elkan. A high performance explanation-based learning algorithm. *Artificial Intelligence*, 1994.

[Sha87a] E. Shapiro. Concurrent Prolog: A progress report. In E. Shapiro, editor, *Concurrent Prolog, Collected Papers*, volume 1, pages 157–187. MIT Press, 1987.

[Sha87b] E. Shapiro. A subset of concurrent Prolog and its interpreter. In E. Shapiro, editor, *Concurrent Prolog, Collected Papers*, volume 1, pages 27–83. MIT Press, 1987.

[Sha89] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[Shi92] R. Shinghal. *Formal Concepts in Artificial Intelligence.* Chapman & Hall, 1992.

[Sib90] N. F. Sibai. Parallel unification: Theory and implementations. In *Parallelization in Inference Systems*, pages 51–81. Springer-Verlag, December 1990.

[SJ94] C. B. Suttner and M. R. Jobmann. Simulation analysis of static partitioning with slackness. In H. Kitano, V. Kumar, and C. B. Suttner, editors, *Parallel Processing for Artificial Intelligence 2*, volume 15, pages 93–105. North-Holland, 1994.

[SK91] B. Selman and H. Kautz. Knowledge compilation using horn approximations. In *Proceedings of the AAAI-91*, volume 2, pages 904–909, 1991.

[SKL90] V. A. Saraswat, K. Kahn, and J. Levy. Janus: A step towards distributed constraint programming. In S. Debray and M. Hermenegildo, editors, *1990 North American Logic Programming Conference*, pages 431–446. MIT Press, November 1990.

[SL90a] J. 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.

[SL90b] J. K. Slaney and E. Lusk. Parallelizing the closure computation in automated deduction. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 28–39. Springer-Verlag, July 1990.

[SLK90] J. 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. Summary.

[SRV88] Z. Somogyi, K. Ramamohanarao, and J. Vaghani. A stream and-parallel execution algorithm with backtracking. In R. A. Kowalski and

K. A. Bowen, editors, *Logic Programming. Proceedings of the Fifth International Conference and Symposium*, pages 1142–1159. MIT Press, 1988.

[SS93] A. M. Segre and D. Scharstein. Bounded-overhead caching for definite-clause theorem proving. *Journal of Automated Reasoning*, 11:83–113, 1993.

[SS94] D. Sturgill and A. M. Segre. A novel asynchronous parallelism scheme for first-order logic. In Alan Bundy, editor, *12th International Conference on Automated Deduction*, pages 484–498. Springer-Verlag, June/July 1994.

[SS95] G. Sutcliffe and C. B. Suttner. Atp system results for the tptp problem library. Technical Report 96/16, James Cook University, Australia, 1995.

[SSY94] G. Sutcliffe, C. Suttner, and T. Yemenis. The tptp problem library. In Alan Bundy, editor, *12th International Conference on Automated Deduction*, pages 252–266. Springer-Verlag, June/July 1994.

[ST85] 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.

[Sti88] M. E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4:353–380, 1988.

[Sti89] M. E. Stickel. A Prolog technology theorem prover: A new exposition and implementation in Prolog. Technical Report 464, SRI International, June 1989.

[SYH+87] K. Shibayama, M. Yamamoto, H. Hirata, Y. Kanoh, T. Sanetoh, and H. Hagiwara. Kpr: A logic programming language-oriented parallel machine. In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Logic Programming '87, Proceedings of the 6th Conference*, pages 113–131. Springer-Verlag, June 1987.

[Tan86] H. Tanaka. A parallel inference machine. *Computer*, pages 48–54, 1986.

[Ter88] M. Terada. Implementation of full GHC by communicating processes. In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Logic Programming '88, Proceedings of the 7th Conference*, pages 122–135. springer, April 1988.

[TF86] A. Takeuchi and K. Furukawa. Parallel logic programming languages. In E. Shapiro, editor, *Third International Conference on Logic Programming*, pages 242–254. Springer-Verlag, July 1986.

[TF87] A. Takeuchi and K. Furukawa. Parallel logic programming languages. In E. Shapiro, editor, *Concurrent Prolog, Collected Papers*, volume 1, pages 188–201. MIT Press, 1987.

[Tic91] E. Tick. *Parallel Logic Programming*. MIT Press, 1991.

[TK86] K. Takahashi and T. Kanamori. On parallel programming methodology in GHC. In E. Wada, editor, *Logic Programming '86, Proceedings of the 5th Conference*, pages 92–103. Springer-Verlag, June 1986.

[TK94] E. Tick and M. Korsloot. Determinacy testing for nondeterminate logic programming-languages. *ACM Transactions on Programming Languages and Systems*, 16(1):3–34, 1994.

[TL87] P. Tinker and G. Lindstrom. A performance-oriented design for or-parallel logic programming. In J. L. Lassez, editor, *Logic Programming. Proceedings of the Fourth International Conference and Symposium*, volume 2, pages 601–615. MIT Press, 1987.

[TM87] T. Tarui and T. Maruyama. A preliminary evaluation of parallel inference machine for stream parallel languages. In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Logic Programming '87, Proceedings of the 6th Conference*, pages 134–147. Springer-Verlag, June 1987.

[UC90] K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *Computer Journal*, 33(6):494–500, 1990.

[Ued87] K. Ueda. Guarded horn clauses. In E. Shapiro, editor, *Concurrent Prolog, Collected Papers*, volume 1, pages 140–156. MIT Press, 1987.

[VG92] A. Verden and H. Glasser. An and-parallel distributed Prolog executor. In P. Kacsuk and M. J. Wise, editors, *Implementations of Distributed Prolog*, pages 143–157. John Wiley & Sons, 1992.

[VY84] P. J. Voda and B. Yu. Rf-maple: A logic programming language with functions, types, and concurrency. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 341–347. North-Holland, November 1984.

[War83] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, October 1983.

[War87] D. H. D. Warren. The sri model for or parrallel execution of Prolog. abstract design and implementation issues. In *1987 IEEE Symposium on Logic Programming*, pages 92–102. IEEE Press, August 1987.

[Win92] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, third edition, 1992.

[WM88] L. Wos and W. W. McCune. Challenge problems focusing on equality and combinatory logic: Evaluating automated theorem-proving programs. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 714–729. Springer-Verlag, May 1988.

[YA86] R. Yang and H. Aiso. P-Prolog: A parallel logic language based on exclusive relation. In E. Shapiro, editor, *Third International Conference on Logic Programming*, pages 255–269. Springer-Verlag, July 1986.

[YCS90] G. Yaoqing, S. Chengzheng, and H. Shouren. Study of a parallel inference machine for parallel execution of logic programs. In *CONPAR 90*. Springer-Verlag, September 1990.