EFFICIENT COMBINATORIAL SEARCH ALGORITHMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Chee Fen Yu

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 1986

EFFICIENT COMBINATORIAL SEARCH ALGORITHMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Chee Fen Yu

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 1986

ACKNOWLEDGMENTS

I wish to express my thanks to my mentor, Professor Benjamin W. Wah. I also wish to thank Professor Philip H. Swain, Professor Jose A. Fortes, Professor John S. P. Wang, and the late Professor King-Sun Fu for their unstinting service on my thesis committee.

In addition, I gratefully acknowledge the support of the National Science Foundation through grants ECS81-05968 and DMC85-19649, and CIDMAC, a research unit of Purdue University, sponsored by Purdue University, Cincinnati Milicron Corporation, Control Data Corporation, Cummins Engine Company, Ransburg Corporation, and TRW.

TABLE OF CONTENTS

Page

APPENDICES

LIST OF TABLES

Table Page

Table Page

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

Yu, Chee Fen. Ph.D., Purdue University. December 1986. Efficient Combinatorial Search Algorithms. Major Professor: Benjamin W. Wah.

The search for acceptable solutions in a combinatorially large problem space is an important problem in the fields of artificial intelligence, operations research, and computer science. Ways of improving the efficiency of this search process includes reducing the number of nodes searched by the process and reducing the average time to search each node.

One way to reduce the number of nodes searched is to use an appropriate selection scheme. Best-first branch-and-bound (B&B) algorithms generally expand less nodes than depth-first B&B algorithms. However, a best-first B&B algorithm requires exponential memory space. A depth-first B&B algorithm requires linear memory space. In a two-level memory system, this difference in space requirements can cause the average expansion time of each node to differ for the two B&B strategies. This thesis presents results on the efficiency of the B&B strategies in a two-level memory. The best B&B strategy depends on the characteristics of the problem domain as well as the characteristics of the two-level memory. A best-first B&B algorithm should be used when it expands much less nodes than the corresponding depth-first B&B algorithm, and when the secondary memory is very fast. A depth-first B&B algorithm should be used when it expands approximately the same number of nodes as the corresponding best-first B&B algorithm, and when the secondary memory is very slow. The choice for intermediate situations is not so clear. One result of

our research is the modified B&B algorithm, which can be used in these intermediate situations. The modified B&B algorithm has been designed to match the characteristics of a two-level memory system.

Dominance relations prune unnecessary nodes in search graphs and so reduce the number of nodes searched by the search process. There is no systematic procedure to derive dominance relations because they are problem-dependent. A possible approach to this problem is the use of knowledge-based techniques. This thesis also contains a study of the machine learning of dominance relations. Among the results are a representation for optimization problems that exposes useful domain information, a classification of dominance relations that identifies the useful forms of dominance relations, and the applicable learning mechanisms. This research also includes a study of a system that learns dominance relations by experimentation. A prototype of this system has been able to learn dominance relations for several scheduling and mathematical programming problems.

CHAPTER 1
INTRODUCTION

1.1 Introduction and Motivation

Many problems in engineering and science involve finding one or more correct solutions among a set of possible solutions. The set of possible solutions constitutes the *problem space* for the problem.

A function $f(n)$ is of $O(g(n))$ when there exists a constant c such that $|f(n)| \leq c\,|g(n)|$ for all value of $n \geq 0$. A complexity function has polynomial complexity if the complexity function is $O(p(n))$ for some polynomial function p, where n denotes the problem size. Any complexity function that cannot be so bounded has exponential complexity. This definition of exponential complexity includes certain non-polynomial functions like $n^{\log n}$ which are not normally regarded as polynomial functions. A problem space is said to be *combinatorially large* if the number of possible solutions to the problem is of exponential complexity.

If the knowledge of the problem is *complete* in terms of the objects, properties, and relations relevant to the solution of the problem, then we can construct a rigorous computation model of the problem and solve it by *evaluation*. If the knowledge of the problem is *incomplete*, then it will be necessary to try some or all of the possible solutions in the problem space until the desired solution or set of solutions is found. In other words, a *search* of the problem space is required. Thus, a *combinatorial search problem* is a problem whose solution involves a search in a combinatorially large problem space and a *combinatorial search algorithm* is a search algorithm used to solve combinatorial search problems. Examples of combinatorial search problems include theorem proving, game playing, evaluating a logic program, solving a database query, designing a computer system, assigning registers for a compiler, finding the shortest path in a graph, evaluating the maximum flow in a network, solving a mathematical programming problem, and searching for a permutation order to sort a set of numbers. These problems exist in a large spectrum of engineering and science applications, including artificial intelligence and operations research.

Efficient search methods are very important for combinatorial search problems. An efficient search method requires significantly less time to solve any non-trivial instance of a combinatorial search problem than an inefficient search method. Often this difference in time requirements determines whether or not a problem instance may be solved. Table 1.1 is a dramatic illustration of the importance of efficient search algorithms for combinatorial search problems. It compares the growth rates among several complexity functions, where the functions express execution time in microseconds. Notice the much more explosive growth rates of the two exponential complexity functions.

Even more revealing is an examination of the effects of improved computer technology on algorithms having these time complexity functions. Table 1.2 shows how the largest problem instance solvable in one hour would change if we had a computer 100 or 1000 times faster than the present machine. Observe that with the 2^n algorithm a thousand-fold in computing speed only adds 10 to the size of the largest problem instance we can solve in an hour, whereas with the N^5 algorithm this size almost quadruples.

Before we consider the problem of improving the efficiency of combinatorial search algorithms, we will introduce some of the terminology and basic concepts of combinatorial search algorithms.

1.2 Terminology

Combinatorial search problems are divided into decision problems and optimization problems. A *decision problem* seeks any solution that satisfies a given set of constraints while an *optimization problem* seeks the solution that satisfies the set of constraints as well as optimizes a given objective function. A *feasible solution* is any solution that satisfies the given set of constraints and an *optimal solution* is any feasible solution that optimizes the objective function.

A search problem can be represented as a *search graph*. By convention, the places in a search graph are called *nodes* and the connections between nodes are called *links*. Each search graph has a *root node* and one or more *terminal nodes*. No link enters the root node while no link leaves a terminal node. A special case of the search graph is the *search tree* in which each node except the root has exactly one link entering it. A search graph can always be transformed into a search tree by duplicating common nodes. By convention, the links in a search tree are called *branches*. It is common to talk about trees

Table 1.1 Comparison of several polynomial and exponential time complexity functions.

Time Complexity Function	Size n					
	10	20	30	40	50	60
n	0.00001 second	0.00002 second	0.00003 second	0.00004 second	0.00005 second	0.00006 second
n^2	0.0001 second	0.0004 second	0.0009 second	0.0016 second	0.0025 second	0.0036 second
n^3	0.001 second	0.008 second	0.027 second	0.064 second	0.125 second	0.216 second
n^5	0.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	0.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	0.059 second	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{13} centuries

Table 1.2 Effect of improved technology on several polynomial and exponential time algorithms.

Time Complexity Function	Size of Largest Problem Instance Solvable in 1 Hour		
	With Present Computer	With Computer 100 Times Faster	With Computer 1000 Times Faster
n	N_1	$100\ N_1$	$1000\ N_1$
n^2	N_2	$10\ N_2$	$31.6\ N_2$
n^3	N_3	$4.64\ N_3$	$10\ N_3$
n^5	N_4	$2.5\ N_4$	$3.98\ N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$

using terms borrowed from genealogy. Branches directly connect *parents* with *children*. A pair of nodes has the *ancestor-descendant* relationship if there is a chain of branches between the two. If the number of children is always the same for every node that has children, that number is called the *branching factor*. Drawing in the children of a node is called *expanding* the node. Nodes are said to be *active* until they are expanded, whereupon they become *dormant*. The set of active nodes is usually kept in an ordered list known as the *active list*.

Each *branch* in a search tree represents the assignment of a value to an unassigned parameter. For example, the unassigned parameters in 0/1-th knapsack problem are the set of unassigned items. Expansion of a node involves the selection of an unassigned item and the creation of two alternatives: (a) include the item in the knapsack, thereby decreasing the capacity of the knapsack by the size of the item and increasing its profit by the value of the item, and (b) exclude the item from the knapsack.

Each node in a search tree, except the terminal nodes, can be either an AND node or an OR node. An AND node is a problem reduction from the problem represented by this node to multiple smaller subproblems represented by its children. To solve the problem represented by an AND node, all the subproblems represented by its children must be solved. An example of an AND node is one that adds the solutions from all the subtrees expanded from this node. In contrast, an OR node represents a state-space approach that transforms the state represented by this node, by operators represented by the arcs, to states represented by its children. An OR node is solved if any of its children can be solved. The need to search for the desired solution is due to the presence of OR nodes.

1.3 Combinatorial Search Procedures

There are two types of combinatorial search procedures: problem-specific procedures and general procedures. Problem-specific procedures are tailored to the problem characteristics. Examples of problem-specific procedures include polynomial-time approximation algorithms with guaranteed bounds [GaJ79, Sah77], polynomial-time algorithms that optimize the average performance [Smi84] and algorithms that are guaranteed to yield optimal or near-optimal solutions with a given probability [Kar76]. However, experience has shown that efficient algorithms for slightly different problems can be very different and that it is very difficult to design good problem-specific algorithms for most

problems. We do not study problem-specific procedures in this thesis.

General procedures are applicable to a wide range of problems [KoS74, Iba76b, Sek81]. but may also incorporate some problem-specific knowledge. Figure 1.1 shows some of these general procedures.

The first group of general procedures are appropriate for problems involving an adversary: *minimax search*, *alpha-beta pruning*, *progressive deepening*, *heuristic pruning*, and *heuristic continuation*. They are common in programs that play board games and use techniques that depend on the presence of an adversary. We will not study these procedures in this thesis.

The second group of procedures consists of simple, basic procedures which can only be used for decision problems. This group of procedures includes *depth-first search*, *hill climbing*, *breadth-first search*, *beam search*, and *best-first search*. Each of these procedures has the following basic form:

```
insert root node into active list;
while active list is not empty or goal has not been reached [
    if first element in active list is not the goal node then[
        remove first element from the active list;
        expand the first element;
        add the first element's children, if any, to the active list; ] ]
if goal node has been found then announce success else announce failure
```

The different procedures differ mainly in the way they add the first element's children to the active list. A depth-first search always adds them to the front of the active list but a breadth-first search always adds them to the end of the active list. Hill climbing procedures use a heuristic measure of the distance from the node to its nearest solution node (i.e., *goal distance*). A hill climbing procedure sorts the first element's children by their heuristic values before adding these children to the front of the active list. Beam search is like breadth-first search but only expands the best w nodes (by the estimated remaining distance) at each level. A best-first search orders the entire active list by the estimated goal distance.

The relative efficiencies of these methods depends on the nature of the problem space. A depth-first search is good when blind alleys do not get too deep. A breadth-first search is good when the number of alternatives at the choice points are not too large. Hill climbing is good when there is a natural measure of goal distance and a good choice is likely to be among the promising choices at each choice point. Beam search is good when there is a natural measure of goal distance and a good path is likely to be among the promising

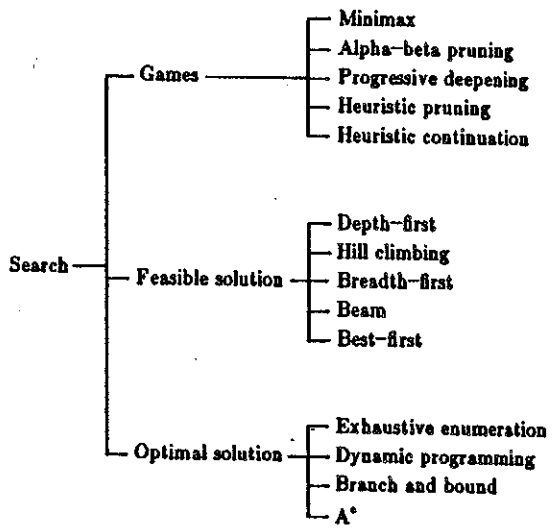


Figure 1.1 Search procedures.

partial paths at all levels. Best-first search is good when there is a natural measure of goal distance and a good path may look bad at shallow levels.

The final group of general procedures finds the best solution. This group includes *exhaustive enumeration, discrete dynamic programming* [BeD62, DrL77, KaH67, Iba73, Whi69], *branch and bound (B&B)* [LaW66, Mit70], and *A** [Nil71].

Exhaustive enumeration tries all possible solutions. The other procedures discards paths that are known not to lead to the optimal solution. Thus, these procedures have the following basic form:

```
insert root node into active list;
while active list is not empty or goal has not been reached [
    if first element in active list is not the goal node then[
        remove first element from the active list;
        expand the first element;
        prune unnecessary nodes, if any;
        add the first element's children, if any, to the active list; ] ]
if goal node has been found then announce success else announce failure
```

The *dynamic programming principle* [BeD62] states that the path from a node, S, to a non-terminal node, I, does not affect the path from node I to a solution node, G. Consequently when looking for the best path from S to G, all paths through the intermediate node, I, other than the best path from S to I, can be ignored. Dynamic programming procedures may use depth-first, or breadth-first, or best-first organization of the active list. The key concept in branch-and-bound algorithms is the *lower bound value*, which is a heuristic underestimate of the value of the best solution that can be derived from a node. No further work needs to be done on a node whose lower-bound value exceeds some known solution because the value of the best solution derived from this node cannot be smaller than its underestimate. This principle is known as the *lower-bound test* [LaW66]. The method used to order the active list leads to the identification of three different B&B algorithms: depth-first B&B, breadth-first B&B, and best-first B&B. The A* algorithm combines the best-first B&B algorithm with the dynamic programming principle. The B&B procedures and dynamic programming algorithms have been extended to include general pruning rules known as *dominance relations* [BeD62, AHU74, Iba77a].

Exhaustive enumeration is only useful for small problem spaces. Dynamic programming is good when many different paths reach common non-terminal nodes in the search tree. Branch-and-bound search is good when there is a

good lower-bound. The A* procedure is good when both the basic B&B procedure and dynamic programming are good. Both A* and dynamic programming may be viewed as a special case of the B&B search with dominance relations [KuK83].

1.4 Improving the Efficiency of Search Procedures

A comparison of the general procedures described in Section 1.3 shows that the more efficient procedure is generally one that searches a smaller portion of the problem space, i.e., the procedure with the smaller search tree (or graph). Ways of reducing the size of the search tree (or graph) include the use of appropriate representation, the use of an an appropriate selection scheme, and the use of appropriate pruning schemes. Another way to reduce the time required to solve an instance of the problem is to reduce the time required to expand each node. As suggested by Table 1.2 in Section 1.1, this method is only useful for procedures that have reasonable time complexities. It is not very useful if used to extend the solvable problem size.

A problem can often be represented in multiple forms. For example, the 0/1-knapsack problem may be represented by either an OR-tree and solved with a B&B procedure [HoS78] or an AND/OR-tree and solved by dynamic programming. It is possible for different representations to lead to procedures with drastically different time complexities. A dramatic example of this disparity in time complexities is the checkerboard and dominos problem. The upper left and lower right squares are removed from a n-by-n checkerboard. Each domino is one square by two squares in size. The problem is to decide whether or not the checkerboard can be covered completely by dominos. An OR-tree description leads to search procedures that try different arrangements of the dominos on the checkerboard. Such procedures have exponential complexity. This problem can as be represented as the repeated removal of two adjacent squares from a board consisting of n^2-2 red squares and n^2 black squares. This representation leads to a trivial solution of the problem.

Although efficient representations have been established for many search problems, the general question of deriving the optimal representation is an open question, especially the kinds of innovative problem descriptions which can lead to dramatic improvements in the effort required to solve a problem. We do not explicitly study the problem of the optimal representation for a particular problem. However we will investigate some representation issues in conjunction with the other ways of reducing the size of the state tree.

The selection scheme orders the elements of the active list and thus chooses nodes for expansion (which is always the first element in the active list). The three basic selection schemes are depth-first selection, breadth-first selection, and best-first selection. For problems with a suitable lower-bound function, a best-first scheme always has the smallest search tree (or graph) if no node has a lower-bound value equal to the optimal solution value [LaW66, LiW84, Pea84]. B&B procedures commonly use relaxation-guided lower-bound functions. A relaxation of a discrete optimization problem is a problem with the same set of inputs and a set of feasible solutions that is a superset of the set of feasible solutions for the discrete optimization problem, and has a cost function such that common feasible solutions have the same cost in both problems. A relaxation-guided lower-bound function is a fast algorithm for solving the relaxed problem. Such lower-bound functions are problem dependent and are very difficult to design because they require extensive knowledge about the different types of problems.

A pruning scheme identifies and eliminates *ineffective nodes* known not to lead to a solution better than another existing node. One pruning scheme is the dynamic programming principle-if two nodes have identical states, then the search procedure needs only expand one of the two nodes.

Another pruning scheme involves dominance relations [BeD62, AHU77, Iba77, HoS78]. Let subtree T_i be the subtree rooted at node N_i. If N_i dominates N_j, then T_i contains a solution node with a value that is no worse than the value of the best solution in T_j. The converse is not true. A solution in T_i no worse than the best solution in T_j does not imply the presence of a dominance relation between N_i and N_j. Good dominance relations can result in a search tree of polynomial complexity.

As an example, consider the problem of finding the shortest path between city S and city T. Let P_1 and P_2 be paths from S to an intermediate city N and P_3 be a path from N to T. P_1 dominates P_2 if P_1 is shorter than P_2. For each P_3, the path with length P_1+P_3 is shorter than the path with length P_2+P_3. Without the above dominance relations, the shortest-path problem requires exponential time.

The lower-bound test used in B&B algorithms is a dominance relation. The pruning power of the lower-bound tests depends on the quality of the lower-bound function [WaY85]. In the lower-bound test, the best solution found so far dominates all nodes with lower bounds not greater than its solution value. Other more general dominance relations have also been used [HoS78], but there is no systematic procedure to derive dominance relations

because they are problem-dependent. Identifying and encoding problem-specific dominance relations is a painstaking and difficult process, particularly for complex problems. One possible approach to this problem is to use artificial intelligence techniques, specifically, machine learning techniques.

The average time to expand a node can be reduced by reducing the mismatch between the algorithm and the architecture. For parallel computers, this requires the design of parallel search algorithms which can take advantage of the many processors. This has been studied by others [WaM84, LiW85, Li85] and will not be studied in this thesis.

For many serial computers, the desire to have memories of large capacity and moderate cost have resulted in the use of multi-level memories. In such memories, the cost of accessing an item of information depends on its location. The effects of this architectural feature has not been considered in previous work in the design of search procedures. It can affect the choice of the appropriate selection scheme as well as the implementation of the pruning scheme. The B&B algorithms provide a convenient context for studying this issue since the set of B&B algorithms includes all three selection schemes.

In short, the objective of this thesis is to improve the efficiency of serial combinatorial search algorithms. This objective is achieved by studying effective B&B algorithms in a two-level memory environment and by studying machine learning of dominance relations.

1.5 Organization of this Thesis

The remainder of this thesis is organized into four chapters. The next chapter deals with the question of efficient B&B algorithms in a two-level memory. This chapter is followed by two chapters dealing with machine learning of dominance relations. The final chapter contains conclusions and suggestions for future work.

Chapter 2 attempts to find the best B&B algorithm on a two-level memory system. We first establish, in two steps, that the best-first B&B algorithm is not always the best B&B algorithm on a two-level memory system in two steps. We will show that the difference between the number of nodes expanded by the best-first B&B algorithm and the number of nodes expanded by the depth-first B&B algorithm for a given problem instance can take on a wide range of values depending on the accuracy of the lower-bound function. We will then show that for some values of this difference, the depth-first B&B algorithm is better than the best-first B&B algorithm. Having shown that the best-first B&B

algorithm is not always the best choice, we then seek to develop guidelines for choosing the appropriate B&B algorithm. This includes the development of a new B&B algorithm, the modified B&B algorithm, which can be used when neither the best-first B&B algorithm nor the depth-first B&B algorithm is particularly suitable.

Chapter 3 considers task of learning dominance relations. The methodology used follows the caveats of a good methodology in artificial intelligence as outlined by Winston [Win84]. We have focused this study on the *number optimization problems*. By definition, some objects in a number optimization problem must be numbers, and the magnitude of the largest number cannot be bounded by a product of a polynomial and the number of symbols used to describe the problem. One result of applying this methodology is a problem representation which exposes semantic information about the problem. Another result is a classification of dominance relations that uncovers constraints in the domain of dominance relations. These constraints permit some classes of dominance relations to be eliminated from consideration. We have also identified the applicable learning mechanisms, and characterized their strengths, areas of application, and limitations. The primary learning mechanism is experimentation while the secondary learning mechanisms are analogy and deduction. The study of each mechanism is a major research problem in itself and we have concentrated on the primary mechanism.

Chapter 4 discusses a prototype program which learns dominance relations by experimentation. This program uses a generate-and-test paradigm based upon heuristic constraints derived from manipulations of the standard descriptions as well as constraints in the space of relations to reduce the number of candidate relations proposed and tested. In addition, the program is able to automatically define and manipulate new variable symbols. This chapter concludes with a summary of the problems to which this program has been applied and the results that we have obtained.

1.6 Contributions of this Thesis

The work in this thesis has contributed to a better understanding about B&B algorithms in a two-level memory. It shows a need to revise the belief that the best-first B&B algorithm is always the best B&B algorithm. This belief is true only in a single-level memory system. In a two-level memory system, the best B&B algorithm depends on the characteristics of the problem and the memory system. We have also developed a new B&B algorithm that is

useful in situations where both the depth-first and best-first B&B algorithms do not perform well. This new B&B algorithm demonstrates the effectiveness of including architectural considerations in the design of an algorithm; in particular, it shows the effectiveness of the no-wait principle in the design of non-deterministic search algorithms.

As indicated in Section 1.3, the problem of deriving dominance relations is an important, open research problem. We do not claim to have solved this problem. Nonetheless, we believe that the work in this thesis has taken several important steps towards an eventual solution. Perhaps, the most important step is the demonstration that machine learning of dominance relations is possible. We have also laid the ground work for future efforts with our work on the fundamental issues in this problem: a problem representation which exposes relevant semantic information, a classification of dominance relations which identifies the useful classes of dominance relations, and a characterization of the applicable learning mechanisms.

The work on learning dominance relations by experimentation also contributes to the field of learning by experimentation, in particular, learning problem solving heuristics by experimentation. Neither number optimization problems nor pruning heuristics between two arbitrary nodes on the same level have been previously studied in experimentation research. Specific contributions in this area include the mechanisms (knowledge) needed to drive such a learning system, and the work on automatic derivation of new semantically sensible symbols.

CHAPTER 2
B&B ALGORITHMS IN A TWO-LEVEL MEMORY SYSTEM

2.1 Introduction

The branch and bound (B&B) algorithm [Law66, Mit70] has emerged as the principal method for solving constrained optimization problems for which 'direct' methods of solution either do not exist or are inefficient. Formally, a constrained optimization problem is a problem having the form:

$$\begin{array}{lll} \text{Minimize} & C_0(x) & x \in X \\ \text{subject to} & g_i(x) \leq 0 & i = 1, 2, \ldots, m \end{array} \qquad (2.1)$$

X represents the domain of optimization defined by the m constraints, normally an Euclidean n-space, and x denotes a vector $(x_1, x_2, \ldots, x_n)$. A *feasible solution* is a solution vector that lies in X. An *optimal solution* is a feasible solution for which $C_0(x)$ is minimal. Among the constrained optimization problems to which the B&B algorithm has been applied are scheduling problems [Len76], puzzles and other cognitive tasks [Nil71], optimal decision trees [MaM78], knapsack problems [GrH70, HoS74, InK77], traveling salesman problems [BeN68, BeM71, Gar73], flow shop and job shop sequencing [IgS65], facility allocation [ElR66], and integer programming [GaN72, GeM72]. It has also been suggested by Kumar and Kanal [KuK83] that the B&B algorithm is a generalization of many heuristic search algorithms including A* [Nil71], AO* [Nil71], SSS* [StK83], B* [Ber79], alpha-beta [KnM75] and dynamic programming [BeD62, DrL77, KaH67, Iba73, Whi69].

The search process generated by the B&B algorithm is often represented by a search tree. Nodes in this tree represent subinstances. We will use the terms subinstance and node interchangeably in the rest of this chapter. Branches in the search tree represent the decomposition of subinstance P_i into additional subinstances.

The B&B algorithm has four major components. The *branching rule* determines if and how a given node is to be decomposed into additional nodes. The branching rule creates multiple nodes by heuristically selecting some

unassigned parameter(s) in the node and assigning alternative values to the selected parameter(s). For example, the unassigned parameters in the traveling-salesman problem are the set of untraversed edges. The B&B process expands a node by selecting an untraversed edge (i,j) and creating two alternatives: (a) the edge is traversed and the salesman goes directly from City i to City j, and (b) the edge is not traversed and the salesman stays at City i.

The second component of the B&B algorithm is a *lower-bound function*. Let $f(P_i)$ be the value of the best solution that can be derived from node, P_i. The lower-bound function, g, must satisfy the following conditions:

(a) $g(P_i)$ never exceeds $f(P_i)$;

(b) $g(P_i)$ has the same value as $f(P_i)$ whenever P_i is feasible solution;

(c) every child of P_i has a lower-bound value that exceeds $g(P_i)$.

The lower-bound function is problem-dependent. A linear program with relaxed integer constraints is often used in the integer-programming problem [LaD60]. An assignment algorithm or a spanning-tree algorithm can be used for the traveling-salesman problem [Eas58].

The third component of the B&B algorithm, the *elimination rule*, uses the lower-bound function to eliminate those nodes which are known not to contain an optimal solution to the problem instance. The *incumbent* z is the best solution value found so far in the B&B process. In minimization problems, a node P_i is pruned if its lower-bound exceeds the incumbent as it cannot lead to a solution value better than the incumbent. In other words, P_i is eliminated if:

$$g(P_i) \geq z \qquad (2.2)$$

The B&B process continues until all nodes have been either expanded or eliminated.

A relaxed form of this elimination rule will find a suboptimal solution with a guaranteed accuracy [LaW66]. Suppose that a deviation of 10% from the optimum is tolerable. A feasible solution of 150 permits the elimination of all nodes with lower-bounds not less than 136.4 (or $150/(1+0.1)$) as these nodes cannot lead to solutions which are better than 150 by more than 10%. This technique significantly reduces the amount of intermediate storage and time required by the B&B process. We can define the *allowance function* $\epsilon(z)$: $\mathbf{R} \rightarrow \mathbf{R}$ (set of reals) such that P_i is terminated if:

$$g(P_i) \geq z - \epsilon(z). \qquad (2.3)$$

The final incumbent z_F will be related to the optimal solution value z_O by the inequalities [Iba76a]:

$$z_F - \epsilon(z_F) \leq z_O \leq z_F \tag{2.4}$$

Two common allowance functions are:

$$\epsilon(z) = \epsilon \qquad \epsilon \geq 0 \text{ (absolute error deviation) and} \tag{2.5}$$

$$\epsilon(z) = \frac{\epsilon z}{1+\epsilon} \qquad \epsilon \geq 0,\ z \geq 0 \text{ (relative error deviation)} \tag{2.6}$$

The final component of the B&B algorithm is the *selection rule* that chooses one of the currently active nodes for expansion. The B&B process maintains the set of active nodes in an ordered list and always selects the first element of this list. A *breadth-first B&B algorithm* maintains the active list in a first-in/first-out order; a *depth-first B&B algorithm* maintains the active list in a last-in/first-out order; a *best-first B&B algorithm* maintains the active list in increasing order of lower-bounds.

The B&B algorithm for minimization problems can be summarized as follows:

```
incumbent z = ∞;
active list = {P0};
while active list ≠ Ø do [
    apply selection rule to active list;
    expand selected node by the branching rule;
    for P ∈ {children generated} do [
        if P is a feasible solution then [
            update incumbent z;
            apply elimination rule ]
        else if (g(P) < (z−ε(z))) then
            insert P into active list
    ]
]
```

We will illustrate the B&B process with the integer-programming problem [Mit70]. The integer-programming problem is the constrained optimization:

$$\begin{aligned}
&\text{Minimize} && CX \\
&\text{subject to} && AX \geq B \\
& && X^T = (x_1, x_2, \cdots, x_n); \\
& && x_i\text{: non-negative integer, } i=1,2,\ldots,n; \\
& && A \text{ is a 1-by-n constant matrix;} \\
& && B \text{ is a m-by-1 constant matrix;} \\
& && C \text{ is a n-by-m constant matrix;}
\end{aligned} \tag{2.7}$$

These problems differ from ordinary linear-programming problems in that the variables can only have non-negative integer values.

One formulation of the B&B algorithms for the integer-programming problem uses the dual simplex method as the lower-bound function. The dual simplex method ignores the integrality constraints on the values of the x_i's and solves the problem instance as a linear program. If the simplex solution is integral, then a feasible solution has been generated; otherwise the B&B process applies the branching rule. The branching rule selects a variable with a non-integral value (say $x_i = 4.4$) and uses this variable to create two new nodes. One node restricts the value of x_i to not more than the floor of its value in the simplex solution ($x_i \leq \lfloor 4.4 \rfloor$ or $x_i \leq 4$). The other node restricts the of x_i to not less than the ceiling of its value in the simplex solution ($x_i \geq \lceil 4.4 \rceil$ or $x_i \geq 5$). The branching rule always chooses the variable with the greatest (up or down) penalty. The up penalty for a variable x_i with value of a_i is the estimated increase in the simplex solution value due to the introduction of the integral constraint $x_i \geq \lceil a_i \rceil$. The down penalty is similarly associated with the constraint $x_i \leq \lfloor a_i \rfloor$. The lower-bound of each new node is the sum of the value of the simplex solution of its parent and the associated penalty.

Figure 2.1(b) shows the best-first B&B tree for the problem in Figure 2.1(a). Initially, there is only the problem instance so the best-first B&B process selects this problem instance for expansion. Application of the dual simplex method yields a solution of 14.2. This is not a feasible solution as the variables have non-integral values and the B&B process calculates the up and down penalties. x_1 has the greatest penalty (U = 1.8). This results in the creation of two new nodes, one with $x_1=0$, and the other with $x_1 \geq 1$.

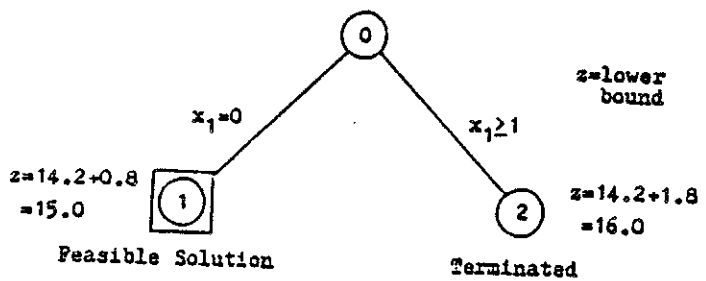
The best-first B&B process always selects the active node with the smallest lower-bound; N_2 has a smaller lower-bound than N_3. Application of the dual simplex method to node N_2 yields a simplex solution in which all variables have integral values. This feasible solution has a value of 15. N_3 has a lower-

$$\min x_0 = 7x_1 + 3x_2 + 4x_3$$
$$x_1 + 2x_2 + 3x_3 \geq 8$$
$$3x_1 + x_2 + x_3 \geq 5$$
$$x_1, x_2, x_3 \geq 0, \text{ integer}$$

(a)

Optimal dual simplex solution
$x_0 = 14.2$
$x_1 = 0.4$
$x_2 = 3.8$
$x_3 = 0$

Variable	Down Penalty	Up Penalty
x_1	0.8	1.8
x_2	0.3	0.13



Optimal dual simplex solution
$x_0 = 15.0$
$x_1 = 0$
$x_2 = 5$
$x_3 = 0$

(b)

Figure 2.1 (a) A 0/1-knapsack problem instance and (b) its best-first B&B tree.

bound of 15.0 and the elimination rule eliminates N_3. There are no more active nodes and the best-first B&B process terminates with an optimal solution of 15.

The best-first B&B algorithm expands the smallest number of nodes if no node other than the optimal solution node has a lower-bound value equal to the value of the optimal solution [LaW66, LiW84]. Unfortunately, the best-first B&B algorithm has a maximum memory space requirement whose expectation is is an exponential with a sublinear exponent [WaY82, WaY85]. In contrast, the number of nodes in last-in/first-out stack of the depth-first B&B algorithm never exceeds the height of the search tree. For many search problems, the height of the search tree is equal to the number of variables in the problem and is reasonably small. The space requirement of the breadth-first B&B algorithm is similar to that of the depth-first B&B algorithm. Thus, for any non-trivial problem instance, the best-first B&B algorithm will require much more memory than the other two B&B algorithms.

Computer memories of very large overall capacity are often organized into a hierarchy of at least two levels, main memory and secondary memory. The smaller main memory usually consists of faster but more expensive semiconductor memory, while the larger secondary memory usually consists of slower disk memory. A program can only reference its information (i.e. instruction code and data) when it resides in main memory; information having immediate likelihood of being referenced should be kept in main memory. This *storage allocation problem*, i.e. determining how information is to be distributed among the levels of memory at each moment of time, must be solved before a memory hierarchy can be used effectively.

An elegant and universally accepted solution to the storage allocation problem is the idea of virtual memory [Den70]. At the heart of this idea is the notion that the memory address used by a program to reference information is a concept distinct from the physical memory location in which the information item is held. The computer hardware and software directs program-generated addresses to the memory locations containing the information addressed and moves information between main memory and secondary memory. The term *virtual address* denotes an address used by the programmer, while the term *memory address* denotes an address used by the memory. There is no a priori correspondence between virtual addresses and memory addresses, so an *address map* associates virtual addresses with memory addresses.

When presented with a virtual address, the mapping device generates the corresponding memory address if the item is present in main memory and a *missing-item fault* otherwise. The fault interrupts the program until the

missing item is retrieved from secondary memory and placed in main memory. The mapping device then redefines the address map and completes the reference. Whenever main memory is full, some item has to be removed to make way for the item entering, the particular item being chosen at the discretion of the *replacement rule*. There is also a *fetch rule* which decides when to load an item. A *demand rule* loads an item into main memory only when a fault for it occurs. *Prefetching* is said to occur if action is taken to load an item before it is referenced.

An address map of reasonable size requires the grouping of information into sets of contiguous addresses called blocks. The entries in the mapping table will now refer to blocks, which are far less numerous than individual addresses. One method-*segmentation*-organizes address spaces into blocks (segments) of arbitrary size. Another-*paging*-organizes memory space into blocks (pages) of fixed size. Paging is the more popular of the two methods. The equal sized blocks of main memory are known as *page frames*. Page frames serve as sites of residence for *pages* which are blocks of virtual addresses of the same size.

Virtual memory gives the programmer the illusion of a very large main memory even though the computer has a relatively small main memory. However, virtual memory is not without problems. Space in a virtual memory may be an illusion; unnecessarily large and carelessly organized programs may generate excessive overhead in the automatic storage allocation mechanism. Virtual memory management mechanisms perform most efficiently when programs tend to exhibit good *locality*, i.e. they tend to concentrate their references in small regions of address space. Programs with poor locality will continually be interrupted while missing items are retrieved from secondary memory.

Studies have been carried out on non-search algorithms in a paged virtual-memory environments. These include rules for enhancing locality [Mor66], automatic restructuring of programs [HaG71, Fer74, Fer75, Fer76, BaS76], matrix manipulation [McC69, FiP79], sorting [BGM70, AEH84], and database operations [Smi78].

In this chapter, we investigate B&B algorithms in a *single-user computer system* with a two-level memory. The time-efficiency of an algorithm in single-user systems may be characterized by its expected completion time.

We will start by establishing the need to revise the belief that the best-first B&B algorithm is always the best B&B algorithm. Previous theoretical studies of serial B&B algorithms [IbK76a, IbK76b, IbH77, IbK78, KoS74, and

Nil80] have concentrated on the complexity of the algorithm, either its worst-case behavior or its average complexity. These complexity functions are not precise enough to guide a consideration of the trade-offs between the different B&B algorithms. The effects of the accuracy of the lower-bound function on the average performance of the A*-algorithm (best-first search) and informed backtracking (depth-first search) have been studied for decision problems [Pea84]. Other models of the cutoff mechanism which do not explicitly consider the accuracy of the lower-bounds have also been used to study the average performance of best-first B&B algorithms [Smi84] and depth-first B&B algorithms [Smi84, Sto84]. Section 2.2 describes an analysis of the numbers of nodes expanded by best-first and depth-first B&B algorithms using the accuracy of the lower-bound function and the distribution of the solution values. This will show that the possible differences between the number of node expanded by the best-first B&B and depth-first B&B algorithms span a wide range of values, including values where the best-first B&B algorithm may not be the best B&B algorithm.

Section 2.3 investigates the relationship between the number of nodes expanded and the effectiveness of the best-first B&B and depth-first B&B algorithms. We then consider alternatives that are effective in situations when neither the best-first B&B algorithm nor the depth-first B&B algorithm is effective. Section 2.4 describes a virtual memory management system tailored to the characteristics of the best-first B&B algorithm. Section 2.5 considers the modified B&B algorithm that is a compromise between the depth-first and best-first B&B algorithms.

Both of these alternatives are of limited value in multi-user systems. A specialized virtual-memory management system adversely affects other users who are not running B&B algorithms. Moreover, there is a lack of a suitable metric for the time efficiency that can be used as an optimization measure. The completion time of a program in a multi-user system depends on both the algorithm and the system load. Other possible metrics such as the processing time and the sum of the processing and secondary-storage times do not adequately describe either the trade-off or the overlap between computations and secondary-memory accesses.

2.2 A Comparison of the Best-First and Depth-First B&B Algorithms

Any analytical model of the depth-first B&B algorithm capable of predicting the number of node expansions must consider the accuracy of the lower-bound function. To compare the performances of the B&B algorithms for all possible distributions of the accuracy is a non-trivial, if not impossible, problem. This section presents a mathematically tractable solution that assumes a constant distribution for the accuracy.

We assume a state-space which is a complete s-way tree of height h (Figure 2.2). The root of the tree is at Level 0, and the leaves (solution nodes) are at Level h. Let $\mu(i) = s^i$ be the number of nodes in Level i, and let $P_{i,j}$, $0 \le i \le h$, $0 \le j \le \mu(i)-1$, be the j'th node in the i'th level of the tree. In addition, let S_j be the value of solution node $P_{h,j}$, $0 \le j \le \mu(h)-1$. We assume that the solution values are independently and identically distributed (iid) random variables with distribution function $F_S(s)$ and density function $f_S(s)$, where

$$f_S(x) = \begin{cases} \dfrac{\beta\, e^{\beta x}}{(e^{\beta b} - e^{\beta a})} & a \le x \le b \\ 0 & \text{otherwise} \end{cases} \qquad (2.8)$$

This is a truncated exponential distribution between a and b and it represents many interesting situations as β is varied. When $\beta = -\infty$, it is a delta function at a and represents the situation in which all solutions are optimal. As β is increased, the standard deviation of the distribution first increases and then decreases. At $\beta = \infty$, the function is a delta function at b. A large standard deviation may be interpreted as a low probability of finding solution values close to the optimal solution value.

The subtree rooted at $P_{i,j}$ contains $\mu(h-i)$ solution nodes, so its best solution value, $z_{i,j}$, is

$$z_{i,j} = \min\left\{S_{j\cdot\mu(h-i)}, \ldots, S_{(j+1)\cdot\mu(h-i)-1}\right\} \qquad (2.9)$$

The optimal solution, z_0 will be given by $z_{0,0}$. $z_{i,0}, \ldots, z_{i,\mu(i)-1}$ are iid random variables because the S_j's are iid random variables and their distribution is

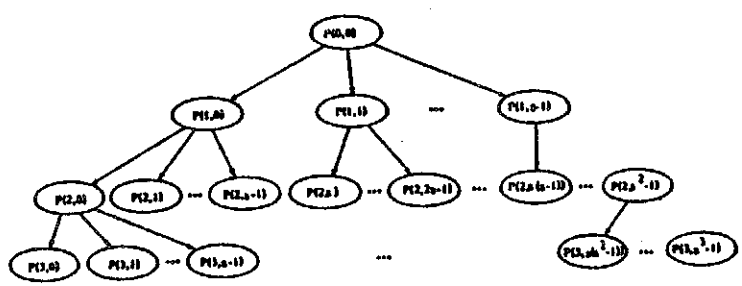


Figure 2.2 A s-way state-space tree.

$$F_{z,i}(x) = \begin{cases} 0 & x < a \\ 1-[1-F_S(x)]^{\mu(h-i)} = 1-\left[\dfrac{e^{\beta b}-e^{\beta x}}{e^{\beta b}-e^{\beta a}}\right]^{\mu(h-i)} & a \le x \le b \\ 1 & x > b \end{cases} \quad (2.10)$$

Let $g_{i,j}$ be the lower-bound of node $P_{i,j}$, and let $\delta_{i,j}$ be the accuracy of the lower-bound function at node $P_{i,j}$. $g_{i,j} = \delta_{i,j} \cdot z_{i,j}$. By definition, $\delta_{h,j} = 1$ as $P_{h,j}$ is a solution node. We have only studied the case of $\delta_{i,j} = \delta$ in this thesis, where $i < h$ and δ is a constant in the range $[a/b, 1]$. The elimination rule is ineffective for $\delta < a/b$, and the entire state-space will have be searched. $g_{i,0}$, ..., $g_{i,\mu(i)-1}$ are iid random variables as $z_{i,0}, \ldots, z_{i,\mu(i)-1}$ are iid random variables, and their distribution function for $i < h$ is

$$F_{g,i}(x) = \begin{cases} 0 & x < \delta a \\ F_{z,i}(x/\delta) & \delta a \le x \le \delta b \\ 1 & \delta b < x \end{cases} \quad (2.11)$$

We will use $N_B(i,j)$ to denote the number of nodes expanded by the best-first B&B algorithm in the subtree rooted at $P_{i,j}$. $P_{i,j}$ will be expanded if and only if $g_{i,j} \le z_O$, so $N_B(i,j)$, $0 \le j \le \mu(i)-1$, can be expressed as a following recurrence equation.

$$N_B(i,j) = \begin{cases} 1 + \sum_{j_1=0}^{r-1} \left[\Pr(g_{i+1,sj+j_1} \le z_O) \cdot N_B(i+1, sj+j_1) \right] & i < h-1 \\ 1 & i = h-1 \end{cases} \quad (2.12)$$

$\Pr\{g_{i,j} \le z_O\}$ and $N_B(i,j)$ are independent. This permits $E[N_B(i,j)]$, the expected value of $N_B(i,j)$, to be expressed as

$$E[N_B(i,j)] = 1 + \sum_{j_1=0}^{r-1} \left\{ E[\Pr\{g_{i+1,sj+j_1} \le z_O\}] \cdot E[N_B(i+1, sj+j_1)] \right\} \quad i < h-1 \quad (2.13)$$

$E[N_B(0,0)]$ is the expected number of nodes expanded by the best-first B&B algorithm as the root is always expanded. $g_{i,j}$, $j = 0, \ldots, \mu(i)-1$, are iid random variables so $E[N_B(i,0)] = \cdots = E[N_B(i,\mu(i)-1)]$, $0 \le i \le h$, and

$$E[N_B(i,0)] = 1 + s \cdot E[\Pr\{g_{i+1,0} \le z_O\}] \cdot E[N_B(i+1,0)] \quad i < h-1 \quad (2.14)$$

where

$$E[\Pr\{g_{i,j} \le z_O\}] = \int_a^b F_{g,i}(x) f_{z_O}(x)\,dx$$

$$= \int_a^{\delta b} F_{z,i}(x/\delta) f_{z_O}(x)\,dx + \int_{\delta b}^{b} f_{z_O}(x)\,dx \quad (2.15)$$

From Eq. (2.9), z_O will have the following density function.

$$f_{z_O}(x) = \begin{cases} \mu(h) f_S(x)[1-F_S(x)]^{\mu(h)-1} & a \le x \le b \\ 0 & \text{otherwise} \end{cases}$$

$$= \begin{cases} \dfrac{\mu(h)\beta e^{\beta x}(e^{\beta b}-e^{\beta x})^{\mu(h)-1}}{(e^{\beta b}-e^{\beta a})^{\mu(h)}} & a \le x \le b \\ 0 & \text{otherwise} \end{cases} \quad (2.16)$$

Substituting Eq's (2.10) and (2.16) into Eq. (2.15) yields

$$E[\Pr\{g_{i,j} \le z_O\}] = 1 - \int_a^{\delta b} \frac{\mu(h)\beta e^{\beta x}(e^{\beta b}-e^{\beta x/\delta})^{\mu(h-i)}(e^{\beta b}-e^{\beta x})^{\mu(h)-1}}{(e^{\beta b}-e^{\beta a})^{\mu(h-i)+\mu(h)}}\,dx \quad (2.17)$$

Similarly, let $N_D(i,j)$ be the number of nodes expanded in the subtree rooted at $P_{i,j}$ by the depth-first B&B algorithm, given that node $P_{i,j}$ is expanded and $t_{i,j}$ is the value of the incumbent when $P_{i,j}$ is the first element in the active list. The depth-first B&B algorithm expands $P_{i,j}$ if and only if $g_{i,j} \le t_{i,j}$. Thus, $N_D(i,j)$ can be described by the recurrence equation:

$$N_D(i,j) = \begin{cases} 1 + \sum_{j_1=0}^{r-1} \left[\Pr(g_{i+1,sj+j_1} \le t_{i+1,sj+j_1}) \cdot N_D(i+1, sj+j_1) \right] & i < h-1 \\ 1 & i = h-1 \end{cases} \quad (2.18)$$

Taking the expectations of Eq. (2.18) yields

$$E[N_D(i,j)] = 1 + \sum_{j_1=0}^{r-1} \left[E[\Pr\{g_{i+1,sj+j_1} \le t_{i+1,sj+j_1}\}] \cdot \right.$$

$$\left. E[N_D(i+1, sj+j_1)] \right] \quad i < h-1 \quad (2.19)$$

where

$$E[Pr\{g_{i,j} \le t_{i,j}\}] = \int_a^b F_{g_{i,j}}(x) f_{t_{i,j}}(x)\,dx$$

$$= \int_a^{\delta b} F_{s_{i,j}}(x/\delta) f_{t_{i,j}}(x)\,dx + \int_{\delta b}^b f_{t_{i,j}}(x)\,dx \qquad (2.20)$$

The expected number of nodes expanded by a depth-first B&B will be $E\{D(0,0)\}$. $j \cdot \mu(h-i)$ solution nodes have already been found when $P_{i,j}$ becomes the first element in the active list. Thus,

$$t_{i,j} = \min\left\{S_0, \ldots, S_{j\cdot\mu(h-i)-1}\right\} \qquad (2.21)$$

and $t_{i,j}$ has the following density function.

$$f_{t_{i,j}}(x) = \begin{cases} j\mu(h-i) f_S(x)[1-F_S(x)]^{j\mu(h-i)-1} & a \le x \le b \\ 0 & \text{otherwise} \end{cases}$$

$$= \begin{cases} \dfrac{j\mu(h-i)\beta e^{\beta x}(e^{\beta b} - e^{\beta x})^{j\mu(h-i)-1}}{(e^{\beta b} - e^{\beta a})^{j\mu(h-i)}} & a \le x \le b \\ 0 & \text{otherwise} \end{cases} \qquad (2.22)$$

Substituting Eq's (2.10) and (2.22) into Eq. (2.20) yields

$$E[Pr\{g_{i,j} \le t_{i,j}\}] = 1 - \int_a^{\delta b} \frac{j\mu(h-i)\beta e^{\beta x}(e^{\beta b} - e^{\beta x/\delta})^{\mu(h-i)}(e^{\beta b} - e^{\beta x})^{j\mu(h-i)-1}}{(e^{\beta b} - e^{\beta a})^{(j+1)\mu(h-i)}}\,dx \qquad (2.23)$$

Eq's (2.17) and (2.23) have no closed form solutions but can be transformed into recurrences. However, O(hs) recursions are required, and this number is very large for non-trivial h and s. Instead, Eq's (2.17) and (2.23) were evaluated by numerical integration.

Figure 2.3 shows the results of evaluating $E[N_B(0,0)]$ for a tree with s=2, h=15, and various values of δ and β (a=10, b=20). Figure 2.4 shows the difference between $E[N_D(0,0)]$ and $E[N_B(0,0)]$ for the same tree. The best-first B&B algorithm always expands less nodes than the depth-first B&B algorithm. The difference between $E[N_D(0,0)]$ and $E[N_B(0,0)]$ first increases and then decreases as δ increases from 0.55. A small δ corresponds to a very inaccurate lower-bound function. Such a lower-bound function renders and the elimination rule impotent, so the best-first B&B process has to expand most of the problem space. This will be very similar to the portion of the problem space explored by the depth-first B&B process. A δ close to 1 corresponds to a

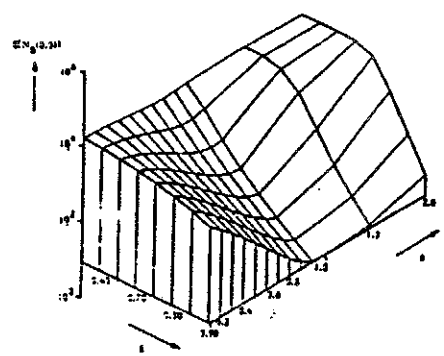


Figure 2.3 $E[N_B(0,0)]$ for a state-space tree with k=2, h=15, a=10, b=20.

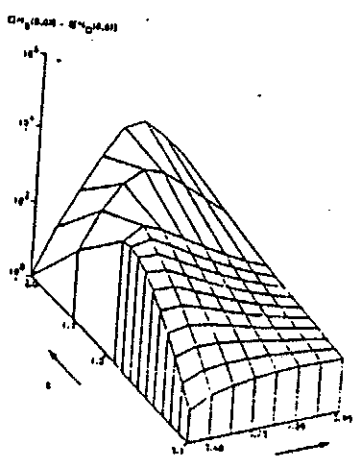


Figure 2.4 Difference between $E[N_D(0,0)]$ and $E[N_B(0,0)]$ for a state-space tree with k=2, h=15, a=10, b=20.

very accurate lower-bound function which is able to eliminate most nodes in the search space. For any particular problem, both the best-first and depth-first B&B algorithms use the same elimination rule. Hence, for a problem with a large δ, the number of nodes expanded by the best-first B&B algorithm is close to the number of nodes expanded by the depth-first B&B algorithm. Between these two extremes, there are substantial disparities between the number of nodes expanded by the depth-first and best-first B&B algorithms. The regions for which the above behavior takes place depend on the value of β.

The above behavior can be illustrated by the integer-programming, vertex-cover, and 0/1-knapsack problems.

The vertex-cover problem involves a graph G=(V,E), where V is the set of vertices and E is the set of edges. A vertex-cover for the graph G is a subset $V' \subseteq V$ such that for each edge $\{u,v\} \in E$, at least one of u and v belongs to V'. The vertex-cover problem seeks the vertex-cover with the smallest number of vertices.

The 0/1-knapsack problem involves a set U of n items. The knapsack has a capacity K. Each item $u \in U$ has a weight w(u) and a value r(u). The 0/1-knapsack problem seeks the subset $U' \subseteq U$ which maximizes $\sum_{u \in U'} v(u)$ and satisfies $\sum_{u \in U'} w(u) \leq K$.

The lower-bound functions for the 0/1 knapsack and vertex-cover problems are very accurate greedy algorithms. The lower-bound function for the integer-programming problem is a linear program with the integrality constraints removed for all unassigned variables and is less accurate than the lower-bound functions of the vertex-cover and 0/1-knapsack problems. Table 2.1 shows some results from our simulations with these problems. For the knapsack and vertex-cover problems, the number of nodes expanded by the depth-first B&B algorithm was almost identical to that expanded by the best-first B&B algorithm. For integer-programming problems, the ratio of the number of nodes expanded in a depth-first B&B algorithm to that in a best-first B&B algorithm varies between 1.4 and 17.6.

2.3 B&B Algorithms in a Paged Virtual-Memory Environment

In this section, we examine the overheads of the best-first B&B and depth-first B&B algorithms in a paged virtual memory environment. One implementation of the best-first B&B algorithm uses an unordered list of nodes and a priority queue* of pointers to nodes. In each iteration, the best-first B&B

* The priority queue can be implemented by a heap, which is a complete binary tree such that the value of each node is at least as small as that of its descendants.

Table 2.1 Number of nodes expanded by the best-first (BFS) and depth-first (DFS) B&B algorithms for randomly generated 30-object 0/1-knapsack, 80-node vertex-cover, and 20-by-20 integer programming problems.

Problem	Number of Nodes Expanded					
	0/1 Knapsack		Vertex-Cover		Integer-Program	
	BFS	DFS	BFS	DFS	BFS	DFS
1	1701	1731	4351	6205	1541	2169
2	5475	5519	5839	7290	2096	9425
3	6394	6442	8045	9759	2237	5356
4	1406	1474	5349	7134	1242	9752
5	1103	1145	4401	5186	594	1178
6	15309	15403	8253	8277	1396	3051
7	808	860	12140	12463	355	2687
8	7137	7202	7663	7647	1456	14485
9	18265	18396	12847	21892	246	3852
10	14257	14337	14104	15524	3475	7391

process gets a pointer to the node with the smallest lower-bound from the priority queue and deletes this node from the node list. After expanding the selected node, it inserts the newly-generated nodes into the node list and updates the priority queue by inserting pointers to the new nodes. A pointer node is much smaller in size than a node, so the pointer heap can be assumed to be resident in main memory and incurs no paging overhead.

Each page holds up to n_p nodes and the portion of the node list resident in main memory cannot occupy more than n_m pages. Suppose that the best-first B&B process expands n_{bf_s} nodes before it finds the optimal solution for a given problem instance and that the depth-first B&B process expands n_{df_s} nodes for the same problem instance. Let Ξ be the ratio of the number of nodes expanded by the depth-first B&B algorithm to that expanded by the best-first B&B algorithm ($=n_{df_s}/n_{bf_s}$). Assume that each node expansion generates s nodes. Let r_{bf_s} and w_{bf_s} be the expected numbers of page reads and page writes incurred by the best-first B&B algorithm.

For most problems, the best-first B&B algorithm generates few feasible solutions before finding the optimal solution. These few invocations of the elimination rule incurs little paging overhead. Thus, we can assume that the overhead due to the elimination rule is small compared to the overhead due to the expansion of nodes and may be ignored.

Let n_t be the number of nodes that can be expanded without exceeding the size of the main memory. The initial free memory space is $(n_m n_p - 1)$ nodes as only the root of the B&B tree is present in main memory. Each node expansion removes one node from main memory but generates s new nodes; thus, each node expansion increases the number of nodes in main memory by s-1.

$$n_t = \left\lfloor \frac{n_m n_p - 1}{s - 1} \right\rfloor \tag{2.24}$$

The node with the smallest lower-bound is equally likely to be in any of the pages occupied by the node list as the node list is not ordered. Consider the i-th node expansion. The $(i(s-1)+1)$ active nodes occupy $\left\lceil (i(s-1)+1)/n_p \right\rceil$ pages. n_m of these pages are resident in main memory; hence, the retrieval of the node with the smallest lower bound causes a page fault with probability $\left(1 - \lfloor n_m / \left\lceil (i(s-1)+1)/n_p \right\rceil \rfloor\right)$. Hence, the expected number of page reads in a best-first B&B process is given by:

$$r_{bfs} = \sum_{i=n_t+1}^{n_{bfs}} \left(1 - \frac{n_m}{\left\lceil \frac{i(s-1)+1}{n_p} \right\rceil} \right) \qquad n_{bfs} > n_t \tag{2.25}$$

Each node expansion generates s new nodes. One new node can be inserted into the space vacated by the expanded node. Thus, the expected number of page writes is

$$w_{bfs} = \left\lceil \frac{n_{bfs}(s-1)+1}{n_p} \right\rceil - n_m \qquad n_{bfs} > n_t \tag{2.26}$$

If there are sufficient secondary memory buffers, the best-first B&B process can resume as soon as the contents of the page selected for replacement has been copied into a secondary memory buffer. The actual page write is overlapped with processing. In contrast, the best-first B&B process has to be suspended until the page read has been completed.

Let h_e be the CPU time required to expand a node. and let h_d be the average page-read time. The overhead incurred by a best-first search, D_{bfs}, can be approximated by the summation of the total page-read times and the maximum of the CPU times and the page write times.

$$D_{bfs} = r_{bfs}h_d + \max(w_{bfs}h_d, n_{bfs}h_e) \tag{2.27}$$

From Eq. (2.25), $r_{bfs} \simeq w_{bfs} \simeq n_{bfs}$ when $n_{bfs} >> n_t$. In this case, the best-first B&B algorithm will be more efficient than the depth-first B&B algorithm if Ξ exceeds Ξ_e, where

$$\Xi_e = \frac{\max(h_e, h_d/n_p) + h_d/n_p}{h_e} \tag{2.28}$$

Similarly, the depth-first B&B algorithm will be more time efficient than the best-first B&B algorithm when Ξ is less than Ξ_e. The vertex-cover and 0/1-knapsack problems are two problems for which this condition is likely to be true. Thus, it is possible for the best-first B&B algorithm to be less efficient than the depth-first B&B algorithm in a two-level memory system.

The minimum time required by any B&B algorithm to solve a problem instance is $h_e \cdot n_{bfs}$. The depth-first B&B algorithm approaches this optimal performance when $n_{dfs} \simeq n_{bfs}$. The best-first B&B algorithm should be chosen only when $\Xi >> \Xi_e$. Similarly, a depth-first search should be used when $\Xi << \Xi_e$. For problems characterized by intermediate values of Ξ, the choice between the depth-first and best-first searches is less clear. Either a depth-first

or a best-first search may be used, although neither is particularly suitable. In the next two sections, we study alternatives which may be used for problems with intermediate values of Ξ.

2.4 A Virtual Memory System for the Best-First B&B Algorithm

In work prior to the work of this thesis, we have designed a virtual memory system which is tailored to the characteristics of the best-first B&B algorithm [YuW83, Yu83]. Figure 2.5 depicts the virtual-memory system. Nodes in secondary memory are organized as a B^+-tree.*** Each leaf in a B^+-tree is a page, and the nodes reside in the leaves of the B^+-tree. Non-terminal nodes in the B^+-tree are pointer nodes that are much smaller than the leaf nodes, so the memory management system keeps most of the pointer tree in main memory. This significantly reduces the number of secondary-memory accesses required to access a node in secondary storage.

The main memory contains a partial list of nodes (*m-list*) and a heap of pointers to nodes in the m-list. The best-first B&B process inserts newly generated nodes into the m-list. Whenever the m-list is full, the memory management system moves part of the m-list to pages in the B^+-tree. The actual transfer of nodes will be overlapped with computation if there are sufficient secondary-memory buffers.

The best-first B&B process always expands the active node with the smallest lower-bound. This node may either be in the *m*-list or the first page of the B^+-tree. The overhead of accessing the active node with the smallest lower-bound will be reduced by keeping part or all of the first B^+-tree page in main memory (*s-buffer*).

Inserting a node into a normal B^+-tree involves reading the appropriate page into main memory, inserting the node into the page image in main memory, and writing the page image back to disk. A more efficient scheme sets the block size to an integral number of disk sectors not smaller than the node size and uses a bit map in main memory to show the status of all page blocks. Inserting a node into a page, therefore, consists of searching for an

*** A B-tree of order b is a search tree that is either empty or satisfies the following properties [Com79]: (i) the root node has at least two children; (ii) each node contains at least b keys and b+1 pointers; and (iii) each node contains at most 2b keys and 2b+1 pointers. A B^+-tree is a variant of the B-tree in which all records reside in the leaves. The upper levels are organized as a B-tree and serve as an index to locate a record. The leaf nodes are linked from left to right for easy sequential processing.

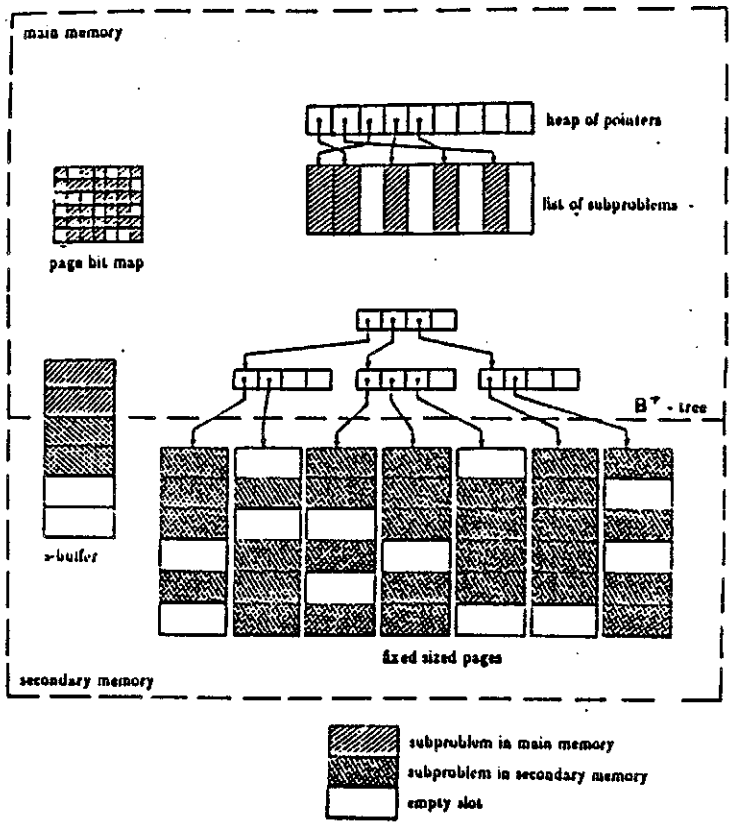


Figure 2.5 A virtual-memory system for the best-first B&B algorithm.

empty block in the bit map and writing the block when the disk head is properly positioned.

A suitable replacement algorithm for this virtual-memory system is the *Back-Replacement policy*, $S(\phi)$, which replaces the ϕ nodes with the ϕ-largest lower bounds. A value of ϕ between $0.7\,n_m$ and $0.9\,n_m$ has been found to be suitable for integer-programming and vertex-cover problems. The page size should be between $0.65\,n_m$ and $0.85\,n_m$.

2.5 The Modified B&B Algorithm

The modified B&B algorithm partitions the range of possible lower-bounds into n disjoint regions, $\{[u_0,u_1), [u_1,u_2), \ldots, [u_{n-1},u_n)\}$ (Figure 2.6). Normally $u_0 = g_0$ and $u_n = g^e_{opt}$, where g_0 is the lower-bound of the initial problem and g^e_{opt} is an estimate of the optimal solution value. The set of active nodes in each region is usually large, so some of the nodes may have to be stored in secondary memory. Nodes in each region $[u_i,u_{i+1})$, $i = 0,\ldots,n-1$, are not ordered by lower-bounds but are kept in a LIFO stack S_i because of the high overhead of accessing nodes in secondary storage.

The implementation of the modified B&B algorithm depends on whether it uses the file system or the virtual-memory system. We will describe the version for a file system and note the needed changes for a virtual-memory system.

2.5.1 Modified B&B Algorithm for a File System

Each stack S_i consists of a secondary-memory file, F_i, and a main-memory buffer, B_i. Buffer B_i is the top portion of S_i. The main-memory space allocated to the storage of nodes is organized as a pool of fixed-sized frames. Each buffer consists of a number of frames. Nodes falling within the range of S_i are inserted into B_i. If the frames allocated to B_i are all full, the modified B&B process will request a new empty frame from the frame pool manager. The frame manager will always try to satisfy this request from its frame pool but if the frame pool is empty, it has to reclaim the oldest full frame from the buffers. Assuming sufficient secondary-memory buffers, the overhead of reclaiming a buffer is the time to copy the contents of the reclaimed frame into a secondary memory buffer. The actual transfer of nodes from the secondary-memory buffer to secondary memory is completely overlapped with computation.

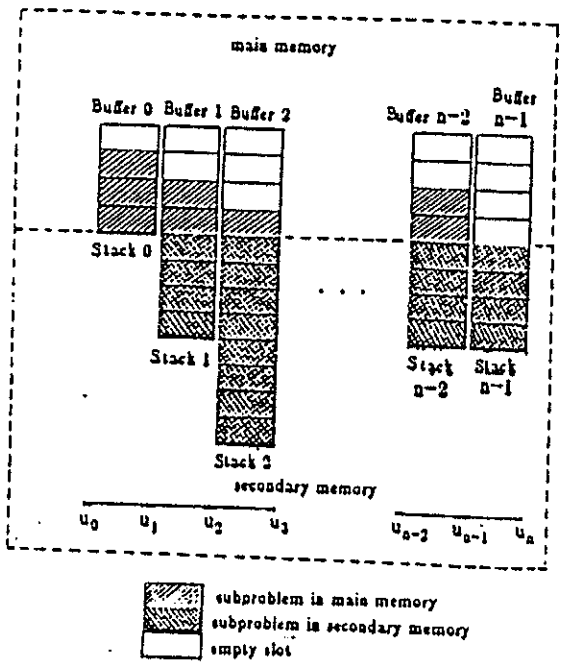


Figure 2.6 Data structure for the modified B&B algorithm.

Let the *current stack* be the stack that contains the active node with the smallest lower-bound, and let the *optimal stack* be the stack containing the optimal solution. A stack is *selected* when it becomes the current stack. Similar labels apply to the corresponding buffers and files. Stack S_j is defined to be *smaller* than stack S_k if $j < k$, and *larger* if $i > k$. Similar relationships hold for the buffers B_j and B_k.

The new data structure entail modifications to the selection and elimination rules. The *modified selection rule* selects the top element of the smallest non-empty buffer. This may or may not be current buffer. An empty current buffer triggers a transfer of nodes from the current file into the current buffer. The modified selection rule enables nodes to be expanded while this transfer is being carried out and avoids the long wasteful wait incurred by strict adherence to the fallible lower-bound heuristic.

Let the incumbent be z, and let the allowance function be $\epsilon(z)$. Suppose that $(z-\epsilon(z))$ lies in the range of stack S_k. The *modified lower-bound elimination rule* eliminates all active stacks S_i with $i > k$ that have not been previously eliminated. Nodes in F_k with lower-bounds greater than $(z-\epsilon(z))$ are not eliminated immediately because of the high overhead of accessing them in secondary memory. Furthermore, it is possible that a future incumbent will eliminate stack S_k. Thus, the culling of nodes in F_k is postponed until they have been read into main memory during the expansion of S_k. This necessitates an additional lower-bound test on each selected node before it is expanded. Figure 2.7 summarizes the *modified B&B algorithm*.

The modified B&B algorithm expands at least as many nodes as the corresponding best-first B&B algorithm. The depth-first expansion of the solution stack, S_r, results in the expansion of some nodes in S_r with lower-bounds greater than the optimal solution value. In addition, the modified selection rule results in the expansion of some nodes in stacks $S_{r+1},\ldots,S_{n-1}$. The best-first B&B algorithm never expands these nodes.

The modified B&B algorithm with one stack is equivalent to the depth-first B&B; the modified B&B algorithm with infinitely many stacks is equivalent to a best-first B&B algorithm. As the number of stacks increases, the number of expanded nodes generally decreases, while the secondary-memory overhead generally increases. The stack boundaries and the number of stacks should be chosen to maximize the overlap between computation and secondary-memory accesses.

```
procedure modified B&B algorithm;
/* Range of possible lower-bounds is divided into regions, each of which is oc-
   cupied by a stack S_i, 0 ≤ i ≤ n-1.  S_i has a main-memory buffer B_i and a
   secondary-memory file F_i. */
{
    incumbent = ∞;  B_0 = {P_0};  S_i = Ø, i = 1, ..., n-1;
    while there are non-empty stacks do {
      let S_k be the current stack;
      if B_k is empty and no read from F_k to B_k is outstanding then
         initiate read from F_k to B_k;
      if all buffers are empty then
         wait until read from any buffer is completed;
      apply modified selection rule;
      if (lower-bound of selected node < (z-ε(z))) then {
         expand selected node;
         for P ∈ {expanded nodes} do {
            if P is a feasible solution then {
               update incumbent;
               apply modified elimination rule }
            else if (g(P) < (z-ε(z))) then {
               if B_i is full then {
                  if frame pool is empty then
                     initiate write from B_i to F_i
                  get frame from frame pool };
               push P into the corresponding B_i }
         }
      }
    }
}
```

Figure 2.7 The modified B&B algorithm.

Before we present some strategies for choosing the stack boundaries, we will develop analytical expressions for the expected overhead for the modified B&B process. These analytical expressions form the basis of one scheme for choosing stack boundaries.

2.5.1.1 Expected Overhead for Real-Valued Problems

This analysis uses some results from the wall model of the best-first B&B algorithm. The wall model was developed prior to the work of this thesis and has been reported elsewhere [Yu83, WaY82, WaY85]. We will only include the relevant aspects of the model in this thesis; specifically, the expected numbers of nodes which have already been expanded and generated when the best-first B&B process expands a node with a specified lower-bound. As shown in Figure 2.8, this model consists of two moving walls. The left wall represents the lower-bound of the node being expanded. The back wall represents the incumbent. These values are normalized by the lower-bound of the initial problem, $g(P_0) = g_0$. For any node P_j, let $\Delta(P_j) = g(P_j) - g_0$. Initially, the left wall is at $\Delta(P_0) = 0$, and the back wall is at infinity. The left wall always moves to the right as the best-first B&B process always expands the active node with the smallest lower-bound which always increases the smallest lower-bound value. Moreover, the back wall always moves to the left as the incumbent can only decrease in value. Thus, successive node expansions cause the two walls to approach each other. The best-first B&B process terminates when the two walls meet.

The difference between the lower-bound of a node and the lower-bound of its parent is assumed to be an exponential random variable with density function, $f_G(y) = \lambda\, e^{-\lambda y}$ In addition, each node expansion is assumed to generate s new nodes.

Let $N(x)$ be the number of expanded nodes when the front wall is at x. The best-first B&B process must have already expanded all nodes with lower-bounds less than x before it expands a node with lower bound x. Thus, $N(x)$ may also be interpreted as the number of nodes in the search tree with lower-bounds not exceeding x. Let P_j be a child of P_i. Suppose that the increase in lower-bounds during this node expansion is known to be y. Then, $\Delta(P_j) \leq x$ if and only if $\Delta(P_i) \leq x-y$. This together with the assumption that each node expansion generates s new nodes leads to a renewal equation for the expectation of $N(x)$.

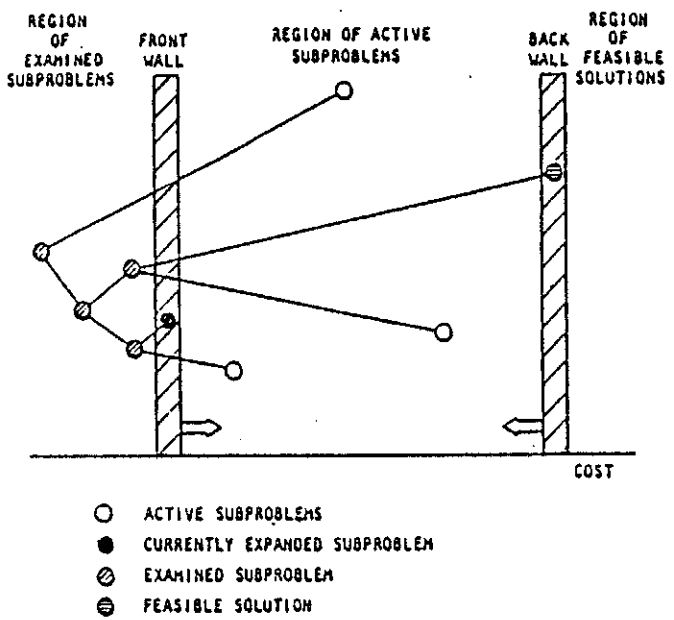


Figure 2.8 Wall model of the best-first B&B algorithm.

$$E[N(x)] = 1 + \int_{y=0}^{x} s\,E[N(x-y)]\,f_G(y)\,dy \qquad (2.29)$$

Assuming a reasonably large x, $f_G(y) \simeq 0$ for any $y > x$ as $f_G(y) \to 0$ as $y \to \infty$. This assumption simplifies Eq. (2.29) to

$$E[N(x)] = 1 + s \int_{y=0}^{\infty} E[N'(x-y)]\,f_G(y)\,dy \qquad (2.30)$$

Eq. (2.30) has a generic solution with the form $E[N(x)] = \left[k\,e^{mx} - \frac{1}{s-1}\right]$

Substituting the generic solution for E[N(x)] in Eq. (2.30) results in the identity:

$$k\,e^{mx} - \frac{1}{s-1} = 1 + s \int_{y=0}^{\infty} \left[k\,e^{m(x-y)} - \frac{1}{s-1}\right] f_G(y)\,dy$$

$$\text{or} \qquad 1 = s \int_{y=0}^{\infty} e^{-my} f_G(y)\,dy \qquad (2.31)$$

$$\text{or} \qquad m = \lambda(s-1) \qquad (2.32)$$

Solution of the constant k requires the use of the boundary condition $E[N(x_0)] = N_0$, where $x_0 + g_0$ is the lower-bound of the N_0-th node to be expanded. The value of N_0 should be chosen to eliminate any initial transients in the density function $f_G(y)$. Setting x to 0 in the generic solution yields $k = \frac{N_0(s-1)+1}{s-1} e^{-mx_0}$. Therefore,

$$E[N(x)] = \frac{N_0(s-1)+1}{s-1} e^{\lambda(s-1)(x-x_0)} - \frac{1}{s-1} \qquad (2.33)$$

Eq. (2.33) is an approximation when s is not an integer. Let M(x) be the number of generated nodes when the front wall is at x. Each node expansion generates s new nodes; so the expectation of M(x) is

$$E[M(x)] = s\,E[N(x)] + 1 \qquad (2.34)$$

Recall that N(x) may be interpreted as the number of nodes in the search space with lower-bounds not exceeding x. The expressions for E[N(x)] and E[M(x)] will be used in the analysis of the modified B&B algorithm in this context. Thus, $E[N(u_i)]$ represents the number of nodes expanded during the expansion of stacks S_1, ..., S_{i-1}, and $E[M(u_i)]$ is the expectation of the corresponding number of generated nodes.

Let $Z_1, \cdots, Z_{N(x)}$ be iid random variables with distribution $F_s(z \mid x, x \geq z)$. The assumption that the N(x) expanded nodes are independent of each other implies the stochastic equivalence of Z_i and $\Delta(P_i)$. Let z_i be the value of Z_i and assume, without any loss of generality, that $z_1 \leq \cdots \leq z_{N(x)}$. For any $z<x$, $E[N(z)]$ is the expected number of expanded nodes with lower-bounds not exceeding z. Stochastically equivalent random variables have identical means.

$$N(x)F_s(z \mid x, x \geq z) = E[N(z)]. \tag{2.35}$$

$E[N(z)]$ in Eq. (2.33) may be replaced by its expression in Eq. (2.35). This replacement leads to

$$F_s(z \mid x, x \geq z) = \frac{1}{(s-1)N(x)}\left(s e^{\lambda(s-1)z} - 1\right) \tag{2.36}$$

The density function, $f_s(z \mid x, x \geq z)$, is the differential of $F_s(z \mid x, x \geq z)$ with respect to z.

$$f_s(z \mid x, x \geq z) = \frac{1}{N(x)}\, s\,\lambda\, e^{\lambda(s-1)z} \tag{2.37}$$

Let Π_j, $j = 1, \ldots, N(x)$ be the N(x) expanded nodes and Ψ_k, $k = 1, \cdots, M(x)$ the M(x) generated nodes when the left wall is at x. Suppose that Π_j is Ψ_k's parent. If l_k is the difference between the lower-bounds of Ψ_k and Π_j then

$$\Delta(\Psi_k) = \Delta(\Pi_j) + l_k = Z_j + l_k \tag{2.38}$$

Eq. (2.38) implies that $\Delta(\Psi_1), \ldots, \Delta(\Psi_{M(x)})$ are iid random variables as both $Z_1, \ldots, Z_{N(x)}$ and $l_1, \ldots, l_{M(x)}$ are sets of iid random variables. Let $\gamma_i(x)$ be the probability that $\Delta(\Psi_k)$, $k = 1, \ldots, M(x)$ lies within the range of Stack i, (that is, $(u_i - g_0) < \Delta(\Psi_k) \leq (u_{i+1} - g_0)$) when the front wall is at x. Suppose that $Z_j = z$. $\Delta(\Psi_k)$ lies within $[u_i - g_0, u_{i+1} - g_0)$ if $l_k \in [u_i - g_0 - z, u_{i+1} - g_0 - z)$. Thus, the value of $\gamma_i(x)$ given Z_j will be

$$(\gamma_i(x) \mid Z_j = z) = \int_{y=u_i-g_0-z}^{u_{i+1}-g_0-z} \lambda e^{-\lambda y}\, dy$$

$$= \left(e^{-\lambda(u_i-g_0)} - e^{-\lambda(u_{i+1}-g_0)}\right) e^{\lambda z} \tag{2.39}$$

Eq. (2.39) may be unconditioned by using Eq. (2.37). Hence

$$\gamma_i(x) = \int_{z=0}^{x} \left(e^{-\lambda(u_i-g_0)} - e^{-\lambda(u_{i+1}-g_0)}\right) \frac{s\,\lambda\, e^{\lambda s z}}{N(x)}\, dz$$

$$\simeq \frac{1}{N(x)}\left(e^{-\lambda(u_i-g_0)} - e^{-\lambda(u_{i+1}-g_0)}\right) e^{\lambda s x} \tag{2.40}$$

There are M(x) generated nodes. Each generated node has probability $\gamma_i(x)$ of falling within the range of Stack S_i. The stochastic equivalence of Z_i and $\Delta(P_i)$ implies that the expected number of nodes in the region $[u_i - g_0, u_{i+1} - g_0)$ is

$$E[W_i(x)] = E[M(x)]\,\gamma_i(x) \simeq s\,E[N'(x)]\,\gamma_i(x)$$

$$\simeq \left(e^{-\lambda(u_i-g_0)} - e^{-\lambda(u_{i+1}-g_0)}\right) s\, e^{\lambda s x} \tag{2.41}$$

Let g_{opt} be the optimal solution value and let S_r be the solution stack (i.e., $u_r < g_{opt} \leq u_{r+1}$). In addition, let $E[q \mid r]$ be the expected number of nodes expanded before the modified B&B process finds the optimal solution in S_r. It is extremely difficult to model the LIFO expansion of S_r as it requires the lower-bounds of the individual nodes in S_r. Moreover, g_{opt} is not known until the problem is solved. Instead, we use the simplification that all nodes in S_r are expanded. This implies

$$E[q \mid r] = E[N(u_{r+1} - g_0)] \simeq \frac{s}{s-1}\, e^{\lambda(s-1)(u_{r+1}-g_0)} \tag{2.42}$$

The optimal solution value is assumed to be uniformly distributed over the range $[g_0, g^*_{opt})$. The actual distribution depends on the particular problem instance, and the assumed distribution is selected for reasons of mathematical tractability. With this assumption, the probability that S_r is the solution stack, π_r, for a problem with real-valued solutions will be given by

$$\pi_r = \frac{u_{r+1} - u_r}{g^*_{opt} - g_0} \tag{2.43}$$

For a problem with integer-valued solutions, π_r is

$$\pi_r = \frac{\left(\lfloor u_{i+1} \rfloor - \lfloor u_i \rfloor\right)}{\left(\lfloor g^*_{opt} \rfloor - \lfloor g_0 \rfloor\right)} \tag{2.44}$$

The modified B&B process expands $E[g \mid r]$ nodes when the optimal solution is in S_r. The probability of this event is π_r. Let h_c be the mean computational

time required to expand a node. The expected computational overhead, $E[t_c]$, is

$$E[t_c] = \sum_{r=0}^{n-1} E[q|\,r]\,h_e\,\pi_r$$

$$= \sum_{r=0}^{n-1} \left[\frac{s}{s-1}\,e^{\lambda(s-1)(u_{r+1}-g_0)}\right] h_e\,\pi_r \qquad (2.45)$$

Only the secondary-memory overhead due to the transfer of nodes will be considered in this analysis. The main memory allocated to node storage is divided into n_m frames. Each frame holds n_p nodes. n_0 frames are reserved for the depth-first expansion of the current stack.

Let m_i be the number of nodes in S_i when it is either selected or eliminated, and let $E[m_i|\,r]$ be its expectation when S_r is the solution stack. If $1 \le i \le r$ then m_i is the number of nodes in S_i when it becomes the current stack. $E[m_i|\,r] \simeq E[W_i(u_i-g_0)]$ if few nodes in S_i have been expanded before S_i becomes the current stack. If $(r+1) \le i \le (n-1)$ then m_i is the number of nodes in S_i when S_i is eliminated. $E[m_i|\,r] \simeq E[W_i(u_{r+1}-g_0)]$ if no stacks are eliminated prior to the generation of the optimal solution. This is a good approximation when the modified B&B process generates few feasible solutions before it finds the optimal solution, as in the integer-programming and vertex-cover problems. In short,

$$E[m_i|\,r] \simeq \begin{cases} E[W_i(u_i-g_0)] & 1 \le i \le r \\ E[W_i(u_{r+1}-g_0)] & (r+1) \le i \le (n-1) \end{cases}$$

$$\simeq \begin{cases} \left(e^{-\lambda(u_i-g_0)} - e^{-\lambda(u_{i+1}-g_0)}\right) s\, e^{\lambda s(u_i-g_0)} & 1 \le i \le r \\ \left(e^{-\lambda(u_i-g_0)} - e^{-\lambda(u_{i+1}-g_0)}\right) s\, e^{\lambda s(u_{r+1}-g_0)} & (r+1) \le i \le (n-1) \end{cases} \qquad (2.46)$$

n_0 frames are always reserved for the depth-first expansion of the current stack. A correct choice of the value of n_0 ensures that there is no overflow of nodes from the current buffer to the current file. Furthermore, the lower-bound test will remove all nodes falling within the range of an eliminated stack so no nodes will ever be written to an eliminated file. Hence, no more than m_i nodes from Stack S_i will be written to the secondary memory. Let b_i frames be required to hold m_i nodes. The expectation of b_i given the solution stack S_r is

$$E[b_i|\,r] = \left\lceil \frac{E[m_i|\,r]}{n_p} \right\rceil$$

$$\simeq \begin{cases} \left\lceil \dfrac{\left(e^{-\lambda(u_i-g_0)} - e^{-\lambda(u_{i+1}-g_0)}\right) s\, e^{\lambda s(u_i-g_0)}}{n_p} \right\rceil & 1 \le i \le r \\ \left\lceil \dfrac{\left(e^{-\lambda(u_i-g_0)} - e^{-\lambda(u_{i+1}-g_0)}\right) s\, e^{\lambda s(u_{r+1}-g_0)}}{n_p} \right\rceil & (r+1) \le i \le (n-1) \end{cases} \qquad (2.47)$$

Let c_i be the number of frames allocated to B_i when S_i either becomes the current stack or is eliminated. If no stack has been previously eliminated, the number of buffers holding frames is $(n-i-1)$ when $1 \le i \le r$ and $(n-r-1)$ when $(r+1) \le i \le (n-1)$. The frame-pool manager always reclaims the oldest full frames so the number of frames allocated to each buffer is proportional to the total number of nodes in each stack. Thus, the expected value of c_i when S_r is the solution stack is

$$E[c_i|\,r] = \begin{cases} \dfrac{\left(e^{-\lambda(u_i-g_0)} - e^{-\lambda(u_{i+1}-g_0)}\right)}{\left(e^{-\lambda(u_{\min(i,r+1)}-g_0)} - e^{-\lambda(u_n-g_0)}\right)} (n_m - n_0) & \sum_{j=\min(i,r+1)}^{n-1} E[b_j|\,r] \ge n_m - n_0 \\ E[b_i|\,r] & \sum_{j=\min(i,r+1)}^{n-1} E[b_j|\,r] < n_m - n_0 \end{cases} \qquad (2.48)$$

The modified B&B process writes at most $(b_i - c_i)$ frames to file F_i. These frames will be read back into the main memory if S_i is selected but will be discarded if S_i is eliminated. Let h_d be the overhead of transferring one frame of nodes between main memory and secondary memory. The expected secondary-memory overhead for S_i, given the solution stack S_r, will be the product of h_d and the number of frames transferred between main memory and secondary memory.

$$E[d_i|\,r] = \begin{cases} 0 & i = 0 \\ 2(E[b_i|\,r] - E[c_i|\,r])\,h_d & 1 \le i \le r \\ (E[b_i|\,r] - E[c_i|\,r])\,h_d & (r+1) \le i \le (n-1) \end{cases} \qquad (2.49)$$

Unconditioning for the solution stack S_r yields

$$E[t_d] = \sum_{i=0}^{n-1} \sum_{r=0}^{n-1} E[d_i | r]\, \pi_r \qquad (2.50)$$

The expected overhead of an algorithm in our computer model is the larger of the expected computation time and the secondary memory overhead. The expected overhead for the boundary assignment $\{u_0, \ldots, u_n\}$ is

$$E[D(u)] = \max\,(E[t_c], E[t_d]) \qquad (2.51)$$

2.5.1.2 Expected Overhead for Discrete-Valued Problems

The analysis for discrete-valued problems is similar to that for real-valued problems. We will only present the highlights of the analysis in this section. The increases in lower-bounds during a node expansion are now governed by a discrete distribution, the negative-binomial distribution,

$$f(y;r,p) = \binom{r+y-1}{y} p^r (1-p)^y \quad y = 0, 1, 2, \cdots \qquad (2.52)$$

r and p are parameters to be determined from the problem under consideration. This distribution has been verified experimentally for the vertex-cover problem.

Using the arguments and notations of Section 2.5.1.1, we can derive an approximate renewal equation for the expected number of expanded nodes when the front wall is at x.

$$E[N(x)] = 1 + s \sum_{y=0}^{\infty} E[N(x-y)] \times f(y;r,p) \quad x = 0, 1, \cdots \qquad (2.53)$$

The solution to Eq. (2.53) is:

$$E[N(x)] = \frac{N_0(s-1)+1}{s-1} \left(\frac{q}{1-ps^{1/r}} \right)^{(x-x_0)} - \frac{1}{s-1} \quad x = 0, 1, \cdots \qquad (2.54)$$

As before, the expectation of the corresponding number of generated nodes is

$$E[M(x)] \simeq s\,E[N(x)] + 1 \qquad (2.55)$$

Eq. (2.54) also enables us to derive an expression for the expected computational overhead.

$$E[t_c] = = \sum_{r=0}^{n-1} \frac{N_0(s-1)+1}{s-1} \left(\frac{q}{1-ps^{1/r}} \right)^{(u_{r+1}-x_0)} h_c\, \pi_r \qquad (2.56)$$

where h_c is the mean computational time to expand a node, and Π_r is the

probability that stack S_r is the optimal stack.

As mentioned in Section 2.5.1.1, N(x) also represents the number of nodes in the search space with lower-bounds not exceeding x The difference between N(y) and N(y-1) is the number of nodes with lower-bound y. This results in the following approximate expression for E[L(y)], the expected number of nodes with lower-bound y.

$$E[L(y)] \simeq \begin{cases} E[N'(y)] - E[N'(y-1)] & y = 1, \ldots, x \\ E[N'(0)] & y = 0 \end{cases} \qquad (2.57)$$

A node generated by the expansion of a node with lower-bound z lies within Stack S_i if the increase in lower-bound lies within $[u_i - z, u_{i+1} - z)$. The increase in lower-bound during a node expansion is an iid negative binomial random variable, so the probability of this event is $\sum_{v=u_i-z}^{u_{i+1}-z} f(v;r,p)$. Thus, the expected number of generated nodes in the region $[u_i - g_0, u_{i+1} - g_0)$ when the front wall is at x can be expressed as

$$E[W_i(x)] = \sum_{y=0}^{x} L(y) \left[\sum_{z=u_i-y}^{u_{i+1}-y-1} f(z;r,p) \right] \qquad (2.58)$$

As in Section 2.5.1.1, m_i denotes the number of nodes in stack S_i when it is either eliminated or selected. Using Eq. (2.45), we can show that the expectation of m_i given the optimal stack, S_r, is

$$E[m_i | r] \simeq \begin{cases} E[W_i(u_i - g_0)] & 1 \le i \le r \\ E[W_i(u_{r+1} - g_0)] & (r+1) \le i \le (n-1) \end{cases}$$

$$\simeq \begin{cases} \sum_{y=0}^{u_i - g_0 - 1} L(y) \left[\sum_{z=u_i-y}^{u_{i+1}-y-1} f(z;r,p) \right] & 1 \le i \le r \\ \sum_{y=0}^{u_{r+1} - g_0 - 1} L(y) \left[\sum_{z=u_i-y}^{u_{i+1}-y-1} f(z;r,p) \right] & (r+1) \le i \le (n-1) \end{cases} \qquad (2.59)$$

The number of frames required to hold these $E[m_i | r]$ nodes will be given by

$$E[b_i | r] = \left\lceil \frac{E[m_i | r]}{n_p} \right\rceil$$

$$\simeq \begin{cases} \left\lceil \dfrac{\sum_{y=0}^{u_i-g_0-1} L(y) \left\lceil \sum_{z=u_i-y}^{u_{i+1}-y-1} f(z;r,p) \right\rceil}{n_p} \right\rceil & 1 \le i \le r \\ \left\lceil \dfrac{\sum_{y=0}^{u_{r+1}-g_0-1} L(y) \left\lceil \sum_{z=u_i-y}^{u_{i+1}-y-1} f(z;r,p) \right\rceil}{n_p} \right\rceil & (r+1) \le i \le (n-1) \end{cases} \tag{2.60}$$

Using the arguments, assumptions, and notation of Section 2.5.1.1, we can then determine the number of frames allocated by each stack when it is either selected or eliminated.

$$E[c_i|\, r] = \begin{cases} \dfrac{\left\lceil \sum_{y=0}^{u_{\min(i,r+1)}-g_0-1} L(y) \left\lceil \sum_{z=u_i-y}^{u_{i+1}-y-1} f(z;r,p) \right\rceil}{\left\lceil \sum_{y=0}^{u_{\min(i,r+1)}-g_0-1} L(y) \left\lceil \sum_{z=u_{\min(i,r+1)}-y}^{u_z-y-1} f(z;r,p) \right\rceil} \sum_{j=\min(i,r+1)}^{n-1} E[b_j|\, r] & \ge n_m - n_0 \\ E[b_i|\, r] & \\ & < n_m - n_0 \end{cases} \tag{2.61}$$

As before, the expected secondary memory overhead for Stack S_i due to the transfer of nodes when S_r is the solution stack will be given by

$$E[d_i|\, r] = \begin{cases} 0 & i = 0 \\ 2(E[b_i|\, r] - E[c_i|\, r])\, h_d & 1 \le i \le r \\ (E[b_i|\, r] - E[c_i|\, r])\, h_d & (r+1) \le i \le (n-1) \end{cases} \tag{2.62}$$

where h_d is the overhead for each page read. Only the secondary memory overhead due to the transfer of nodes is considered in this analysis, so the modified B&B process incurs a secondary memory overhead given by

$$E[t_d] = \sum_{i=0}^{n-1} \sum_{r=0}^{n-1} E[d_i|\, r]\, \pi_r \tag{2.63}$$

The expected overhead for n stacks and a boundary assignment $(u_0, \ldots, u_n)$ is

$$E[D(n)] = \max\,(E[t_c], E[t_d]) \tag{2.64}$$

2.5.1.3 Choosing Stack Boundaries

The analytical expressions for E[D(n)] form the basis of one scheme for choosing stack boundaries. This analytical strategy finds the assignment that minimizes E[D(n)] for a given n by enumerating E[D(n)] over feasible assignments. analytical scheme is the equal-width strategy which is a heuristic method that divides the range of possible lower-bounds into equal-width regions.

We have carried out extensive simulations of the analytical (AA) and equal-width (EWA) strategies using integer-programming and vertex-cover problems. Table 2.2 shows some of these results; the values for the overhead have been normalized with h_c, the mean computational time required to expand a node. These simulations suggest that the analytical and equal-width strategies have generally similar performances.

These extensive simulations also suggest the use of either *three or four stacks* for integer-programming and vertex-cover problems to minimize the overhead. Table 2.2 clearly shows this.

2.5.2 Modified B&B Algorithm for a Virtual-Memory System

The stacks reside in virtual space when the modified B&B algorithm is implemented with the virtual-memory management system. The virtual-memory management system automatically handles all node transfers between main memory and secondary memory so there is no need for the frame pool and its associated routines. However, reads from secondary memory to main memory are no longer overlapped with computations and the overhead analysis must be altered accordingly. Otherwise, the analysis is similar to that for the file system and will not be shown here.

2.6 Comparison of the Different B&B Algorithms

We have carried out extensive simulations to compare the different B&B algorithms. The simulation programs were written in the C language. These simulations used over one hundred hours of CPU time on the VAX 11/780. These simulators simulate a computer with a main memory of 128 Kbytes (10% of the virtual space).

Table 2.2 Normalized overheads for the analytical (AA) and equal-width (EWA) boundary assignments for five randomly generated, 80-node vertex-cover problems and five randomly generated, 20-by-20 integer programming problems.

Number of Stacks	Normalized Overhead									
	Problem 1		Problem 2		Problem 3		Problem 4		Problem 5	
	EWA	AA	EWA	AA	EWA	AA	EWA	AA	EWA	AA
Vertex-cover problems with $h_d = 5\,h_c$ and $h_d = 10\,h_c$										
2	9426	6196	8690	7274	11752	9714	11891	7129	13160	5186
3	7047	6139	7157	6363	9231	9548	10423	10942	7260	5186
4	5987	5942	8671	8257	9213	9120	11873	7662	5607	5176
5	8271	8402	7218	6975	9888	10216	9894	6996	6894	5164
Integer-programming problems with $h_d = 5\,h_c$										
2	4598	2058	2580	3561	581	646	2906	2965	2300	3086
3	2555	1793	3412	3334	1419	612	2953	2510	1833	1633
4	2040	3793	2200	2102	976	581	2237	3055	3541	2087
5	1541	1565	2951	3610	680	581	2468	2287	1875	2037
Integer-programming problems with $h_d = 10\,h_c$										
2	4958	2058	5160	3561	581	646	2906	4930	2300	3086
3	4370	5270	2680	2976	1300	605	4480	2510	2500	4010
4	3600	3580	4400	4970	976	1710	3400	6220	5290	2070
5	1680	2090	4700	3990	1360	713	3410	3440	3750	5360

Figure 2.9 shows some of our simulation results for 20-variable, 20-constraint integer-programming problems. Each node requires 1974 bytes of memory space. The modified B&B scheme uses three stacks and a logical file block size of 8 Kbytes. The direct implementation uses a page size 8 Kbytes. The modified virtual-memory scheme uses a page size of 16 Kbytes. We assumed a disk with a mean seek time of 28 ms., a mean rotational delay of 8.33 ms., and track capacities of 16 and 64 Kbytes (corresponding to transfer rates of approximately 1 and 4 Mbytes/sec.). The overhead is the completion time of the algorithm normalized by h_c. For the range of h_d/h_c simulated, the modified B&B scheme performs significantly better than the other schemes. Other simulations exhibited similar results.

Figure 2.10 shows some of our simulation results for the 80-node vertex-cover problems. Each node requires 330 bytes of memory. Here, the depth-first B&B algorithm performs as well as the modified B&B algorithm. This agrees with our expectation as the vertex-cover problem has a very accurate lower-bound function.

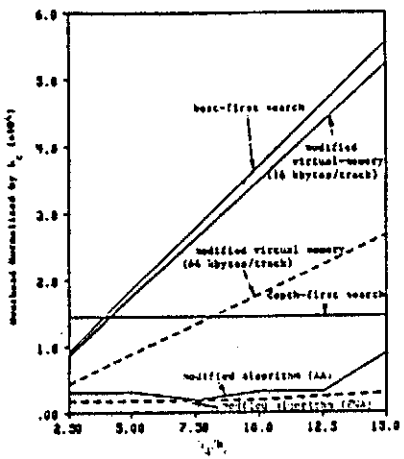
2.7 Summary

In a two-level memory system, the most time-efficient B&B algorithm depends on the numbers of nodes expanded by the B&B algorithms and the relative speeds of the main and secondary memories (Table 2.3).

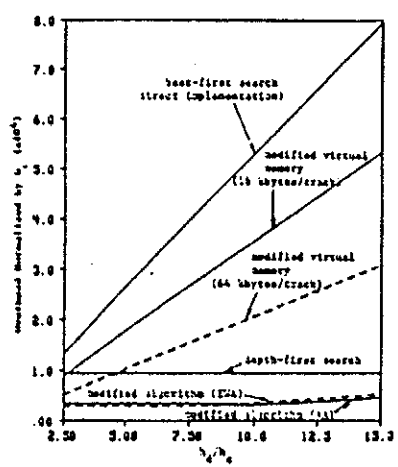
The best-first B&B algorithm expands the smallest number of nodes but requires a very large memory space. In contrast, the depth-first B&B algorithm is space efficient but expands more nodes than the best-first B&B algorithm. The modified B&B algorithm represents a compromise between the best-first and depth-first B&B algorithms.

Thus, the best-first B&B algorithm should only be used when the extra overhead due to its memory requirement is small. This requirement is met when the average time to transfer a node between main memory and secondary memory is close to the average time to expand a node. For slower secondary memories, the best-first B&B algorithm should be chosen only when it expands much less nodes than the other B&B algorithms.

The depth-first B&B algorithm should be used to solve the problem for which there is only a small difference between the number of nodes expanded by the depth-first B&B algorithm and the number of nodes expanded by the best-first B&B algorithm. A greater difference is acceptable if the secondary

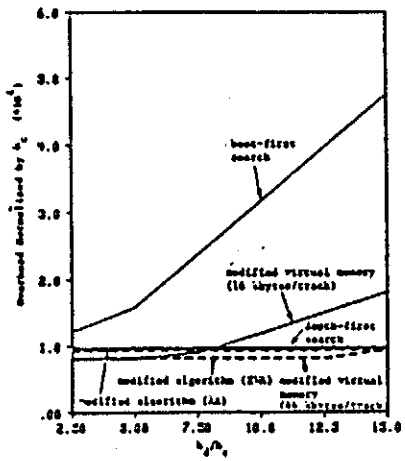


(a) Problem 1

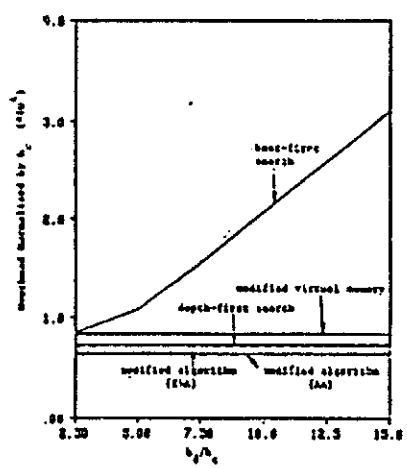


(b) Problem 2

Figure 2.9 Simulation results for B&B algorithms on two 20-by-20 integer programming problems. (Main memory size is 128 Kbytes.)



(a) Problem 1



(b) Problem 2

Figure 2.10 Simulation results for B&B algorithms on two 80-node vertex cover problems. (Main memory size is 128 Kbytes. The completion time for the modified B&B scheme is the same for both 16 Kbytes/track and 64 Kbytes/track in the second problem.)

Table 2.3 Summary of B&B algorithms to be used.

n_{dfs} / n_{bfs}	Relative Speed of Secondary Memory		
	fast	moderate	slow
small	dfs	dfs	dfs
moderate	bfs/mbb	mbb	dfs/mbb
large	bfs	bfs	bfs/mbb

Keys: dfs: depth-first search;
bfs: best-first search;
mbb: modified branch-and-bound algorithm;
n_{bfs}: number of nodes evaluated in bfs;
n_{dfs}: number of nodes evaluated in dfs.

memory is much slower than the main memory.

There will be problems for which both the best-first B&B algorithm and the depth-first B&B algorithm perform poorly: the depth-first B&B algorithm expands an excessive number of nodes, while the best-first B&B algorithm uses an excessive amount of memory. The modified B&B algorithm may be used for these problems.

The design of the modified B&B algorithm illustrates one important characteristic of many non-deterministic search problems. The non-determinism arises because the search is guided a fallible heuristic function that does not always lead in the right direction and results in backtracking when a wrong decision is made. Strict adherence to the heuristic function may not always be rewarding when it incurs a high overhead. In this case, adherence to the heuristic function should be relaxed. In virtual memory systems, this relaxation may be accomplished by processing other nodes while waiting for the node indicated by the heuristic function to be transferred from the secondary memory to the main memory. item indicated by the

CHAPTER 3
LEARNING DOMINANCE RELATIONS

3.1 Introduction

Dominance relations are powerful, problem-dependent pruning rules used in combinatorial search algorithms to eliminate nodes that will not lead to the optimal solution. The problem-dependent nature of dominance relations has inhibited the development of a general procedure for the derivation of dominance relations. However, it is possible to overcome this obstacle by knowledge-based artificial intelligence techniques, specifically machine learning techniques. This chapter studies machine learning of dominance relations. Section 3.2 describes the five step methodology used in this study. Sections 3.3 to 3.6 describes the results of applying the first three steps and part of the fourth step in this methodology to the problem of learning dominance relations. Chapter 4 presents the rest of the results.

3.2 Methodology

Winston [Win84] has suggested the following commandments of a good AI methodology:

- First, identify the problem.
- Then select or devise an appropriate representation.
- Next expose constraints or regularities.
- Only now create particular procedures.
- Finally, verify via experiments.

Our methodology closely follows these commandments. We start by identifying the problem that we want to solve. Successful AI systems often employ domain knowledge. Combinatorial search problems come from many different problem domains. Moreover, little is known about the process of deriving dominance relations, so it is prudent to identify a particular subset of the combinatorial search problems to study instead of tackling the general problem at this stage. Section 3.3 discusses this topic in greater detail.

This is followed by a study of appropriate representations. A representation is a set of conventions about how to describe a class of things. A description makes use of these conventions to describe a member of this class of things. In theory, all computer-based representations are identical. In practice, one scheme may be better than another because it offers more convenience to the user. There are two issues to address in this area: the representation for the dominance relations and the representation for the search problems. Section 3.4 discusses these issues.

Next, we look for constraints in the problem domain which can simplify the task of finding dominance relations. This involves a two-step process. First, we seeks a characterization of the structure of the domain occupied by dominance relations. Next, we seek the constraints, if any, on the permissible forms of the dominance relations exposed by their characterization. Possible constraints are trivial relations and relations with implementation overheads that clearly exceed the savings resulting from their use. Section 3.5 covers this topic.

We then set out to devise the learning mechanisms which carry out the actual work of learning the dominance relations. We employ a two-phase procedure. The *initial phase* seeks to identify and describe the applicable learning mechanisms. These descriptions include the aspects of the problem to which each learning mechanism may be applied, the important issues in their implementations, and the interactions, if any, among the different learning mechanisms. Section 3.6 discusses the initial phase of the process of devising learning mechanisms for the dominance relations.

The *refinement phase* develops the actual procedures (production rules) that implement the identified learning mechanisms. Each learning mechanism is developed using the *design-prototype-test* method commonly used in the development of knowledge intensive systems. Here, an initial design is created and a prototype implementing the design is tested. The test results often suggest modifications to the design. Once the appropriate modifications have been made to the prototype, further testing of the prototype is carried out. The new test results, in turn, may suggest additional modifications to the design. This process continues until the design goals have been met. At times, the test results may require such extensive modifications that a redesign of the entire system may be needed.

When there are more than one learning mechanism, it is prudent to interlace the refinements of the different learning mechanisms. Work on a learning mechanism may bring new insights to the design of another learning

mechanism. This allows work to proceed more smoothly than if attention were paid solely to one mechanism.

Sometimes, it may also be necessary to modify the structures resulting from steps one, two, and three of the methodology so that they are consistent with the results generated by the continuing refinement of the different learning mechanisms.

Chapter 4 describes the design of a prototype system that implements one of the learning mechanisms for dominance relations. We have experimented and verified the capabilities of the prototype system. Chapter 4 also summarizes the results of applying the prototype system to several test problems.

3.3 Problem Definition

As indicated in Section 1.2, combinatorial search problems may be divided into decision problems and optimization problems. Two functions may be associated with the optimization problem, I, [GaJ79]: Length[I] is the number of symbols used to describe I under some reasonable encoding scheme, and Max[I] is the magnitude of the largest integer occurring in problem I. The optimization problem, I, is a *number optimization problem* if there is no polynomial, p, such that $Max[I] \leq p(Length[I])$. A similar definition holds for the *number decision problem*. The solution of a combinatorial search problem requires *pseudo-polynomial time* if its time complexity function is bounded from above by a polynomial function of the two variables Length[I] and Max[I]. Most known dominance relations are associated with problems solved by dynamic programming, and these problems usually have pseudo-polynomial time solutions.

It has often been said among AI researchers that learning something new requires knowing a lot about it already. Thus, it is prudent to study a problem domain known to be rich in dominance relations. Many problems that are known to have dominance relations are number optimization problems; thus, the problems studied in this thesis are the number optimization problems. Moreover, dominance relations are generally more effective in problems with pseudo-polynomial time solutions than in problems with exponential time solutions. Dominance relations do not change the time complexity of the search process. As a result, dominance relations generally produce a smaller percentage improvement in problems with exponential time solutions.

3.4 Representation

The first facet of the representation problem is the choice of a representation for the dominance relations. A dominance relation is essentially a statement that if two nodes satisfy the conditions set forth in the dominance relation, then one node will contain a solution no worse than the other node. This statement can be naturally expressed by an IF-THEN rule (production rule). The left-hand side establishes the conditions that two nodes must satisfy in order for one node to dominate the other node. The right-hand side establishes the relationship between the best solutions contained in the two nodes. This production rule can easily be transformed into an implication in first-order predicate logic, so first-order predicate logic is an equally good representation for dominance relations. This easy transformation permits the switching between production rules and predicate logic as the situation dictates.

The other facet of the representation problem is the representation for the number optimization problems. A good representation exposes aspects of the problem involved in learning dominance relations. This requirement implies the need for relevant domain knowledge about the problem. A good representation also permits techniques developed for one problem to be used for other related problems. An important aspect of this requirement is a systematic method of incorporating domain knowledge in the representation so that relationships may be drawn between the domain knowledge of different problems.

One way of expressing a number optimization problem is in the form of a constrained optimization:

$$\begin{array}{lll} \text{Minimize} & C_0(x) & x \in X \\ \text{subject to} & g_i(x) \leq 0 & i = 1, 2, \ldots, m \end{array}$$

This description is inadequate for learning dominance relations as it does not include any domain information. The relevant domain information is related to the subject matter of the problem. One classification of combinatorial search problems [GaJ79] lists twelve major subject areas: graph theory, network design, sets and partitions, storage and retrieval, sequencing and scheduling, mathematical programming, algebra and number theory, games and puzzles, logic, automata and languages, program optimization, and miscellaneous. Each major subject area is divided into several minor subject areas. The number optimization problems can occur in any of these subject areas except logic.

Studying all of these subject areas is non-trivial and time-consuming. As this study is an initial study of the problem of learning dominance relations, we choose a more detailed study of some of the subject areas instead of a less detailed study of more subject areas. The work of this thesis uses problems in mathematical programming and problems in sequencing and scheduling. We have substantial experience with problems in these two areas, and have abundant heuristic knowledge about them. Another reason for this choice is that the problems in these two subject areas are somewhat related. Mathematical programming problems studied in this thesis are the 0/1-knapsack problem, the inventory problem, and machine reliability by duplication. Problems in sequencing and scheduling studied in this thesis are sequencing to minimize weighted completion time, sequencing to minimize latest completion time, sequencing to minimize weighted lateness, sequencing to minimize task weights, sequencing to minimize tardy task weights, and the two-machine flowshop problem.

We will use the 0/1-knapsack problem, the inventory problem, and sequencing to minimize weighted completion time to illustrate the design of the problem representation.

The 0/1-knapsack problem consists of a set U of n items. The knapsack has a capacity K. Each item $u \in U$ has a weight w(u) and a value r(u). The desired solution is the subset $U' \subset U$ which maximizes $\sum_{u \in U'} v(u)$ and satisfies $\sum_{u \in U'} w(u) \leq K$.

In sequencing to minimize weighted completion time, there is a set T of n tasks. Each task $t \in T$ has a length $\ell(t)$, a weight w(t), and a deadline d(t). The desired solution is the one-processor schedule σ for T which minimizes $\sum_{t \in T} (\sigma(t) + \ell(t))w(t)$ and satisfies $(\sigma(t) + \ell(t)) \leq d(t)$ for each task $t \in T$.

The inventory problem deals with the storage of an item over a set M of n periods. Initially, there are v units of the item. In each period $m \in M$, the item has a unit purchase price of b(m) and a unit sale price of s(m). The goal is to determine the amount purchased $x(m_i)$ and the amount sold $y(m_i)$ in each period m_i, $1 \leq i \leq n$, which will maximize $\sum_{i=1}^{n} (x(m_i) \times s(m_i) - y(m_i) \times b(m_i))$. There can at most be B units of the item at any time, so $v + \sum_{i=1}^{j} (x(m_i) - y(m_i)) \leq B$ for all $j \in \{1, \ldots, n\}$. Amounts purchased are delivered at the end of each month, so the amount sold in each month must not exceed the stock at the end of the previous month, and $y(m_i) \leq v + \sum_{j=1}^{i-1} (x(m_j) - y(m_j))$ for all $i \in [1,n]$.

Although these three problems involve quite different domains, they are quite similar in form (Table 3.1). Each problem description has one set of entities. A set of decisions must be made on a set of entities and a solution is found when decisions have been made for every entity. Each description contains constraints which define the allowable sets of decisions. Each description also contains an optimization criterion which defines the desired optimal solution. This is either the maximization or the minimization of a objective expression. The objective expression and the constraints are arithmetic expressions.

There are three types of symbols in the descriptions: *decision symbols*, *entity variables*, and *problem constants*. Decision symbols are associated with the set of decisions. Entity variables are associated with the set of entities. The entity variables may be subdivided into *return properties*, *resource properties*, and *limit properties*. The return property appears in the optimization criteria. Resource properties and limit properties appear in the constraints. Resource properties are usually combined with decision symbols in the constraints. Limit properties establish the limits in the constraints. It is possible for an entity variable to be both a return property and a resource property. The remaining symbols in the descriptions are problem constants. Problem constants include set sizes, initial values, and global limits.

Before a search problem can be solved efficiently, there must be a bookkeeping scheme that allows efficient orderly exploration of all possible solutions. A key in such a definition of the search space is the transformation of the set of decisions into a form suitable for efficient sequential decision making. Let this new set be known as the set of working decisions. The subset U' in the 0/1-knapsack problem has to be transformed into the set of membership decisions $\{x(u) : u \in U\}$. In scheduling to minimize weighted completion time, the schedule time for task t, $\sigma(t)$, is usually simplified into the position of task t in the schedule, $\rho(t)$. These transformations often require deep domain knowledge. For instance, the replacement of $\sigma(t)$ by $\rho(t)$ is possible only after knowing that there is no waits between consecutive tasks in the optimal schedule. Problem solvers make working decisions by assigning values to *decision variables*.

Let the problem description actually used by the problem solver be known as a *working description*. The first step in forming a working description from

Table 3.1 Similarities between the problems descriptions of the 0/1-knapsack, scheduling to minimize weighted completion time, and inventory problems.

Problems	0/1 KNAPSACK	SCHEDULING	INVENTORY
Entity	item u, set U of items	task t, set T of tasks	period m, set M of periods
Decision	subset U'	schedule σ	amount purchased x(m), amount sold y(m), $\forall m \in M$
Constraints	$\sum_{u \in U'} w(u) \leq K$	$\lvert\sigma(t) + \ell(t)\rvert \leq d(t)$, $\forall t \in T$	$v + \sum_{i=1}^{j} [x(m_i) - y(m_i)] \leq B$, $y(m_i) \leq v + \sum_{j=1}^{i-1} [x(m_j) - y(m_j)]$, $\forall i,j \in \{1,n\}$
Objective Criteria	maximize $\sum_{u \in U'} v(u)$	minimize $\sum_{t \in T} [\lvert\sigma(t) + \ell(t)\rvert - w(t)]$	maximize $\sum_{i=1}^{n} [x(m_i)s(m_i) - y(m_i)b(m_i)]$
Symbols Decision Symbols Entity Properties System Constants	U' v(u), w(u) K, n	$\sigma(t)$ $\ell(t)$, w(t), d(t) n	 x(m), y(m) b(m), s(m) v, n
Working Decision	x(u)	$\rho(t)$	x(m), y(m)

the problem description is to replace the decision symbols by decision variables. Working descriptions use state variables to assist in the assignment of values to the decision variables. State variables record the effects of previous decisions.

A complete specification of a working decision requires a two-tuple *(e-tuple, v-tuple)* where *e-tuple* identifies its entity and *v-tuple* specifies its value. Decision variables in a *general-description* explicitly specify both the e-tuple and the v-tuple. A general description is not necessary for most problems. The decision variable, v_j, in a *value-description* determines the v-tuple of the corresponding working decision. Value-descriptions usually assume some fixed order of the entities $E_1, E_2, \ldots, E_{n_E}$ The decision corresponding to variable v_j is (E_j, v_j). A value-description produces a fixed height search tree in which all solution nodes are at a fixed distance from the root. In contrast, the decision variable, i_j, in an *identity-description* only specifies the value of the e-tuple. One common type of identity-description assumes a constant value for the v-tuple of the decision whose e-tuple is given by the decision variable. This assumption is often combined with the assumption of some other constant value for the v-tuples of all unassigned decisions. Another type of identity-description assumes that the decision made by decision variable i_j is (i_j, j). The identity-description results in a search tree that may either be a fixed-height tree or a variable-height tree.

The 0/1-knapsack problem has a value-description as well as an identity-description. Decision variable, v_j, is the membership status of item u_j. v_j has value 0 if item u_j is to be excluded from the knapsack and has value 1 if u_j is to be included in the knapsack. In contrast, decision variable, i_j, represents the identity of an item that is to be included in the knapsack.

Sequencing to minimize weighted completion time also has a value-description as well as an identity-description. Decision variable, v_j, is the position of task t_j in the schedule. Decision variable, i_j is the identity of the task occupying position j in the schedule.

The inventory problem only has a value-description. The lack of a suitable implicit value for the v-tuple prevents the inventory problem from having an identity-description. For even values of j, v_j is the amount purchased in month $j/2$. When j is odd, v_j is the amount sold in month $\lfloor j/2 \rfloor$.

The similarities between the different problem descriptions strongly indicate the presence of a domain-independent structure in the problem descriptions. Each problem description appears to have the same set of domain-independent elements: the entities, the decision variables, the state

variables, the entity variables, the problem constants, the optimization function, and the constraints. Furthermore, this set of elements is complete, that is, the problem description can always be completely described in terms of the basic elements.

This structure can be exposed by a frame representation [Min75]. The slots in the frame are fixed and represent the different elements of the problem description. The slot fillers describe the actual values of these elements for the particular problem being described.

Figure 3.1 shows the description frames for the 0/1-knapsack problem. The problem description frames for the other problems studied in this thesis are similar. The slots in the description frame are domain-independent. Hence, we can expect this frame representation to be applicable to all number optimization problems.

3.5 Constraints in the Problem Domain

As indicated in Section 3.2, the first step in exposing constraints in the problem domain is to find a suitable characterization of the problem domain. One possible classification of dominance relations is a classification based upon the possible positions of the dominating and dominated nodes. Each type of dominance relations is characterized by a three tuple (T_1, T_2, T_3). T_1 is the set of permissible levels for the dominating node, T_2 is the set of permissible differences between the levels of the dominating and dominated nodes, and T_3 is the relation between the dominating and dominated nodes.

The possible symbolic values for T_1 are *term* and *all*. The dominating node must be a terminal node when T_1 is *term* but can be any terminal or non-terminal node when T_1 is *all*.

The value of T_2 can be *same*, *specific*, or *all*. If T_2 has the value *same*, the dominating node and dominated node must be on the same level, that is, they are *equilevel* nodes. When T_2 is *specific*, the difference between the level of the dominating node and the level of the dominated node must be a symbolic constant. There is no restriction on the position of the dominated node when T_2 is *all*.

T_3 has a value of either *sib* or *all*. The dominated node must be a sibling of the dominating node if T_3 has the value *sib* but can be any arbitrary node if T_3 has the value *all*. T_3 can take on a value of *sib* only when T_2 has a value of *same*; otherwise, the resulting classification does not make any sense.

ENTITY	$u_1, \ldots, u_n$
DECISION VARIABLE	$x(u_i) = \{0,1\}$, $i=1, \ldots, n$
STATE VARIABLES	R(apply(x(u_i),NODE))
	= R(NODE) + x(u_i)•r(u_i);
	W(apply(x(u_i),NODE))
	= W(NODE) + x(u_i)•w(u_i);
	R(ROOT)=0; W(ROOT)=0
ENTITY VARIABLES	r(u_i), w(u_i)
PROBLEM CONSTANTS	K, n
OPTIMIZATION	maximize R(S);
	S = apply(x(u_n)...x(u_1),ROOT)
CONSTRAINT	W(NODE) $\leq$ K

(a) Value description

ENTITY	$u_1, \ldots, u_n$
DECISION VARIABLE	$x(u_i) = (u_1, \ldots, u_n)$ $i=1,\ldots,n$
STATE VARIABLES	R(apply(x(u_i),NODE))
	= R(NODE) + r(x(u_i))
	W(apply(x(u_i),NODE)
	= W(NODE) + w(x(u_i))
ENTITY VARIABLES	r(u_i), w(u_i)
PROBLEM CONSTANTS	K, n
OPTIMIZATION	maximize R(NODE)
CONSTRAINT	W(NODE) $\leq$ K

(b) Identity description

Figure 3.1 Description frames for the 0/1-knapsack problem.

Table 3.2 lists examples of the different types of dominance relations. A relation of type {*term*, *same*, *sib* or *all* } is a trivial relation as it is a relation between terminal or solution nodes. The value of the solution represented by the terminal nodes are known, so it is trivial to determine which is the better solution.

The lower-bound test, used in B&B algorithms as the elimination rule, has type {*term*, *all*, *all*}. The dominating node in the lower-bound test is the best solution node found so far and it can dominate any any node whose lower bound is worse than its value.

A greedy algorithm always chooses the optimal value for each decision variable. This implies that the search tree generated is a skewed tree in which one node dominates all its siblings at each level in the search tree and that greedy algorithms are dominance relations of type {*all*, *same*, *sib*}.

The upper-bound test is an example of a dominance relation of type {*all*, *all*, *all*}. In the upper bound test, a node dominates another node if the upper-bound of the first node is not greater than the lower-bound of the second node. No restrictions are placed on the positions of the dominating and dominated nodes.

Dominance relations of type {*all*, *specific*, *all*} are usually trivial relations or relations that correspond to well-behaved relations in other descriptions. For example, each node in the problem space generated by the identity-description for the 0/1-knapsack problem is a feasible solution and dominates its parent node. This relation corresponds to the trivial relation between terminal nodes in the corresponding value-description.

This interpretation of dominance relations with type {*all*, *specific*, *all*} implies that there is no need to explicitly look for dominance relations of this form as they will either be trivial relations or be found in another form.

Another constraint on the form of the useful dominance relations comes from studying the implementation overheads of the different types of dominance relations. The implementation overhead can be measured by the number of nodes to which each newly-generated node must be compared in order to determine if it either dominates another active node or is dominated by an active node (Table 3.3). The number of comparisons required by dominance relations between nodes on arbitrarily different levels is on the order of the number of active nodes. Unless the dominance relation is very powerful, the number of active nodes grows exponentially as the search proceed. It is very likely that the implementation overhead of such dominance relations will exceed the savings resulting from the nodes that they eliminate. Thus, there is

Table 3.2 Examples of the different classes of dominance relations.

T_1	T_2	T_3	EXAMPLE
term	same	sib	trivial relations
		all	trivial relations
	specific	all	
	all	all	lower-bound test
all	same	sib	greedy algorithm
		all	dominance relations in dynamic programming
	specific	all	
	all	all	upper-bound test

also no need to look for dominance relations of types {*all, all, all*}.

Furthermore, dominance relations of types {*term, same, sib*} and {*term, same, all*} are trivial relations. Hence, a study of methods to find dominance relation may be restricted to dominance relations belonging to the classes {*all, same, sib*} and {*all, same, all*}.

3.6 Applicable Learning Mechanisms

Learning is a very general term denoting the way people (and computers) increase their knowledge and improve their skills. Figure 3.2 shows a simplified model of a machine learning system [CoF82]. The environment supplies some information to the learning element, the learning element uses this information to make improvements in the explicit knowledge base, and the performance element uses this knowledge base to perform its task. Information gained during attempts to perform the task provide feedback to the learning element.

The key factor in the design of learning systems is the information that it receives from the environment - particularly its level and its quality. The level of information refers to its degree of generality relative to the needs of the performance element. High-level information is abstract information that is relevant to a broad class of problems. Low-level information is detailed information that is relevant to only a few problems. The learning element bridges this gap between the level of the information provided by the environment and the level required by the performance element. Thus, if the learning system is given very abstract (high-level) information, it must fill in the missing details that allow the performance element to interpret the information in particular situations. Correspondingly, if the system is given very specific (low-level) information about how to perform in particular situations, the learning element must generalize this information by ignoring unimportant details into a rule that can be used by the performance element in other related situations.

The learning element does not know in advance exactly how to fill in the missing details or ignore unimportant details because its knowledge is imperfect. Consequently, it must guess, that is, form hypotheses, about how the gap between the levels should be bridged. After guessing, the system must receive some feedback that allows it to evaluate its hypotheses and revise them if necessary. Thus, a learning system learns by trial and error.

The level of information provided by the environment determines the kinds of hypotheses that the system must generate and determines the type of

Table 3.3 Upper bounds on the implementation overheads of different classes of dominance relations (LEV(A) is the level of node A).

T_1	T_2	T_3	DOMINATED NODE	DOMINATING NODE
term	same	sib	O(1)	O(1)
		all	O(1)	O(1)
	spec.	all	$O(\lvert$ active terminal nodes $\rvert)$	$O(\lvert$ nodes on level specific $\rvert)$
	all	all	$O(\lvert$ active terminal nodes $\rvert)$	O(|active nodes|)
all	same	sib	O(|siblings|)	O(|siblings|)
		all	$O(\lvert$ nodes on the same level $\rvert)$	$O(\lvert$ nodes on the same level $\rvert)$
	spec.	all	O(|nodes on level {LEV(A) - specific}|)	O(|nodes on level {LEV(A) + specific}|)
	all	all	O(|active nodes|)	O(|active nodes|)

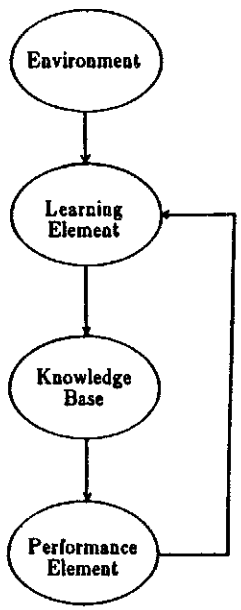


Figure 3.2 A simplified model of a machine learning system

learning strategy used [CoF82, CMM83, Mic86]. Several basic strategies have been distinguished: *rote learning, learning by instruction, learning by deduction, learning by analogy,* and *learning by induction.* The latter subdivides into *learning from examples* and *learning by observation and discovery.* These strategies are ordered by the increasing complexity of the hypothesis (inference) and reflects increasing effort on the part of the learning element and correspondingly decreasing effort on the part of the environment (teacher).

In rote learning, the environment supplies knowledge in a form that can be used directly. The learning system need not understand or interpret the information supplied by the environment. All it does is to memorize the information for later use. One major concern here is how to index the stored knowledge for future retrieval. Rote learning may be viewed as a form of data reduction [LHK79], that is, it reduces a calculation to an access task by saving the input/output details of some computation and so bypass any future need for the intermediate computation process. Rote learning was used in Samuel's famous checker-playing program [Sam59, Sam67]; the program was able to improve its performance by memorizing board positions and their associated minimax scores. It uses previously memorized board positions to improve the speed and depth of its look-ahead search during subsequent games.

Rote learning can only be used for problems with known dominance relations and so cannot learn new dominance relations.

In learning from instruction, knowledge is acquired from a teacher or an organized source such as a textbook. Here, the system is given general-purpose knowledge or advice. It transforms this high-level knowledge into a form that can be used readily by the performance element. This transformation is called *operationalization.* The system must understand and interpret the high-level knowledge and relate it to what it already knows. Operationalization is an active process that can involve such activities as deducing the consequences of what it has been told, making assumptions and filling in the details, and deciding when to ask for more advice. Work in this area includes McCarthy's proposal for an 'advice-taker', the TEIRESIAS program [DaL82], and the program FOO [MoH79].

At present, there is neither an organized source of knowledge about dominance relations nor an general expert in the derivation of dominance relations. Thus, learning by instruction is of little use in learning dominance relations although this assessment may change as more is known about dominance relations in the future.

In learning by deduction or deductive learning, the learner draws deductive, truth-preserving inferences from the knowledge given and stores useful conclusions. Learning by deduction includes knowledge reformulation, knowledge compilation, creation of macro-operators, caching, chunking, equivalence-preserving operationalization, and other truth-preserving transformations.

Learning by deduction includes aspects of automatic deduction or mechanical theorem proving [Nil80, WOL84]. An important, early advance in automatic deduction was the development of *resolution procedures* based upon the *resolution principle* [Rob65]. The resolution procedures constitute a logically complete method for proving theorems in first-order predicate calculus. However, the search space generated by the resolution method grows exponentially with the number of formulas used to describe a problem, so that problems of even moderate complexity cannot be solved in a reasonable time. Several domain-independent heuristics, e.g., *set of support* [WRC65], were proposed to deal with this issue, but they proved to be too weak to produce satisfactory results. There are too many possible inferences that can be drawn at any one time, and finding the relevant inferences can be impossible without domain-specific guidance. Furthermore, the efficiency of deductive reasoning is greatly affected by the way in which the body of knowledge is formalized.

Nonetheless there is an important class of problems that resist solution by any other method and must be solved by deduction. Deduction permits logically complex queries of a knowledge base containing generalizations even when we cannot evaluate a query directly. Recent work in deduction include work on common-sense reasoning [McD78, Doy79] and *nonmonotonic logic* [Bob80]. Nonmonotonic logic can retract a conclusion in the face of further evidence, so its set of inferable conclusions does not increase monotonically with the set of premises as in conventional deductive logics. While many procedures have been implemented that support this type of reasoning, their theoretical foundations are questionable.

Dominance relations can learned by deduction, in particular, by automated deduction. A dominance relation is easily expressed in logic as an implication statement. Dominance relations are true theorems. Thus, if a candidate relation which have the prescribed form can be proved to be logically true, then it must be a dominance relation for the problem under study. Automated deduction is very good at finding proofs. Section 3.6.1 discusses in further detail the use of automated deduction to find dominance relations.

In learning by analogy, the learning system recognizes analogies in a related performance task and transform relevant knowledge in the knowledge base of the related task into a form suitable for use by its performance element. The study of learning by analogy has been mainly on the use of analogy in automated problem solving [Eva68, Kli71, MoN73, BrT79, McD79, Win79, Car83, Bur86]. and the use of analogy to guide the the modification of programs [MaW75, DeM77, Ama86].

The close relationship between everyday notions of analogy and similarity has led to several models of analogical reasoning based on forms of partial mapping [Eva68, Win79]. These algorithms assume that the best partial match can be found by accumulating evidence for each of a number of possible object-to-object mappings between the two task and then choosing the one that scores the highest. In these systems, evidence for a match consist essentially of the number of relational connections preserved between the corresponding objects for a given alignment of objects. This approach presupposes that well-defined, bounded representational models of the tasks in both the *base* (that is, familiar) domain and the *target* domains are available as inputs. This presumption is often not true in learning situations, particularly if the new domain is totally unfamiliar. Matching cannot be used to construct such a representation where there was none before. Moreover, as the complexity of the descriptions increases, the conceptual representations for many real situations may contain many objects that do not take part in a specified analogy.

An alternative approach [Bur86, Car83] is based on analogical mapping. Such an approach uses a set of heuristics to delimit what is to be "imported" from a base to a target domain at a given time. One such heuristic involves mapping previously formed abstractions, such as underlying causal and planning rules. Focusing on such structures and their associated special cases and known problems allows for a much more top-down form of analogical reasoning.

Analogy is capable of learning new dominance relations. If the learning system has available to it a pool of problems with known dominance relations, it may be able to recognize analogies between the new problem domain and known problem domains. The common substructure between the two descriptions can then guide the transformation the dominance relations in the known domains into relations in the new domain. If the common substructure has been correctly identified and the transformation is sound, then the derived relation are likely to be dominance relations in the new domain. Section 3.6.2

describes learning dominance relations by analogy in greater detail.

Learning by induction involves the generalization of input information and selection of the most plausible or desirable result. Inductive learning can be subdivided into learning from examples and learning by observation and discovery. In learning from examples, the system is taught how to perform a task by presenting it with examples of how it should behave (positive instances) and examples of how it should not behave (negative instances). Examples can be viewed as pieces of very specific knowledge that cannot be used efficiently by the performance element. The system generalizes the examples to find higher level rules that can be applied to guide the performance element. The examples are provided by a source of information, which can be a teacher that knows the concept [And84, Buc78, DiM83, FHN72, Mic83, Min84, Mit82, Ver78, Win75] or the environment on which the student performs experiments and from which it receives feedback [Wat70, AnS79, Ohl83, Lan83, Ara84]. The latter case is called *learning by experimentation* (this includes *learning by doing* and *learning by problem solving*).

In learning by observation and discovery (also called *descriptive generalization*), one searches without help of a teacher, for regularities and general rules explaining all or at least most observations. This form of learning includes *conceptual clustering* (form object classes describable by simple concepts), constructing classifications, fitting equations to data, discovering laws explaining a set of observations [Lan81, LBS83], and formulating theories accounting for the behavior of a system [DaL82, Len82, Len83a, Len83b].

Simon and Lea [SiL74] describe the problem of inductive learning as the problem of using training instances, selected from some space of possible instances, to guide the search for general rules. They call the space of possible training instances the *instance space* and the space of possible general rules the *rule space*. An intelligent program may also select its own training instances by actively searching the instance space in order to resolve some ambiguity about the rules in the rule space. A learning system moves back and forth between an instance space and a rule space until it converges on the desired rule. When the training instances are in a form far removed from the form of the rules, special processes are needed to interpret the raw training instances. Similarly, when the program needs some new training instances, experiment-planning routines use the current high-level hypotheses to guide the search of the instance space.

One issue involving the instance space is the *quality* of the training instances. High-quality training instances are unambiguous and provide

reliable guidance to the search of the rule space. Low-quality training instances invite multiple, conflicting interpretations and only provide tentative guidance to the rule-space search.

Ambiguity also arises if the program must learn from *unclassified training instances*. In these so-called *unsupervised learning situations*, the program is given heuristic information that it must use to classify the training instances itself. If this heuristic knowledge is weak and imperfect, the rule-space search must treat the resulting classifications as being potentially incorrect.

Another factor affecting the quality of the training instances is the *order in which they are presented*. A good training sequence systematically varies the relevant features to determine which features are important.

With unambiguous training instances, the program can embody a whole set of constraining assumptions about the instances that permit rapid location of the appropriate rules. Ambiguous instances cause the program to consider a much larger space of hypotheses. If the training instances can contain errors, the program must consider that any given incorrect instance is due to either measurement error or classification error.

Another design issue is how to search the instance space. Most work in inductive learning have avoided this issue by assuming either that the instances are presented all at once or else that the program has no control over their selection. Programs that update their hypotheses as additional training instances are made available perform *incremental learning*. Programs that explicitly search the instance space perform *active instance selection*.

Most methods of searching the instance space use a set, H, of hypotheses in the rule space that are currently believed by the program to be the most plausible. One approach tries to *discriminate* as much as possible among the alternatives within H. A training instance can be chosen that "splits H in half", such that half of the hypotheses can be ruled out when the new instance is obtained. Another approach is to choose the most likely hypothesis in H and try to *confirm* it by checking additional training instances (particularly instances with extreme characteristics). This permits the learning system to determine the limits of applicability of the hypothesis under consideration. A third approach, called *expectation-based filtering*, selects training instances that *contradict* the hypotheses in H. The learning program can then focus its attention on those instances in which its current hypotheses break down.

The training instances may need to be transformed before they can be used to guide the search of the rule space. This transformation process can be quite difficult, especially in perceptual learning tasks [Win75].

that can be grouped together to form a schema. The SPARC system [Die80] uses schema instantiation to solve a learning problem that arises in the card game Eleusis.

Data-driven techniques generally support incremental learning. The version space method can easily modify its set H to account for new training instances without any backtracking by the learning program. In contrast, model-driven methods, which test and reject hypotheses based on an examination of the whole body of data, must either backtrack or search the rule space again, because the criteria by which hypotheses were originally tested (or schemas instantiated) have changed.

However, model-driven methods tend to have good noise immunity. Since the whole set of training instances is available, the program can use statistical measures of how well a proposed hypothesis accounts for the data. When a set of hypotheses, H, is tested against noisy training instances, the model-driven methods need not reject a hypothesis on the basis of one or two counterexamples. In data-driven methods, H is revised each time on the basis of the current training instance. A single erroneous instance can cause a large perturbation in H (from which it may never recover).

Learning systems often use the *closed-world assumption*, that is, the rule space contains the desired concept. The closed-world assumption allows programs to locate the desired concept by progressively excluding concepts that are known to be incorrect. In some learning problems, the program can assume that the desired rule or rules exist somewhere in the rule space. Consequently, the search has a well-defined goal. In many situations, however, there is no such guarantee, and the learning program must confront the possibility that its representation of the rule space is inadequate and should be expanded. This is called the problem of *new terms*. One approach to expanding the rule space is to add new terms to the representation. The problem of defining new terms is quite difficult to solve. It is fairly easy to incorporate operators that create new terms in the hypothesis-refinment operator approach. Both BACON and AM programs create new terms by combining and refining existing terms.

In *single-concept learning*, the learning element is presented with positive and negative instances of one concept. It finds some concept that effectively partitions the space of all instances into two regions: positive and negative. In *multiple-concept learning*, there are training instances of several concepts. For each concept description, there is a corresponding region in the instance space. Overlapping concept descriptions can lead to integration problems. Hence, it may be necessary to modify existing rules when adding a new rule or concept

The rule space is usually defined by specifying the kinds of operators and terms that can be used to represent a rule. The designer of a learning system seeks to choose a rule space that is easy to search and that contains the desired rule or rules. Two factors that influence the choice of a representation language for the rule space are the kinds of inference supported by the representation and the *single-representation trick*. The single-representation trick uses the same representation for both the rule space and the instance space. Training instances can then be viewed literally as highly specific pieces of acquired knowledge.

The four main methods that have been used to search the rule space differ primarily in how they refine, H, the set of the currently most plausible rules. The presentation of the training instances drive the search in *data-driven methods*, while a priori model guides the search in *model-driven methods*.

The first data-driven method is the *version-space method* (and several related techniques). This approach uses the single-representation trick to represent training instances as very specific points in the rule space. The set H is initialized to contain all hypotheses consistent with the first positive training instance. New training instances are examined one at a time and pattern-matched against H to determine whether the hypotheses in H should be generalized or specialized.

The second data-driven method does not use the single-representation trick but uses special procedures (or knowledge) to examine the set of training instances and decide how to refine, H, the current set of hypotheses. The program can be viewed as having a set of *hypothesis-refinement operators*. In each cycle, it uses the data to choose one of the operators and then applies it. This approach is used in Langley's BACON system [Lan81, LBS83], Lenat's AM system [DaL82], and Lenat's EURISKO system [Len83b].

The third approach is *model-driven generate and test*. This method repeatedly generates and test hypotheses from the rule space against the training instances. Model-based knowledge is used to constrain the hypothesis generator to generate only plausible hypotheses. Meta-DENDRAL [BuM78] and INDUCE 1.2 [DiM83] use this approach.

The final approach is *model-driven schema instantiation*. It uses a set of *rule schemas*, which are abstract skeletal rules, to provide general constraints on the form of plausible rules. The method attempts to instantiate these schemas from the current set of training instances. The instantiated schema that best fits the training instances is considered the most plausible rule. Schema instantiation methods are useful when a system has many constraints

to the knowledge base. Michalski's AQ11 [MiC80] is a system that learns multiple concepts.

In *single-step tasks*, one rule or a set of independent rules can be applied in one step to accomplish the performance tasks. In *multiple-step tasks*, several rules must be chained together into a sequence. The learning element must now consider possible interactions among rules when modifying the knowledge base. There is also the problem of credit assignment. In a single-step task, the application of the performance standard immediately determines whether or not the rule is correct. In a multiple-step task, feedback is not available until the problem is completely solved or the game is complete. Moreover, the program can only determine if the entire sequence of rules is good and bad. The overall performance standard must be converted into performance standards for each rule. Learning systems which deal with multiple-step tasks include Samuel's checkers player [Sam67], Waterman's poker player [Wat70], Sussman's HACKER planning systems [Sus75], and Mitchell's LEX system for symbolic integration [MUB83].

Learning by examples can be used to learn dominance relations. A weak problem solver can generate the complete search tree of any sample instance of the given problem. In a complete search tree, the values of the best solutions that can be derived from each node are always known. Thus, the nodes in the search tree can provide examples of how the dominance relations, if any, should and should not behave. However, in the absence of an oracle, examples of dominance relations cannot be identified among the nodes in the problem trees without knowledge of the relations. The learning system cannot expect to be presented with a set of examples by a source of information. It must perform experiments and use the feedback from the environment to determine the desired examples. Section 3.6.3 considers learning dominance relations by experimentation in greater detail.

3.6.1 Learning Dominance Relations by Deduction

In the prototypical automated deduction problem we have a set, S, of well-formed formulas (wffs) from which we wish to prove some goal wff, W. A resolution-based system produces proofs by contradiction. We negate the goal wff and add the negated goal wff to the set of axioms. The resolution-based system attempts to derive a contradiction represented by the empty clause NIL. Derivation of the empty clause proves the goal wff.

As examples, we apply automated deduction to candidate relations from a value-description of the 0/1-knapsack problem. For each node, u, in the search tree, let $D_1(u)$ represent the items included in the knapsack by the decisions which have already been made. The state variables are R(u), the cumulative value of the items in $D_1(u)$, and W(u), the cumulative weight of the items in $D_1(u)$. In addition, let L(u) be the level of node u in the search tree, and let B(u) be the value of the best solution contained in node u. Let the candidate relation and goal wff be

$$[=(L(x),L(y))] \& [>(R(x),R(y))] \& [<(W(x),W(y))] \rightarrow [>(B(x),B(y))]$$

where =(u,v), <(u,v), and >(u,v) are the predicates denoting the equal to, less than, and greater than relationships between u and v.

Negation of the goal wff produces the clauses:

C_1: $=(L(x_1),L(y_1))$

C_2: $>(R(x_2),R(y_2))$

C_3: $<(W(x_3),W(y_3))$

C_4: $\neg >(B(x_4),B(y_4))$

The set of axioms consists of clauses that describe pertinent aspects of problem solving and the problem description. First, we tell the automated deduction program that the value B(u) is the sum of G(u) and H(u), the value of the best solution of the remaining decisions.

C_5: $=(B(x_5),SUM(R(x_5),H(x_5))$

In a value-description, equilevel nodes have identical sets of undecided items, D. Thus, if $W(u) < W(v)$ for equilevel nodes u and v then for every solution derived from v which includes the subset $D' \subseteq D$ in the knapsack, there is a solution derived from u which includes the subset $D'' \subseteq D$ in the knapsack such that $D' \subseteq D''$. This implies that $H(x) > H(y)$ when $W(u) < W(v)$ and $L(u) = L(v)$.

C_6: $\neg =(L(x_6),L(y_6)) \vee \neg <(W(x_6),W(y_6)) \vee >(H(x_6),H(y_6))$

Furthermore, we include clauses that describe a pertinent property of the sum of two quantities.

C_7: $\neg >(t_1,t_2) \vee \neg >(t_3,t_4) \vee >(SUM(t_1,t_3),SUM(t_2,t_4))$

Finally, we tell the automated deduction program about a relationship between the functions >(u,v), =(u,v), and <(u,v).

C_8: $=(>(u,v),)no <(u,v) | \neg =(u,v))$

Figure 3.3 shows the resulting resolution tree for the goal wffs. The contradiction of the negated goal wff proves the goal wff and so the candidate relation is a dominance relation.

In contrast, automated deduction is unable is to derive the empty clause for the slightly different candidate relation:

$$=(L(x),L(y)) \& >(R(x),R(y)) \& >(W(x),W(y)) \rightarrow \geq(B(x),B(y))$$

Figure 3.4 shows the resolution tree for this relation. The automated deduction system concludes that either the candidate relation is not a dominance relation or we are ignorant of some step in its proof.

These examples illustrate the strengths, weaknesses, and limitations of automated deduction as a tool for finding dominance relations. The use of sound inference rules ensure that an automated deduction program only derives dominance relations which are true when the premises are true. This leads to two very desirable properties for the results from automated deduction: certainty and generality. If we ensure that the set of premises consists only clauses known to be definitely true, then any relation found by the automated deduction program is definitely a dominance relation. Moreover, as the axioms describe the general problem instance, these results are dominance relations for the general problem as opposed to just a subset of problem instances.

Automated deduction makes explicit all required deduction steps, and requires that all information about the problem be explicitly given to the program, including the implicit information that a person tacitly assumes. This can expose questionable or even incorrect assumptions. More importantly, this makes explicit the reasons why a particular relation is a dominance relation and the processes by which the conclusion is reached. Such knowledge is very valuable as it can lead to an understanding and formalization of the deep causal knowledge about the derivation of dominance relations.

On the other hand, being forced to supply such minute and mundane information can often be a nuisance as well as a serious handicap. The absence of a single necessary axiom will prevent the theorem prover from finding a proof of a true goal wff. This can seriously hinder the ability of automated deduction to find dominance relations in learning situations where there is little formalized knowledge about the domain to which the learning system is being applied.

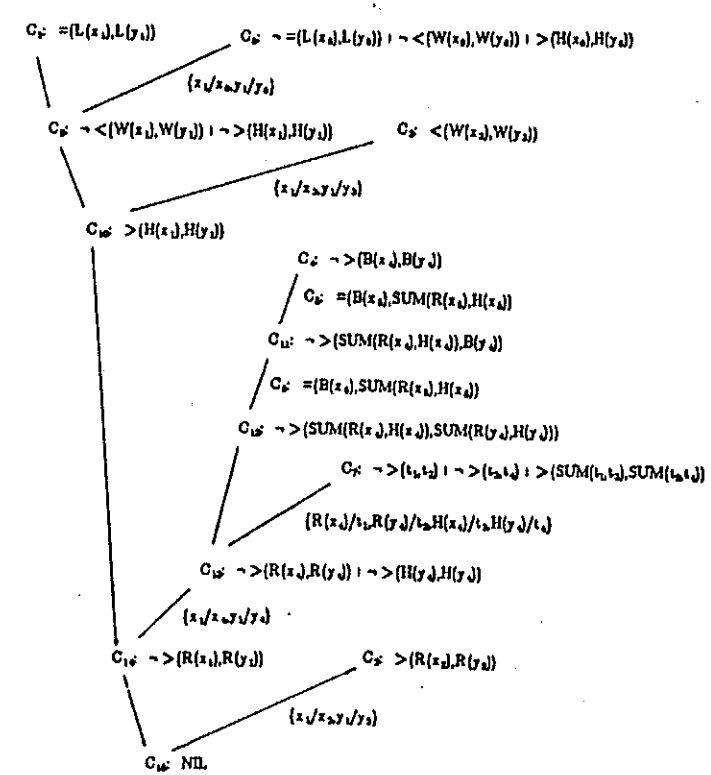


Figure 3.3 Resolution tree for $=(L(x),L(y)) \& >(R(x),R(y)) \& <(W(x),W(y)) \rightarrow >(B(x),B(y))$.

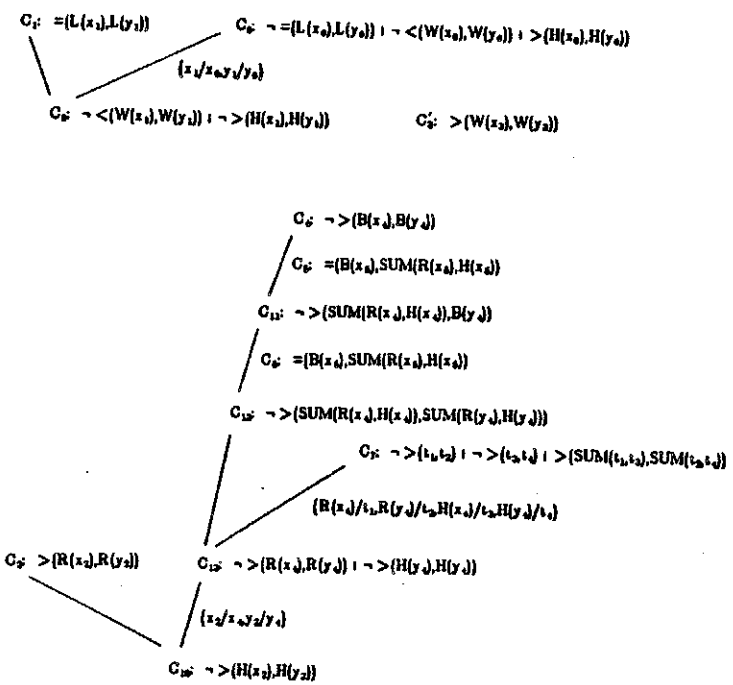


Figure 3.4 Resolution tree for =L(x),L(y)) & >(R(x),R(y)) & >(W(x),W(y)) → >=(B(x),B(y)).

An automated deduction system cannot modify the set of axioms supplied to it. It is necessary to restart the theorem-prover for each new candidate relation. A more serious handicap is that the automated deduction system is restricted to whatever knowledge the program designer has about dominance relation and the problem domain. There is very little causal knowledge about dominance relations, and knowledge about new problem domains may be limited.

It is clear that automated deduction is not sufficient by itself to tackle the problem of finding dominance relations. For one it needs an external mechanism to supply it with candidate relations. It also needs to be coupled with other mechanisms which are able to explore the environment of dominance relations and perhaps automatically modify the set of initial axioms. This should include the ability to innovate and come up with new symbols and not just be constrained to the set of symbols in the initial problem description. The known problem description is often optimized towards state-space methods of solving the problem and may obscure the presence of dominance relations.

The user of an automated deduction system must supply three types of information: the inference rules, the axioms representing the problem, and a strategy. The set of inference rule is well-defined and well-known. Thus, the issues that have to be addressed before dominance relations can be learned by deduction are the derivation of the axioms and the strategy. The axioms supplied by the user must include the explicit problem description as well as implicit knowledge about the problem domain and the search algorithm. Deriving the relevant implicit knowledge may be difficult as it often requires deep knowledge about a new, possibly ill-understood problem domain. The strategy is determined by the preferences assigned to the different axioms ad inference rules. These preferences are problem-dependent and their derivation is subject to the same difficulties as the derivation of the implicit domain knowledge. These facts include axioms representing the problem description, axioms representing implicit knowledge about the problem domain and general problem solving, and the preferences among the various axioms and the inference mechanisms.

3.0.2 Learning Dominance Relations by Analogy

An analogy system finds a known problem domain (base domain) similar to the new problem domain (target domain) and uses analogical mappings between the two domains to transform dominance relations in the base domain

into relations in the target domain. We can illustrate this process by two single machine scheduling problems: scheduling to minimize weighted completion time and scheduling to minimize cumulative lateness.

Scheduling to minimize the cumulative lateness involves a set of n tasks: $t_1, ..., t_n$. Each task, t_i, has a length, $l(t_i)$, and a deadline, $d(t_i)$. The goal in this problem is to find the one-processor schedule $\sigma()$ that minimizes $\sum_{i=1}^{n} |\sigma(t_i) + l(t_i) - d(t_i)|$.

Each task, $t_i' : i = 1, ..., n$, in scheduling to minimize weighted completion time has length, $l'(t_i')$, and weight, $w'(t_i')$. This problem seeks the one-processor schedule, $\sigma'()$, that minimizes $\sum_{i=1}^{n} |\sigma'(t_i') + l'(t_i')| \cdot w'(t_i')$.

Table 3.4 shows the identity-descriptions for these two problems. The decision variable, ρ_i, is the identity of the i-th task in the schedule $\sigma()$. There are two state variables: T(N) and L(N). Let s(N) be the set of assigned tasks in the partial schedule represented by node, N. T(N) is completion time of the most recently scheduled task in the partial schedule represented by node N while L(N) is the sum of the set of tasks in s(N). Hence, ρ'_i and T'(N) are identical to ρ_i and T(N). C'(N) is the sum of the weighted completion of the tasks in s(N).

Let scheduling to minimize cumulative lateness be the base domain and let scheduling to minimize weighted completion times be the target domain. The matching of the problem descriptions shown in Table 3.4 leads to the following analogical mapping:

$t_i \rightarrow t_i'$
$\rho_i \rightarrow rho'_i$
$T(N) \rightarrow T'(N)$
$L(N) \rightarrow C'(N)$
$l(t_i) \rightarrow l'(t_i')$
$d(t_i) \rightarrow w'(t_i')$

Scheduling to minimize cumulative lateness is known to have a greedy solution which always schedules the unscheduled task with the smallest lateness when there is only one task with this value of lateness. This greedy solution corresponds to the dominance relation, IF [SIB(x,y)] [L(x)<L(y)] THEN [B(x)≤B(y)], where B(x) is the value of the best solution derived from node x, and SIB(x,y) is a predicate which tests whether nodes x and y are siblings.

Table 3.4 Identity-descriptions for scheduling to minimize cumulative completion time and scheduling to minimize cumulative lateness.

	Scheduling to Minimize Cumulative Lateness	Scheduling to Minimize Weighted Completion Times
ENTITY	$t_1, ..., t_n$	$t'_1, ..., t'_n$
DECISION VARIABLE	$\rho_i = \{t_1, ..., t_n\}$ i=1,...,n	$\rho'_i = \{t'_1, ..., t'_n\}$ i=1,...,n
STATE VARIABLES	T(P) = T(N) + l(ρ) where P = apply(ρ,N) L(P) = L(N) - [T(P) + d(ρ)] where P = apply(ρ,N)	T'(P) = T'(N) + l'(ρ') where P = apply(ρ,N) C'(P) = C'(N) + [T'(P) * w'(ρ')] where P = apply(ρ,N)
ENTITY VARIABLES	l(t_i) d(t_i)	l'(t'_i) w'(t'_i)
PROBLEM CONSTANTS	n	n
OPTIMIZATION	minimize L(S)	minimize C'(S)
CONSTRAINT		

Application of the analogical mapping leads to the replacement of L(x) by C'(x) and the rule, IF [SIB(x,y)] [C'(x)<C'(y)] THEN [B(x)≤B(y)]. This rule is another way of stating the known greedy scheduling algorithm that minimizes weighed completion times when only one unscheduled task has the smallest weighted completion time.

Analogy can be a very powerful and efficient way of deriving dominance relations when there is a good match between the target domain and the base domain. The known dominance relation in the base domain provides an excellent starting point for the derivation of a new relation. Other learning methods start with much less knowledge. Moreover, the analogical mapping provides powerful heuristic knowledge that drastically reduces the amount of search during the symbolic transformation of the relations.

The most important factor affecting the quality of the process of learning by analogy is the quality of the match between the chosen base domain and the target domain. A poor match requires more "guessing" by the transformation routines and increases the likelihood that the derived rule is incorrect.

Even when there is a good match between the two domains, the rules derived by analogy are heuristics, which may or may not be dominance relations. We cannot be certain that the derived rules are definitely dominance relations because heuristic knowledge is used to find the "best" base domain and to transform the relations in the base domain. Testing of the derived rules with sample problem instance can increase our confidence in the derived rules but not total confidence as testing cannot test all possible problem instances.

The basic mechanism in analogy is the transformation of known relations in an analogous base domain. This mechanism limits the use of analogy to learn dominance relations. Analogy cannot derive rules which are unrelated to known dominance relations so it cannot derive dominance heuristics dissimilar to known dominance relations. Moreover, analogy does not work when there is no suitable base domain. Analogy is not useful in domains which are different from the domains with known dominance relations.

A system that learns dominance relations by analogy must have three types of information: a good set of domains with known dominance relations, routines to measure similarities between domains, and routines to transform relations in the base domain into relations in the target domain. At present, there are only a small number of problems with known dominance relations. However, this problem will be alleviated as more work is done on dominance relations. Both the issues of similarities between domains and transformations between domains will have to be addressed before analogy can be used to derive dominance relations.

3.6.3 Learning Dominance Relations by Experimentation

Learning by experimentation learns by performing experiments on the environment and using the results of these experiments to drive the learning process. This feedback from the environment allows the learning system to adapt its search for dominance relations to the domain under study. Thus, learning by experimentation is not constrained to domains and relations that are similar to known domains and known dominance relations even though it is likely to be more efficient and successful in familiar domains than in strange domains. This graceful degradation permits experimentation to be applied successfully to a much wider set of domains than the other learning mechanisms.

The task of learning dominance relations is a multiple-concept learning task as it is possible for a problem to have more than one dominance relation. To learn dominance relations by experimentation, the system must be able to generate sample problem instances, solve these sample problem instances, and induce possible dominance relations from instances in the search space. A weak problem solver can solve the problem instances. The other two topics are both open questions and must be solved before experimentation can learn dominance relations.

As indicated in Section 3.6, in the absence of an oracle, it is impossible to identify examples of dominance relations among the nodes of the search tree without explicit knowledge of the proposed relation. It is necessary to propose a relation before the system can find examples of this relation. This restriction rules out the use of data-driven methods. Model-driven schema instantiation is also inappropriate because there is no good schema for the dominance relations. Hence, a system that learns dominance relations by experimentation must use model-driven generate-and-test search the space of possible relations (rule space). Model-driven generate-and-test considers one candidate relation at a time, thereby reducing the task of learning dominance relations into a sequence of single-concept learning tasks.

Clearly, heuristic methods have to be used to generate sample problem instances as well as to find dominance relations in the sample search trees. Any relation derived by experimentation is guaranteed to be true only for the set of sample problem instances used in its derivation. The use of fallible heuristic knowledge to generate this set of sample problem instances implies

that experimentation can only find dominance heuristics. There will always be some doubts whether or not the relations found by experimentation are true dominance relations. Some other method of proving the correctness of the dominance heuristics will be necessary.

3.6.4 Combining the Different Learning Mechanisms

In the previous sections, we have seen that both analogy and deduction may not be useful for all problems. Analogy is only useful in problem domains similar to domains with known dominance relations. Deduction is only useful in well-known problems because of the need for the designer of the learning system to supply all information relevant to the derivation of the dominance relations. A learning system will have to work in an unfamiliar environment. Only experimentation works well in these situations. Thus, experimentation must be the primary mechanism that drives a system for learning dominance relations. Analogy and deduction are secondary mechanisms; they are necessary as they can be very effective in the proper situations.

Figure 3.5 shows the different learning mechanisms. Both analogy and experimentation derive dominance heuristics. Deduction derives dominance relations. Any relation proved by deduction to logically follow from the axioms describing the problem is a true dominance relation. Hence, one way of verifying the dominance heuristic is to use deduction to try to prove the dominance heuristics. This scheme also provides a possible solution to the problem of proposing suitable candidate relations for automated deduction.

The process of learning dominance relations by each learning mechanisms may be viewed as the execution of a plan by the learning mechanism. The efficiency of the learning mechanisms can be enhanced by augmenting their general plans with domain specific knowledge. One way is to use analogy on similar domains that possess good plans. Obviously, this aspect of the learning system has to wait until each of the learning mechanisms has been sufficiently well developed so that there is an adequate pool of plans. Not all the steps in a plan are useful so it will be necessary to devise mechanisms to remove those steps which are not useful before adding the plan to the pool of known plans.

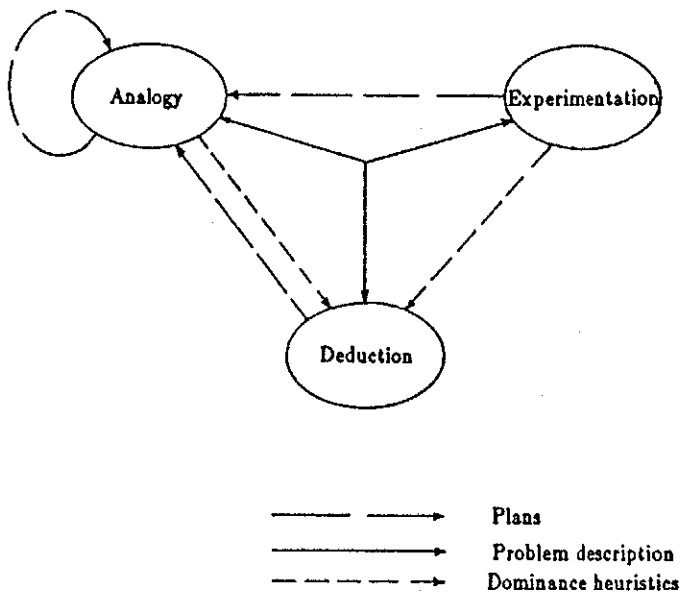


Figure 3.5 Interactions between the learning mechanisms

3.7 Summary

This chapter presents the methodology used to design a system for learning dominance relations as well as some results of applying this methodology. Among the results are the representations for the dominance relations and the number optimization problems, a classification of dominance relations, and the applicable learning mechanisms.

The dominance relation can be naturally expressed as either a production rule or the implication statement of predicate calculus. The similarities between the descriptions of different number optimization problems indicate the presence of a regular structure in the problem descriptions. This structure may be expressed in terms of a complete set of domain-independent elements. A suitable representation for the number optimization problems is the frame. Slots in the frame represent the set of elements. The slot fillers represent the values of these elements in the particular problem description.

Dominance relations may be classified by the permissible levels of the dominating and dominated nodes. This classification is able to expose those dominance relations which are either trivial relations or relations requiring excessive implementation overheads. These consideration suggest that the most useful types of dominance relations are those between nodes on the same level in the search tree, and so the study of learning dominance relations can be restricted to such dominance relations.

Dominance relations can be learned by analogy, deduction, and experimentation. Both analogy and deduction perform poorly in domains which are dissimilar to the domains used to derive their knowledge bases. Experimentation receives feedback from the environment, and can be effective in unfamiliar domains. Hence, experimentation is the primary learning mechanism, while analogy and deduction are secondary learning mechanisms. Nonetheless, both analogy and deduction are important. Analogy can be very effective in domains similar to known domains, while deduction is the only mechanism that guarantees the correctness of its learned relations.

CHAPTER 4
LEARNING DOMINANCE RELATIONS BY EXPERIMENTATION

4.1 Introduction

As indicated in the previous chapter, dominance relations may be learned by analogy, deduction and experimentation. The next step in our methodology is to refine the individual learning mechanisms by developing the procedures and knowledge that implement the learning mechanism.

Learning by deduction only learns relations that can be deduced from the facts known to the deduction program. It is most useful in domains where the relevant facts are known. Learning by analogy transforms a known dominance relation into a relation in the new problem domain. It is most useful when the new problem domain is similar to a domain with known dominance relations. Learning by experimentation is the most general mechanism as the feedback that it receives from its experiments on the environment permit it to learn relations in unfamiliar problem domains and unfamiliar types of relations. Thus, learning by experimentation is the primary learning mechanism for dominance relations.

In the work of this thesis, we have studied the refinement of learning dominance relations by experimentation. This study has led to the development of MUTT-1: a prototype program that is able to learn dominance relations by experimentation. MUTT-1 uses the model-driven generate-and-test paradigm because it is not possible to identify examples of the dominance relation without explicit knowledge of the dominance relation. Generate-and-test is a well-known general paradigm in artificial intelligence [Win84] and it has been applied to the task of learning cleavage rules for DENDRAL's mass-spectrometer simulator [BuM78] and to the task of learning structural descriptions [DiM83]. However, the key to a successful generate-and-test scheme is the domain-specific model which holds the domain knowledge. The models used for learning cleavage rules and structural descriptions cannot be used for learning dominance relations.

The design of MUTT-1 is described in Sections 4.2 to 4.7. Section 4.8 summarizes the results of applying MUTT-1 to several test problems.

4.2 MUTT-1

Figure 4.1 shows the organization of MUTT-1. The searcher implements the generate-and-test search of the space of candidate relations. A candidate relation is accepted as a dominance heuristic if the searcher determines that this relation behaves like a dominance relation during tests with sample search trees. The forester is responsible for the supply of appropriate sample search trees.

The searcher continues to test candidate relations until it either exhausts its resource allocation or finds a dominance heuristic; it then suspends itself and the scheduler takes over. The scheduler decides whether to stop or to continue searching for dominance heuristics. If the scheduler makes the latter decision, it will use the description manager to evaluate the effectiveness of the current description and if necessary make a description switch before restarting the searcher with a new resource allocation.

Details of the individual components will be found in the sections that follow this section. Section 4.3 describes the scheduler, Section 4.4 describes the description manager, and Section 4.5 describes the forester. The design of the searcher depends on the representation of the candidate relations so we will discuss the representation of candidate relations in Section 4.6 before describing the searcher in Section 4.7. These descriptions are illustrated with examples drawn mainly from the application of MUTT-1 to the 0/1-knapsack problem. Examples from other problems are used only when there is no appropriate examples for the 0/1-knapsack, and these cases will be explicitly noted in the descriptions.

4.3 Scheduler

The task of the scheduler of the scheduler is to coordinate the activities of the other components of the system. It incorporates the top-level control structures of the system.

One of its duties is to decide if there is sufficient justification for MUTT-1 to continue to search for dominance heuristics. One event that leads to the activation of the scheduler is the discovery of a dominance heuristic by the searcher. Whenever this event occurs, the scheduler tests whether the

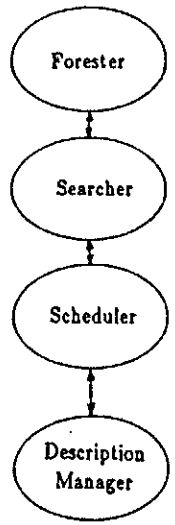


Figure 4.1 Organization of MUTT-1

dominance heuristic is a complete solution to the problem under study, that is, it corresponds to a greedy solution of the problem. An affirmative answer causes the scheduler to stop; otherwise the search for dominance heuristics continues.

When the scheduler is activated by the exhaustion of the searcher's resource allocation, the searcher makes its decision on the recent progress of the searcher. The scheduler measures the searcher's progress by τ, the amount of processing time consumed by the searcher since the most recent dominance heuristic was found or the start of the program if no dominance heuristic has yet been found. The scheduler also has a preset threshold, τ_0, which represents the maximum amount of processing that the user is willing to expend without finding a dominance heuristic. If τ is less than τ_0, the scheduler will decide to continue searching for dominance heuristics. If τ is greater than or equal to τ_0, the scheduler will query the user for guidance on whether to continue or stop. Values between 10 seconds and 100 seconds are suitable.

If the searcher decides to continue searching, it will allocate a new resource slice to the searcher after the description manager ensures that the current description is the correct. The size of the time slice determines the frequency with which the description manager evaluates the effectiveness of the current description. MUTT-1 uses fixed sized time slice of size κ. Let η be the number of working descriptions for the problem and let κ_0 be a preset value. Then, κ has the definition:

$$\kappa = \frac{\tau_0}{\eta \times \kappa_0} \tag{4.1}$$

κ_0 represents the average number of evaluations for each working description during a time period of duration τ_0. Two contradictory factors influence the choice of κ_0's value. κ_0 should be small to maximize the time actually spent on searching for dominance heuristics. However, κ_0 should be large to minimize the likelihood of wasting processing time on an inappropriate working description. Values between 5 and 10 are suitable for κ_0.

The time slice in MUTT-1 is a soft limit; the searcher relinquishes control at the earliest convenient moment after it exceeds the time limit. This simplifies the design and implementation of the time slice mechanism.

4.4 Description Manager

The ideal description manager is able to automatically derive the optimal working description from the problem statement. This capability requires deep causal knowledge, about problem descriptions and the problem domain, and is a major open research topic in artificial intelligence.

The design of MUTT-1 does not address this issue. MUTT-1 uses a a dumb description manager. The user is required to supply the appropriate working descriptions. Furthermore, the user must ensure that these working descriptions are correct and complete. The use of a dumb manager that relies totally on the user is acceptable for a prototype program used as knowledge test-bed, but not for a delivery system. Automated derivation of working descriptions is an issue that must be addressed during the refinement of this prototype program.

The user supplies a working description to MUTT-1 in the form of a w-frame. The w-frame is an extension of the description frame introduced in Section 3.4. It augments the basic problem description with information that assists MUTT-1 in manipulating the problem description. Figure 4.2 shows a stylized rendition of the frame that describes the value-description for the 0/1-knapsack problem. The Typical subslot defines typical values of each entity variable and each problem constant. It includes a LISP function that returns a typical value. These LISP functions may use one or more of the additional LISP functions stored in the LISP-Fn. slot of the w-frame.

The set of w-frames representing the working descriptions, $\{\psi_i : i = 1, \cdots, \eta\}$, are linked together into a description list which is ordered by increasing values of $\epsilon(\psi_i)$. $\epsilon(\psi_i)$ is a heuristic measure that estimates the average time required to find a dominance relation in description ψ_i. It is defined by the formula:

$$\epsilon(\psi_i) = \begin{cases} \dfrac{T(\psi_i)}{N(\psi_i)} & N(\psi_i) > 0 \\ T(\psi_i) & N(\psi_i) = 0 \end{cases} \tag{4.2}$$

where $T(\psi_i)$ is the total computation time consumed by the searcher while using working description ψ_i, and $N(\psi_i)$ is the number of dominance heuristic found by the searcher in working description ψ_i.

When the description manager is activated by the scheduler, it updates the ϵ-value of the current description and compares this ϵ-value with the ϵ-value of the next description in the description list. If the current description

Name: w1
Type: w-frame
Problem: 0/1-knapsack
Description-Type: value-representation
Entities:
  Name: u_i
  Number: n
Entity Variables:
  Entity Variable 1:
    Name: r()
    Value Type: integer
    Range: >0
    Number: n
    Typical:
      Value: [1,10]
      Code: (+ 1 (* (rnd) 9))
  Entity Variable 2:
    Name: w()
    Value Type: integer
    Range: >0
    Number: n
    Typical:
      Value: [1,10]
      Code: (+ 1 (* (rnd) 9))
Problem Constants:
  Problem Constant 1:
    Name: n
    Value Type: integer
    Range: >0
    Typical:
      Value: [2,10]
      Code: (+ 2 (* (rnd) 8))
  Problem Constant 2:
    Name: K
    Value Type: integer
    Range: >0
    Typical:
      Value: $[n, u_1 + \cdots + u_n]$
      Code: (lispfn0)
State Variables:
  State Variable 1:
    Name: R()
    Definition:
      Value: $R(N_r) = R(N_p) + x_1 * r(u_1)$
             $N_r = Apply(x_1, u_1, N_p)$
      Code: (lispfn1)
    Initial Value: 0
    Constraints: nil
    Value:
    Code:
  State Variable 2:
    Name: W()
    Definition:
      Value: $W(N_r) = W(N_p) + x_1 * w(u_1)$
             $N_r = Apply(x_1, u_1, N_p)$
      Code: (lispfn2)
    Initial Value: 0
    Constraints:
    Value: $W(N) \leq K$
    Code: (lispfn3)
Optimization:
  Expression: $R(N_{sols})$
  Extreme Value: maximum
  Code: (lispfn4)
Decision-Variable:
  Name: x_i
  Value: [0,1]
LISP Functions: lispfn0, lispfn1, lispfn2, lispfn3, lispfn4

Figure 4.2 Frame describing value-description of 0/1-knapsack problem.

has a smaller ϵ-value, then it is no longer the 'most effective' description and the description manager initiates a description switch. The first step in a description switch is to save the current state of the searcher into an appropriate file. The description manager then reorganizes the description list to maintain the ordering by ϵ-values. Finally, it restores the searcher to the state stored in the file corresponding to the new current description.

4.5 Forester

The forester supplies test trees to the searcher. The same test tree may be used to test many different relations; thus, the forester retains all previously-generated test trees in a tree pool. The forester always tries to satisfy a request from its tree pool. It will have to generate new trees if either the pool has been exhausted or the trees in the pool do not meet the searcher's demands. These new trees are added to the tree pool for use with future candidate relations.

The first step in generating a new tree is to generate a new problem instance. A new problem instance is generated by creating an instance of the p-frame (problem instance frame) and filling in the slots. The slots in the p-frame describe the structure of each problem instance for the problem under study and how to fill in these slots. The forester derives the p-frame from the w-frame by extracting the relevant sub-slots from the slots describing the problem constants and the entity variables in the w-frame. The p-frame specifies the manner in which values can be assigned to the slots of an instance of the p-frame. Figure 4.3(a) shows the p-frame for the 0/1-knapsack problem. Figure 4.3(b) shows an instance of the p-frame representing the 0/1-knapsack instance, Π_1.

Once a new problem instance has been generated, the forester uses a weak problem solver to solve the problem instance. Table 4.1 shows the search tree, r_1, generated by the solution of the problem instance, Π_1, shown in Figure 4.3(b). This search tree is stored in the Search-Tree slot of Π_1.

4.6 Representation of Candidate Relations

MUTT-1 represents candidate relations by c-rules which are production rules of the following form:

Name: p-frame
Type: concept
Problem: 0/1-knapsack
n:
  Value:
  Code: (+1 (* (ran) 10))
r:
  Value:
  Code: lispfn10
w:
  Value:
  Code: lispfn11
K:
  Value:
  Code: lispfn12
Search-Tree: nil
Lisp Functions: lispfn10, lispfn11, lispfn12

(a)

Name: Π_1
Type: p-frame
Problem: 0/1-knapsack
n: 5
r: {1 4 3 2 5}
w: {3 4 2 1 4}
K: 8
Search-Tree:

(b)

Figure 4.3 (a) The p-frame for the 0/1-knapsack problem, and (b) the p-frame instance for the 0/1-knapsack instance, Π_1.

Table 4.1 Search tree τ_1.

N_i	L(N)	R(N)	W(N)	B(N)	Sibling	Parent	Children
N_0	0	0	0	10	-	-	N_1,N_2
N_1	1	0	0	10	N_2	N_0	N_3,N_4
N_2	1	1	3	8	N_1	N_0	N_5,N_6
N_3	2	0	0	10	N_4	N_1	N_7,N_8
N_4	2	1	4	9	N_3	N_1	N_9,N_{10}
N_5	2	1	3	8	N_6	N_2	N_{11},N_{12}
N_6	2	2	7	7	N_5	N_2	N_{13}
N_7	3	0	0	7	N_8	N_3	N_{14},N_{15}
N_8	3	1	2	10	N_7	N_3	N_{16},N_{17}
N_9	3	1	4	9	N_{10}	N_4	N_{18},N_{19}
N_{10}	3	2	8	9	N_9	N_4	N_{20},N_{21}
N_{11}	3	2	3	8	N_{12}	N_5	N_{22},N_{23}
N_{12}	3	3	8	8	N_{11}	N_5	N_{24},N_{25}
N_{13}	3	3	7	7	-	N_6	N_{26},N_{27}
N_{14}	4	0	0	6	N_{15}	N_7	N_{28},N_{29}
N_{15}	4	1	7	7	N_{14}	N_7	N_{30},N_{31}
N_{16}	4	1	2	8	N_{17}	N_8	N_{32},N_{33}
N_{17}	4	2	6	10	N_{16}	N_8	N_{34},N_{35}
N_{18}	4	2	4	9	N_{19}	N_9	N_{36},N_{37}
N_{19}	4	3	8	9	N_{18}	N_9	N_{38}
N_{20}	4	3	7	8	N_{21}	N_{10}	N_{39}
N_{21}	4	4	9	8	N_{20}	N_{10}	N_{40},N_{41}
N_{22}	4	4	3	8	N_{23}	N_{11}	N_{42},N_{43}
N_{23}	4	5	8	8	N_{22}	N_{11}	N_{44},N_{45}
N_{24}	4	5	8	8	N_{25}	N_{12}	N_{46}
N_{25}	4	6	8	8	N_{24}	N_{12}	N_{47}
N_{26}	4	6	7	7	N_{27}	N_{13}	N_{48}
N_{27}	4	7	8	7	N_{26}	N_{13}	-
N_{28}	5	0	0	6	N_{29}	N_{14}	-
N_{29}	5	1	4	6	N_{28}	N_{14}	-
N_{30}	5	2	1	7	N_{31}	N_{15}	-
N_{31}	5	2	7	7	N_{30}	N_{15}	-
N_{32}	5	3	2	8	N_{33}	N_{16}	-
N_{33}	5	3	6	8	N_{32}	N_{16}	-
N_{34}	5	4	3	10	N_{35}	N_{17}	-
N_{35}	5	4	7	10	N_{34}	N_{17}	-
N_{36}	5	5	4	9	N_{37}	N_{18}	-
N_{37}	5	5	8	9	N_{36}	N_{18}	-
N_{38}	5	6	8	9	-	N_{19}	-
N_{39}	5	7	8	7	-	N_{20}	-
N_{40}	5	8	7	9	-	N_{21}	-
N_{41}	5	8	9	9	-	N_{21}	-
N_{42}	5	9	3	8	-	N_{22}	-
N_{43}	5	9	7	8	-	N_{22}	-
N_{44}	5	10	8	8	-	N_{23}	-
N_{45}	5	10	8	8	-	N_{23}	-
N_{46}	5	11	8	8	-	N_{24}	-
N_{47}	5	11	8	8	-	N_{25}	-
N_{48}	5	12	8	8	-	N_{26}	-

IF t-condition
d-condition$_1$
.
d-condition$_m$
THEN x dominates y

A candidate relation is a hypothesis that if two nodes satisfy the conditions described in the hypothesis then we can always identify the node with the better solution. This hypothesis can only be accepted if all nodes which satisfy the conditions in the hypothesis also agrees with its identification of the node with the better solution.

The left-hand side of the c-rule identifies the conditions which the nodes must satisfy in order for node x to dominate node y. It consists of a t-condition (type condition) and one or more d-conditions (dominance conditions). The t-condition identifies the type of the candidate relation represented by the rule, that is, the set of nodes to which this c-rule may be applied. Let L(x) be the level of node x and SIB(x,y) be a logical predicate which returns the value true if and only if x and y are siblings. The rule representing a candidate relation of type {all, same, sib} has [SIB(x,y)] as its t-condition. The term α-rule will be used to refer to to a c-rule of this form. A candidate relation of type {all, same, all} has a c-rule with [L(x)=L(y)] as its t-condition. The term β-rule denotes a c-rule of this form.

The d-conditions establish the conditions which nodes x and y must satisfy in order for x to dominate y. Each d-condition is a comparison between two symbols. Possible choices for the comparison operator are = (equal to), > (greater than), and < (less than). Possible symbols include state variables, entity variables, problem constants, functions of other symbols, and numbers.

The initial set of symbols for the 0/1-knapsack problem is {R(x), R(y), W(x), W(y), K, r(x), r(y), w(x), w(y)}. State variables have different values for different nodes, so it is necessary to use the symbol representing the state variable value of a particular node. Each state variable gives rise to two symbols, representing the values of the state variable for the two nodes involved in the dominance relation. Hence, the two state variables, R() and W(), contributes four symbols, R(x), R(y), W(x), and W(y), to the initial set of symbols. It is possible to include the entity variable in a node by associating a node with the entity involved in the decision leading to the generation of the node. This convention leads to the symbols, r(x), r(y), w(x), and w(y). Problem constants are independent of the nodes. There is no need to associate

them with the nodes, so problem constant K can be directly added to the list of symbols.

The right-hand side of each c-rule states that node x dominates node y. A c-rule casts this statement as a relationship between the values of the best solutions in the two nodes, ie. B(x) and B(y). By definition, B(x) is either smaller than or equal to B(y) for a minimization problem, that is, [B(x) $\leq$ B(y)].

A c-rule is an *acceptable rule* if it represents a dominance relation. Those c-rules which do not represent a dominance relation may be divided into *unacceptable rules* and *impossible rules*. An unacceptable rule is a c-rule whose set of conditions (left-hand side) cannot consistently identify the domiaating node and the dominated node. An impossible rule is an c-rule which is impossible, that is, no search tree contains a pair of nodes which matches its left hand side.

A node pair (N_i,N_j) is an *instance* of a c-rule if the substitutions, x/N_i and y/N_j, cause the left-hand side of the c-rule to be true. Instance (N_i,N_j) is a *positive instance* if B(N_i)$\geq$B(N_j) and a *negative instance* if B(N_i)<B(N_j). A positive instance supports the c-rule. A negative instance contradicts the c-rule.

Associated with each c-rule is a δ-value which reflects its suitability for use as a dominance relation. Initially, the δ-values of the c-rules are not known. The searcher finds the δ-values of c-rules by testing with search trees. If testing finds only positive instance of the c-rule, then the c-rule has a δ-value of *acceptable*. If testing finds one or more negative instances of the c-rule, then the c-rule has a δ-value of *unacceptable*. If testing fails to find any instance of the c-rule, then the c-rule has a δ-value of *impossible*.

Consider the c-rule, IF [SIB(x,y)] [R(x)>R(y)] THEN [B(x)$\geq$B(y)] (rule α_2' in Figure 4.4). (N_2,N_1) in search tree r_1 (Table 4.1) is an instance of this c-rule because N_2 and N_1 are siblings and R(N_2) is greater than R(N_1). However, B(N_2) is smaller than B(N_1), so (N_2,N_1) is a negative instance of α_2'. In contrast, B(N_8) exceeds B(N_7), and instance (N_8,N_7) is a positive instance of α_2'. The presence of both positive and negative instances of α_2's results in the assignment of a δ-value of unacceptable to α_2'. A c-rule with a δ-value of unacceptable has demonstrated that its left-hand side is unable to consistently identify the node with the better solution, and is an unacceptable c-rule.

Testing will not find any instances of the c-rule, IF [SIB(x,y)] [R(x)=R(y)] THEN [B(x)$\geq$B(y)] (c-rule α_4', and its δ-value will be impossible. A c-rule with

a δ-value of impossible is likely to be an impossible rule, but it may also be an acceptable rule or an unacceptable rule. The ambiguity in the actual nature of the rule comes from the use of a finite number of search trees during testing; it is impossible to test the c-rule against all possible search trees. Testing may have missed the search trees that contains positive and negative instances of the c-rule. An example of this will be the true c-rule, IF [L(x)=L(y)] [R(x)=R(y)] [W(x)=W(y)] THEN [B(x)≥B(y)] (rule β'_{13}). β'_{13} is an acceptable rule, but search tree τ_1 contains no instance of this rule.

An example of a c-rule with a δ-value of acceptable is the rule, IF [L(x)=L(y)] [R(x)>R(y) [W(x)<W(y)] THEN [B(x)≥B(y)] (rule β'_9). β'_9 is able to consistently identify the node with the better solution in the test set of search trees. Thus, it is most likely to be an acceptable c-rule, but may also be an unacceptable rule. As before, this ambiguity is due to the use of a limited sample of search trees during testing. It is possible that there are search trees containing negative instances but are not included in the sample set.

Learning by experimentation can only find the δ-values of the c-rules. It is unable to find the actual nature of the c-rule. The design of MUTT-1 assumes that an adequate sample set of test trees is sufficient justification for accepting the δ-value of a c-rule as the actual nature of the c-rule. Henceforth, we will use interchangeably the terms, acceptable rule and rule with a δ-value of acceptable. The terms, unacceptable (impossible) rule and rule with a δ-value of unacceptable (impossible), will be treated similarly.

4.6.1 Rule Space

The process of finding the δ-values of c-rules may be viewed as a search in the space of possible c-rules. One possible organization of this rule space is a simple directed graph. There are no loops in a simple directed graph so we can use the terminology commonly associated with trees such as parents, children, ancestors, and descendants. Figure 4.4 shows a small portion of the rule graph for the 0/1-knapsack problem. The root of this graph is the null rule, IF [true] THEN [B(x)≥B(y)], and is a dummy vertex inserted to complete the graph. All other vertices represent c-rules. Each directed edge represents the addition of a condition. The terminal vertex of each edge represents the c-rule formed by adding the condition represented by the edge to the c-rule represented by its initial vertex.

The children of a c-rule in this rule graph represent specializations of the c-rule and so there is inheritance of δ-values by descendant c-rules and ancestor

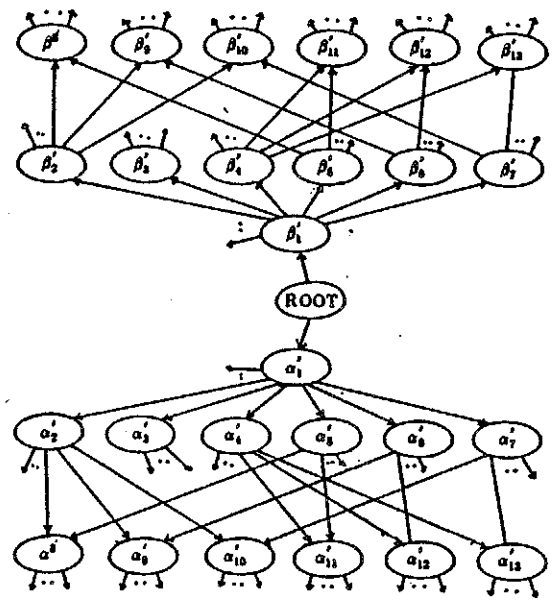


Figure 4.4 A fragment of the rule graph for the 0/1-knapsack problem.

c-rules. Section 4.6.2 discusses this topic in greater detail.

The rule graph contains both α-rules and β-rules. The α-rule is a c-rule with [SIB(x,y)] as its t-condition, and the β-rule is a c-rule with [L(x)=L(y)] as its t-condition. For each α-rule, there is a β-rule with an identical right-hand side and an identical set of d-conditions, and vice-versa. We refer to these two rules as parallel rules. The c-rules in Figure 4.4 have been labeled such that rule α_i' and rule β_i' are parallel rules. Parallel rules only differ in their t-conditions which implies a possible linkage of their δ-values. Section 4.6.3 explores parallelism in greater detail.

4.6.2 Value Inheritance

The parent of a c-rule represents a generalization of the c-rule. Dropping a d-condition from a c-rule transforms the c-rule into one of its parent rules. For instance, the elimination of condition [R(x)>R(y)] from the c-rule α_8' produces the c-rule α_3', while the elimination of condition [W(x)>W(y)] produces the c-rule α_5'. Both α_3' and α_5' are parents of α_8'.

Suppose that (N_i, N_j) is a positive instance of the α_8'. This assumption implies that the clause {[SIB(N_i,N_j)] & [R(N_i)>R(N_j)] & [W(N_i)>W(N_j)] & [B(N_i)≥B(N_j)]} is true. As a conjunction of predicates is true if and only if all the predicates are true, the clause formed by removing one or more conditions from the above clause will still be true. Removing the predicate [W(N_i)>W(N_j)] results in the true clause {[SIB(N_i,N_j)] & [R(N_i)>R(N_j)] & [B(N_i)≥B(N_j)]}. This new clause implies that (N_i, N_j) will also be a positive instance of α_3'. By a similar argument, we can show that (N_i, N_j) is also a positive instance of α_5'. This example illustrates the inheritance of instances by the parents of a c-rule. Every positive (negative) instance of a c-rule will be a positive (negative) instance of its parent c-rule.

By definition, there are no instances of an impossible rule. For this condition to be satisfied, the impossible rule cannot inherit any instance from its children, that is, none of its children can have any instances. Hence, all children of an impossible rule must also be impossible rules.

An acceptable rule has positive instances but no negative instance. The latter condition implies that none of its children can have negative instances. Each child of an acceptable rule is either another acceptable rule or an impossible rule. However, there must be at least one acceptable rule among the children of an acceptable rule. There will be no positive instances of a c-rule if none of its children has any positive instances.

There are negative instances of an unacceptable rule. Inheritance of instances implies that one or more of the children of the unacceptable rule have negative instances. There may also be positive instances of the unacceptable rule. Thus, it is possible for a child of an unacceptable rule to have no instances, only positive instances, only negative instances, or both positive and negative instances; so a child of an unacceptable rule can be another unacceptable rule, an acceptable rule, or an impossible rule.

4.6.3 Parallelism

Siblings are nodes on the same level with the same parent, and [SIB(N_i,N_j)] implies [L(N_i)=L(N_j)]. Parallel rules have the same set of d-conditions and right-hand side. A positive (negative) instance of an α-rule is also a positive (negative) instance of its parallel β-rule. The node pair (N_2,N_1) is a positive instance of the α-rule α_1', so the clause {[SIB(N_2,N_1)] [R(N_2)>R(N_1)] [B(N_2)>=B(N_1)]} is a true clause. Replacing [SIB(N_2),(N_1)] by its implied condition [L(N_2)=L(N_1)] yields {[L(N_2)=L(N_1)] [R(N_2)>R(N_1)] [B(N_2)>=B(N_1)]}. Hence, (N_2,N_1) is also a positive instance of the β-rule β_1'.

A β-rule is a c-rule between two equilevel nodes. Node pair (N_i, N_j) is a *sibling pair* if N_i and N_j are siblings and a *non-sibling pair* if N_i and N_j are not siblings. Siblings have a common parent, while non-sibling equilevel nodes do not. As a result, siblings exhibit greater correlation in the values of their state variables than do non-siblings. Hence, the absence of positive (negative) instances among the sibling pairs does not imply the absence of positive (negative) instances among the non-sibling pairs. For example, there are no instances of the rule IF [SIB(x,y)] [R(x)=R(y)] THEN [B(x)≥B(y)] (rule α_4') but there are positive and negative instances of the rule IF [L(x)=L(y)] [R(x)=R(y)] THEN [B(x)≥B(y)] (rule β_4'). The absence of instances of α_4' implies the absence of instances of β_4' among the sibling pairs. We know that there are instances of β_4', so there must be positive instances and negative instances of β_4' among the non-sibling pairs.

Nonetheless, it is still possible to deduce something about the possible instances of a c-rule from the instances of its parallel c-rule. Table 4.2 summarizes the relationships between the δ-values of parallel rules. Let α_i' and β_i' be a pair of parallel rules, where α_i' is an α-rule and β_i' is a β-rule. Suppose that α_i' is an unacceptable c-rule. Its negative instances are also negative instances of β_i'. Any c-rule with one or more negative instances in an unacceptable c-rule, so β_i' must be an unacceptable rule.

Suppose that α_i' is an acceptable c-rule. Then, there will be positive instances of α_i' and no negative instances of α_i'. The positive instances of α_i' are also positive instances of β_i'. However, the absence of negative instances of α_i' does not imply the absence of negative instances of β_i'. There may or may not be negative instances of β_i'. Hence, β_i' may either be an acceptable c-rule or an unacceptable c-rule.

Suppose that α_i is an impossible c-rule. There are no positive instances of α_i' and no negative instances of α_i'. Neither of this facts provide any useful information regarding the instances of β_i', and β_i' may be an acceptable rule, an unacceptable rule, or an impossible rule. Rule α_x has a δ-value of impossible, but its parallel rule, β_i is an unacceptable rule. In contrast, rule α_i is impossible, while rule α_i is acceptable.

4.7 Searcher

As indicated in Section 4.2, the searcher implements the generate-and-test scheme used by MUTT-1 to search the rule space. A generate-and-test scheme consists of two basic modules: a generator and a tester. The generator enumerates possible relations. The tester evaluates each proposed relation, either accepting it or rejecting it. There are two types of generators: the exhaustive generator and the informed generator. The simpler exhaustive generator generates all possible rules while an informed generator uses possibility-limiting information to restrict the relations which it proposes.

A c-rule has a unique set of symbol pairs if each of the d-conditions compares two different symbols and no two d-conditions compares the same pair of symbols. Consider a c-rule with m d-conditions. Suppose that the symbols in the d-conditions are chosen from a pool of n different symbols. The number of possible symbol pairs is $C_2^n = n(n-1)/2$. There are m d-conditions in the c-rule so there are $C_m^{n(n-1)/2}$ ways of choosing the m symbol pairs for the c-rule. Each set of m symbol pairs corresponds to 3^m possible rules because there are three possible choices for the comparison operator. Hence, the number of possible c-rules, n_c, is

$$n_c = C_m^{n(n-1)/2} \times 3^m$$

$$= \frac{[n(n-1)/2]!}{(n(n-1)/2-m)!\ m!}\ 3^m$$

Table 4.2 δ-values of parallel rules.

δ-value of α-rule	δ-value of parallel β-rule
acceptable	acceptable or unacceptable
impossible	acceptable or impossible or unacceptable
unacceptable	unacceptable

(a) Parallel β-rule.

δ-value of β-rule	δ-value of parallel α-rule
acceptable	acceptable or impossible
impossible	impossible
unacceptable	acceptable or impossible or unacceptable

(b) Parallel α-rule.

$$= \left\lfloor \prod_{i=0}^{m-1} \frac{n^2-n-i}{2(m-i)} \right\rfloor 3^m \qquad (4.3)$$

$= O(m^n)$ since $n > m$

Table 4.3 shows the results of evaluating Equation (4.3) for several values of m and n. The number of possible c-rules increases rapidly as n increases and is very large even for relatively small values of m and n. These results emphasize the need for an informed generator when using generate-and-test to search the space of possible c-rules.

The searcher (Figure 4.5) uses a hierarchical approach to the problem of limiting the number of proposed relations: the cartographer identifies a 'promising' region in rule space, and the generate-test-scrounge process explores the individual c-rules in this region. The cartographer uses a *map* to describe the 'promising' region which it has identified. The derivator and assayer help the cartographer generate the maps.

The generate-test-scrounge process is a modification of the basic generate-and-test scheme. The generator proposes possible c-rules (in the current map) and the tester evaluates the proposed rule. There is also a scrounger which deduces the implications of the tester's decision for other c-rules; thereby permitting decisions to be made on more than one c-rule in a generate-test-scrounge cycle.

The searcher employs a simple control strategy. The generate-test-scrounge process proposes and evaluates c-rules until it has completely explored the current maps. It then activates the cartographer. The cartographer updates the maps by adding new c-rules to the maps. Upon completion of the update, the cartographer deactivates itself, and the generate-test-scrounge process proceeds to evaluate the new c-rules.

As indicated in Section 4.2, the searcher works under a time limit imposed by the scheduler. The searcher checks for violation of this limit at specific instances during its execution, that is, whenever any component of the searcher deactivates itself.

Sections 4.7.1 to 4.7.6 describe in detail the individual components of the searcher.

Table 4.3 Results of evaluating n_c for several values of m and n.

	n_c				
m	n=4	n=5	n=6	n=7	n=8
1	18	30	45	63	84
2	135	405	945	1890	3402
3	540	3240	12285	35910	88452
4	1215	17010	110565	484785	1658475
5	1458	61236	729729	4944807	23882040
6	729	153000	3648645	39558456	274643460

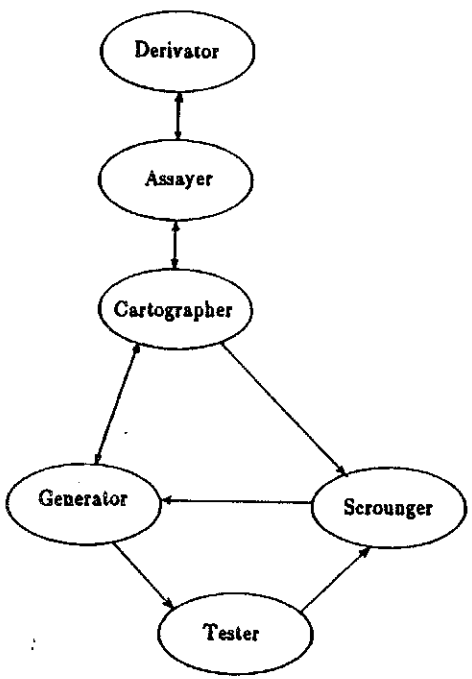


Figure 4.5 Organization of the searcher

4.7.1 Cartographer

The role of the cartographer is to find regions in the α-graph (β-graph) which are likely to contain acceptable c-rules and to produce an α-map (β-map) that describes these regions to the generate-test-scrounge process. The cartographer operates in an incremental fashion: each time the cartographer is activated, it updates the α-map (β-map) by adding new 'promising' c-rules to the α-map (β-map)

The cartographer's concept of a 'promising' c-rule is based upon its belief that an acceptable c-rule must be a semantically sensible construct, that is, its d-conditions are semantically sensible comparisons between semantically sensible symbols. In the current design of MUTT-1, the cartographer delegates much of the responsibility of ensuring semantic sensibility to the derivator and the assayer. When the cartographer becomes active, it uses the assayer to find new d-conditions. The assayer, in turn, uses the derivator to find new variables. The derivator contains heuristic rules that discard nonsensical symbols, and the assayer contains heuristic rules that generate sensible d-conditions from the symbols acceptable to the derivator.

If the assayer cannot find any new d-conditions, the cartographer causes MUTT-1 to stop; otherwise, it adds the operators associated with these new d-conditions to its pool of operators. Operators in the maps represent the addition of a d-condition to the c-rule represented by the node to which the operator is applied. The cartographer then adds the links and vertices resulting from these new operators to the maps.

Not all the c-rules that can be formed from the new operators will be included by the cartographer in the maps. Value-inheritance permits the cartographer to exclude all descendants of any c-rule known to have a δ-value of either impossible or acceptable. Every descendant of an impossible c-rule is an impossible c-rule and can never lead to a dominance heuristic. Every descendant of an acceptable c-rule has a δ-value of either acceptable or impossible but the descendant of a c-rule is a specialization of the c-rule; hence, descendants of an acceptable c-rule can never lead to any dominance heuristic better than that represented by the c-rule.

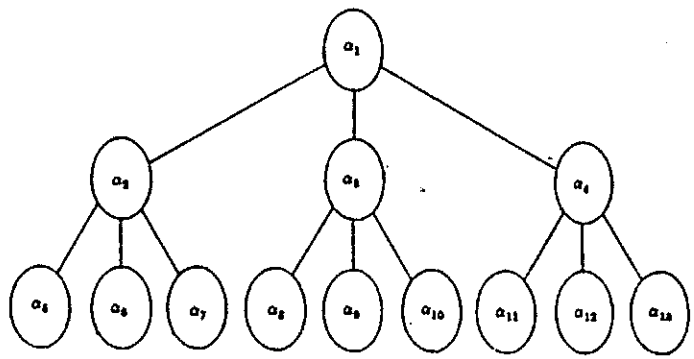
A dominance relation is a statement about the best solutions in the two nodes that satisfies its conditions. The value of the best solution in a node u is the sum of its current return, R(u), and the best possible return from its remaining decisions. Thus, an acceptable rule has to contain statements about the relationship between the current returns of node x and node y as well as the relationship between the future returns of the two nodes. Instances of a

state variable in siblings show greater correlation than instances of a state variable in non-sibling equilevel nodes. Thus, β-rules must express the relationship between the current returns explicitly as a d-condition between R(x) and R(y), while it may be possible for α-rules to express the relationship implicitly by d-conditions between other state variables. However, we can always add the corresponding d-condition between R(x) and R(y) without altering the meaning of the set of d-conditions. These arguments suggest that it is sufficient to restrict the maps to those c-rules which contain a d-condition between R(x) and R(y). A simple way of ensuring that all c-rules in the graph contain a d-condition between R(x) and R(y) is to only apply operators containing R(x) and R(y) to the root of the map.

Even with the above measures, it is easy for the numbers of possible c-rules in the maps to become unmanageably large. To prevent this from happening, the MUTT-1's cartographer imposes a preset limit on the number of d-conditions which a c-rule in the maps can have.

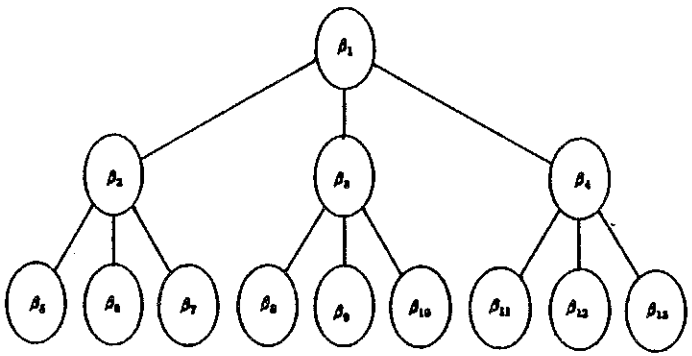
When the cartographer is activated for the first time in a value-description of the 0/1-knapsack problem, the initial α-map G_α^0 consists of the c-rule, α_0: IF |SIB(x,y)| THEN |B(x) ≥ B(y)|, and the initial β-map G_β^0 consists of the c-rule, β_0: IF |L(x) = L(y)| THEN |B(x) ≥ B(y)|. Suppose that the assayer returns the set of d-conditions: { |R(x) < R(y)|, |R(x) = R(y)|, |R(x) > R(y)|, |W(x) < W(y)|, |W(x) = W(y)|, |W(x) > W(y)| }. Only the operators involving d-conditions between R(x) and R(y) are applied to α_0 and β_0. Both α_0 and β_0 have δ-values of unacceptable, so no nodes are eliminated by value-inheritance of δ-values. The resulting maps, G_α^1 and G_β^1, are shown in Figures 4.6 and 4.7.

When the generate-test-scrounge process completes its exploration of G_α^1 and G_β^1, the cartographer sets to work on updating the two maps. It will initiate the derivation of d-conditions. Suppose that this results in the assayer augmenting the set of d-conditions with |KΔW(x) < RW(x)|, |KΔW(x) = RW(x)|, |KΔW(x) > RW(x)|, |KΔW(y) < RW(y)|, |KΔW(y) = RW(y)|, and |KΔW(y) > RW(y)|, where KΔW() and RW() are new state variables derived by the derivator. KΔW() is the difference between the knapsack capacity K and the state variable W(). RW() is the sum of the weights of the objects for which decisions have yet to be made. Suppose that the d-limit is three. Figure 4.8 shows the new α-map G_α^2, and Figure 4.9 shows the new β-map G_β^2s. G_α^2 differs from G_β^2 due to the different δ-values of parallel rules in G_α^1 and G_β^1.



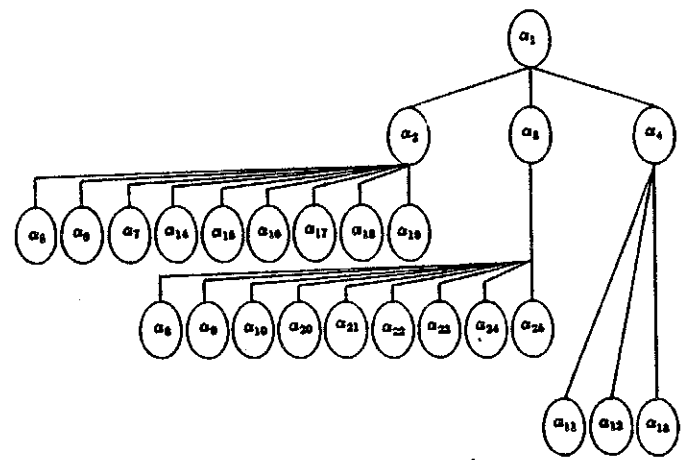
α_1 IF |SIB(x,y)| THEN |B(x)≥B(y)|
α_2 IF |SIB(x,y)||R(x)>R(y)| THEN |B(x)≥B(y)|
α_3 IF |SIB(x,y)||R(x)<R(y)| THEN |B(x)≥B(y)|
α_4 IF |SIB(x,y)||R(x)=R(y)| THEN |B(x)≥B(y)|
α_5 IF |SIB(x,y)||R(x)>R(y)||W(x)>W(y)| THEN |B(x)≥B(y)|
α_6 IF |SIB(x,y)||R(x)>R(y)||W(x)<W(y)| THEN |B(x)≥B(y)|
α_7 IF |SIB(x,y)||R(x)>R(y)||W(x)=W(y)| THEN |B(x)≥B(y)|
α_8 IF |SIB(x,y)||R(x)<R(y)||W(x)>W(y)| THEN |B(x)≥B(y)|
α_9 IF |SIB(x,y)||R(x)<R(y)||W(x)<W(y)| THEN |B(x)≥B(y)|
α_{10} IF |SIB(x,y)||R(x)<R(y)||W(x)=W(y)| THEN |B(x)≥B(y)|
α_{11} IF |SIB(x,y)||R(x)=R(y)||W(x)>W(y)| THEN |B(x)≥B(y)|
α_{12} IF |SIB(x,y)||R(x)=R(y)||W(x)<W(y)| THEN |B(x)≥B(y)|
α_{13} IF |SIB(x,y)||R(x)=R(y)||W(x)=W(y)| THEN |B(x)≥B(y)|

Figure 4.8 G_α^1



β_1 IF ‖L(x)=L(y)‖ THEN ‖B(x)≥B(y)‖
β_2 IF ‖L(x)=L(y)‖‖R(x)>R(y)‖ THEN ‖B(x)≥B(y)‖
β_3 IF ‖L(x)=L(y)‖‖R(x)<R(y)‖ THEN ‖B(x)≥B(y)‖
β_4 IF ‖L(x)=L(y)‖‖R(x)=R(y)‖ THEN ‖B(x)≥B(y)‖
β_5 IF ‖L(x)=L(y)‖‖R(x)>R(y)‖‖W(x)>W(y)‖ THEN ‖B(x)≥B(y)‖
β_6 IF ‖L(x)=L(y)‖‖R(x)>R(y)‖‖W(x)<W(y)‖ THEN ‖B(x)≥B(y)‖
β_7 IF ‖L(x)=L(y)‖‖R(x)>R(y)‖‖W(x)=W(y)‖ THEN ‖B(x)≥B(y)‖
β_8 IF ‖L(x)=L(y)‖‖R(x)<R(y)‖‖W(x)>W(y)‖ THEN ‖B(x)≥B(y)‖
β_9 IF ‖L(x)=L(y)‖‖R(x)<R(y)‖‖W(x)<W(y)‖ THEN ‖B(x)≥B(y)‖
β_{10} IF ‖L(x)=L(y)‖‖R(x)<R(y)‖‖W(x)=W(y)‖ THEN ‖B(x)≥B(y)‖
β_{11} IF ‖L(x)=L(y)‖‖R(x)=R(y)‖‖W(x)>W(y)‖ THEN ‖B(x)≥B(y)‖
β_{12} IF ‖L(x)=L(y)‖‖R(x)=R(y)‖‖W(x)<W(y)‖ THEN ‖B(x)≥B(y)‖
β_{13} IF ‖L(x)=L(y)‖‖R(x)=R(y)‖‖W(x)=W(y)‖ THEN ‖B(x)≥B(y)‖

Figure 4.7 G_β^1



α_1 IF ‖SIB(x,y)‖ THEN ‖B(x)≥B(y)‖
α_2 IF ‖SIB(x,y)‖‖P(x)=P(y)‖ THEN ‖B(x)≥B(y)‖
α_3 IF ‖SIB(x,y)‖‖P(x)>P(y)‖ THEN ‖B(x)≥B(y)‖
α_4 IF ‖SIB(x,y)‖‖P(x)<P(y)‖ THEN ‖B(x)≥B(y)‖
α_5 IF ‖SIB(x,y)‖‖P(x)=P(y)‖‖W(x)=W(y)‖ THEN ‖B(x)≥B(y)‖
α_6 IF ‖SIB(x,y)‖‖P(x)=P(y)‖‖W(x)<W(y)‖ THEN ‖B(x)≥B(y)‖
α_7 IF ‖SIB(x,y)‖‖P(x)=P(y)‖‖W(x)=W(y)‖ THEN ‖B(x)≥B(y)‖
α_8 IF ‖SIB(x,y)‖‖P(x)<P(y)‖‖W(x)>W(y)‖ THEN ‖B(x)≥B(y)‖
α_9 IF ‖SIB(x,y)‖‖P(x)<P(y)‖‖W(x)<W(y)‖ THEN ‖B(x)≥B(y)‖
α_{10} IF ‖SIB(x,y)‖‖P(x)<P(y)‖‖W(x)=W(y)‖ THEN ‖B(x)≥B(y)‖
α_{11} IF ‖SIB(x,y)‖‖P(x)=P(y)‖‖W(x)>W(y)‖ THEN ‖B(x)≥B(y)‖
α_{12} IF ‖SIB(x,y)‖‖P(x)=P(y)‖‖W(x)<W(y)‖ THEN ‖B(x)≥B(y)‖
α_{13} IF ‖SIB(x,y)‖‖P(x)=P(y)‖‖W(x)=W(y)‖ THEN ‖B(x)≥B(y)‖
α_{14} IF ‖SIB(x,y)‖‖P(x)>P(y)‖‖K△W(x)>K'W(x)‖ THEN ‖B(x)≥B(y)‖
α_{15} IF ‖SIB(x,y)‖‖P(x)>P(y)‖‖K△W(x)<K'W(x)‖ THEN ‖B(x)≥B(y)‖
α_{16} IF ‖SIB(x,y)‖‖P(x)>P(y)‖‖K△W(x)=K'W(x)‖ THEN ‖B(x)≥B(y)‖
α_{17} IF ‖SIB(x,y)‖‖P(x)<P(y)‖‖K△W(x)>K'W(y)‖ THEN ‖B(x)≥B(y)‖
α_{18} IF ‖SIB(x,y)‖‖P(x)<P(y)‖‖K△W(x)<K'W(y)‖ THEN ‖B(x)≥B(y)‖
α_{19} IF ‖SIB(x,y)‖‖P(x)>P(y)‖‖K△W(x)=K'W(y)‖ THEN ‖B(x)≥B(y)‖
α_{20} IF ‖SIB(x,y)‖‖P(x)<P(y)‖‖K△W(x)>K'W(x)‖ THEN ‖B(x)≥B(y)‖
α_{21} IF ‖SIB(x,y)‖‖P(x)<P(y)‖‖K△W(x)<K'W(x)‖ THEN ‖B(x)≥B(y)‖
α_{22} IF ‖SIB(x,y)‖‖P(x)<P(y)‖‖K△W(x)=K'W(x)‖ THEN ‖B(x)≥B(y)‖
α_{23} IF ‖SIB(x,y)‖‖P(x)=P(y)‖‖K△W(x)>K'W(y)‖ THEN ‖B(x)≥B(y)‖
α_{24} IF ‖SIB(x,y)‖‖P(x)=P(y)‖‖K△W(x)<K'W(y)‖ THEN ‖B(x)≥B(y)‖
α_{25} IF ‖SIB(x,y)‖‖P(x)=P(y)‖‖K△W(x)=K'W(y)‖ THEN ‖B(x)≥B(y)‖

Figure 4.8 G_α^2

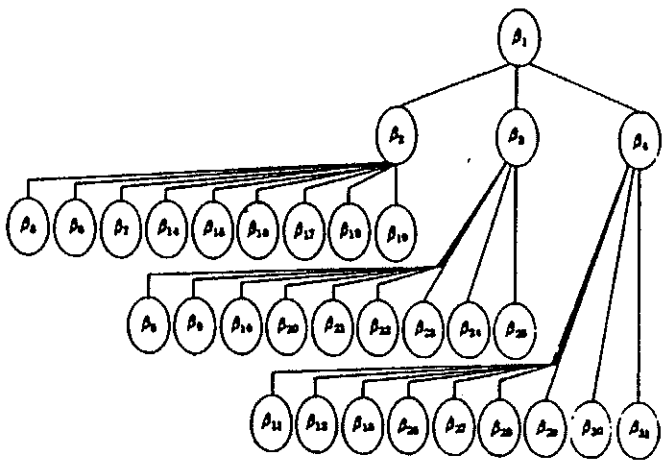


Figure 4.9 G_3^2

4.7.2 Assayer

A symbol is either a variable symbol or a constant symbol. Variable symbols may have different values for different nodes; constant symbols have the same value for all nodes. Instances of state variables are always variable symbols, while problem constants are always constant symbols. The entity variables can be introduced into the d-conditions by associating each node with the entity variables of the entity involved in the decision leading to the node. Different nodes may be associated with different entities in an identity-description. However, all equilevel nodes are associated with the same entity in a value-description. Thus, instances of entity variables are variable symbols in an identity-description but are constant symbols in a value-description.

A d-condition between two constant symbols is a trivial d-condition, and the assayer never generates such d-conditions. Let C be a constant symbol. Also, let $V_i(x)$, $V_i(y)$, $V_j(x)$, and $V_j(y)$ be four variable symbols, where x represents the dominating node and y the dominated node. There are be four types of non-trivial d-conditions:

T_1: $[V_i(x) \ \# \ V_i(y)]$
T_2: $[V_i(x) \ \# \ V_j(x)]$
T_3: $[V_i(x) \ \# \ V_j(y)]$
T_4: $[V_i(x) \ \# \ C]$

A d-condition of type T_1 is a condition between two instances of the same state variable and is always semantically sensible if the variable $V_i()$ is semantically sensible; thus, the assayer always generates d-conditions of type T_1 from the set of variables supplied by the derivator.

In the 0/1-knapsack problem, both R() and W() are semantically sensible variables because they have semantic meaning. R() is the return from the items included in the knapsack, and W() is the weight of the item in the knapsack. Among the possibilities considered by the derivator when defining new state variables are combinations between W() and K. One possible new state variable is $K\Delta W()$, the difference between K and W(). This state variable is semantically sensible; it is the capacity remaining in the knapsack. In contrast, the variable $K\Sigma W()$, the sum of K and W(), has no semantic meaning.

In contrast, d-conditions of type T_2 and T_3 compare instances of different variables while d-conditions of type T_4 compare an instance of a variable with a constant symbol. The assayer generates a condition belonging to one of these three types only when it is able to deduce from the working description that

the symbols in the d-conditions are semantically related. By definition, two symbols are semantically related if they are connected either explicitly or implicitly by a construct in the working description.

A state variable is related to a constant if the constant is an associated problem constant or a symbolic constant within the range of the state variable. In the 0/1-knapsack problem, state variable W() is semantically related to the problem constant K as K is associated with R() by to the constraint $W(N) \leq K$.

Two state variables are related if they share a common construct in the working description, if their definitions share a common symbol, or if they are related to a common variable. One new state variable generated by the derivator for the 0/1-knapsack problem is RW(), the sum of the weights of the unassigned items. It definition is the formula, $RW(N_r) = RW(N_p) - w(\theta)$, where N_r is the node in the search tree generated by the application of operator θ to node N_p. RW() is semantically related by W() as the entity variable w() appears in both their definitions. Recall that, the definition of W() is $W(N_r) = W(N_p) + w(\theta)$. Another new state variable derived by the derivator for the 0/1-knapsack problem is KΔW(), the capacity remaining in the knapsack. This state variable is defined by the formula, $K\Delta W(N) = K - W(N)$. RW() is semantically related to KΔW() since both of them are related to W().

Whenever the assayer in MUTT-1 becomes active, its first action is to call upon the derivator to find new symbols. There is a global symbol list which consists of two sublists: the a-list and the b-list. The a-list holds the variables and constants generated by the derivator during its most recent activation, while the b-list holds the variables and constants generated by the derivator prior to its most recent activation. The cartographer then looks for sensible d-conditions using the rules outlined earlier in this section. It looks for d-conditions between two symbols from the a-list and d-conditions between one symbol from the a-list and one symbol from the b-list. Useful d-conditions between two symbols from the b-list have already been considered by previous activations of the assayer. To facilitate future design changes in the assayer, the list of d-conditions derived by the assayer is also partitioned into two sublists: a u-list and a v-list. The u-list holds the d-conditions derived during the most recent activation of the assayer, while the v-list holds those derived during previous activations.

When the assayer becomes active for the first time in a value-description of the 0/1-knapsack problem, both the a-list and b-list are empty. The derivator will put the symbols r(), w(), R(), W(), and K in the a-list. The

assayer considers r(), w(), and K to be constant symbols, so it does not generate any d-conditions between them. Moreover, r() and w() do not appear in any constraint with either R() or W(), so no d-conditions between them will be generated. K, the knapsack capacity, and W() appear in a constraint. but the assayer ignores this constraint as it is an implicit condition satisfied by all feasible nodes in a problem tree. State variables R() and W() do not appear together in any constraint. The assayer will only generate the d-conditions: $[R(x) < R(y)]$, $[R(x) = R(y)]$, $[R(x) > R(y)]$, $[W(x) < W(y)]$, $[W(x) = W(y)]$, and $[W(x) > W(y)]$,

During the second activation of the assayer, the set of symbols known to the derivator has been augmented by the entity variable, rΩw(), and the state variables, RR(), RW(), and KΔW(). rΩw() is the ratio of r() to w(). RR() is the sum of the return of the unassigned items. The assayer will not generate any d-condition involving rΩw() as it is not related to any variable symbol. As indicated previously, W(), RW(), and KΔW() are all semantically related. RW() and KΔW() are forward variables while W() is a backward variable, so the assayer uses RW() and KΔW() instead of RW() and W() and augments the set of 'sensible' d-conditions with $[RW(x) < K\Delta W(x)]$, $[RW(x) = K\Delta W(x)]$, and $[RW(x) > K\Delta W(x)]$. Similarly, RR() is related to R() as both state variables have r() in their definitions. However, the assayer does not generate any d-conditions between the two variables as R() is a backward variable while RR() is a forward variable. Furthermore, all nodes on the same level have identical values for RR() and RW() and so the assayer does not generate d-conditions between RR() and RW().

4.7.3 Derivator

The derivator derives new state variables and entity variables in an incremental fashion. Each time the derivator is activated, it generates new variables by combining existing variables and constants. It includes these new variables in its set of known variables and constants. These new variables will participate in the derivation of other variables during the derivator's next activation.

New entity variables are derived by the formula

$$E_0 = E_1 \# E_1 \tag{4.4}$$

where E_0 is the new entity variable, E_1 and E_2 are two existing entity variables, and # is one of the arithmetic operators: + (plus), − (minus), ×

(times) , and ÷ (divide). Only arithmetic operators are needed as the domain of MUTT-1 has been restricted to number optimization problems. Indiscriminate application of Eq. (4.4) often results in new entity variables which are semantic nonsense. The derivator uses heuristic rules to restrict its suggestions to combinations that are likely to have semantic meaning. The derivator combines two entity variables only when it finds evidence in the problem description that they are related.

In Section 3.4, we have subdivided the entity variables in the problem description into return properties, resource properties, and limit properties.

Entity variables are related if they share a common construct in the problem description. Two common shared constructs are the objective function and the constraint. It is also possible that two or more constraints have a common variable. For these problems, the derivator includes the constraint obtained by substituting for the common variable in the set of constraints. When combining two variables with a common construct, the derivator uses either the arithmetic operator linking the variables in the common construct or its inverse. This principle guides the derivation of new variables between two return properties, two resource properties, two limit properties, a resource variable and a limit property, and a a return property and a limit property.

The entity variables in the 0/1-knapsack problem do not share any common construct. The entity variables in sequencing to minimize weighted completion time do share common construct. Recall that sequencing to minimize weighted completion time involves a set T of n tasks. Each task $t \in T$ has a length $\ell(t)$, a weight w(t), and a deadline d(t). The desired solution is the one-processor schedule σ for T which minimizes $\sum_{t \in T} [\sigma(t) + \ell(t)]w(t)$ and satisfies $[\sigma(t) + \ell(t)] \leq d(t)$ for each task $t \in T$. The entity variables in this problem are w(), $\ell()$, and d(). $\ell()$ and d() are related by the constraint, $[\sigma(t) + \ell(t)] \leq d(t)$. The derivator groups together these two variables by moving $\ell(t)$ to the right-hand side of the inequality, resulting in the expression, $[\sigma(t) \leq [d(t) - \ell(t)]]$. $\ell()$ and d() are now related by the minus operator, and the derivator generates the new entity variable, $d\Delta\ell()$, which is the difference between d() and $\ell()$. Similarly, the optimization function, $\sum_{t \in T} [\sigma(t) + \ell(t)]w(t)$ causes the derivator to derive the entity variable, $\ell \Lambda w()$, which is the product of $\ell()$ and w().

Another good candidate for a new semantically meaningful entity variable is the ratio of a return property and a resource property. This new variable measure for the per unit return of this entity. In the 0/1-knapsack problem, r()

is a return property, while w() is a resource property. The derivator will derive the new entity variable, $r\Omega w()$, the ratio of r() to w(). It is the return per unit weight of the entity.

When the derivator combines two entity variables of the same type, it regards the new variable as a variable of the same type as the variables used in its definition. The derivator generates a *hybrid property* when the two entity variables in Eq. (4.4) are of different types. The new entity variable $r\Omega w()$ is a hybrid property as it results from an arithmetic combination of a return property with a resource property. Sensible combination of hybrid variables often requires background knowledge which cannot be gleaned from the problem description. The current design of the derivator does not combine hybrid properties with other entity variables.

The derivator always tries to generate sensible state variables. It is much easier to define a nonsensical state variable than a nonsensical entity variable as a state variable is more complex than an entity variable. A state is a data structure which gives a snapshot of the condition of the problem solving process at each stage of its solution and the state variables are elements of this data structure. Thus, the state variable is a summary of an aspect of the decision sequence leading to the node. The derivator's heuristic rules are based upon our experiences in working with search algorithms and can produce some 'promising' state variables.

The first type of state variables known to the derivator are the *summary variables*. Summary variables are summarize the values of entity variables of the set of unassigned entities.

Each item in the 0/1-knapsack problem has two entity variables, r() and w(). The derivator derives the summary variables, RR() and RW(), to correspond to these entity variables. RR() is the sum of r() for the unassigned items, and RW() is the sum of w() for the unassigned items.

The second type of state variables are the *alternate variables*. An alternate variable is an alternative view of another state variable. Although the two variables are equivalent, one variable is often more suitable than the other variable for expressing some fact. The derivator generates an alternate variable when it finds a state variable with a limit; The difference between the limit and the state variable is an alternative view of the state variable.

In the 0/1-knapsack problem, state variable W() is associated with the problem constant K by the constraint $W() \leq K$. The derivator will derive the new variable $K\Delta W()$, where $K\Delta W() = K - W()$. $K\Delta W()$ represents the unused capacity in the knapsack and is an alternative view of W(). However,

KΔW() is more convenient than W() for expressing the condition that all the unassigned items can fit into the knapsack. This condition is |KΔW(x) ≥ RW(x)| when KΔW() is used, and |SUM(W(x),RW(x)) ≤ K| when W() is used. The latter form is more awkward than the former, and is much more difficult to learn.

As indicated in the previous section, the symbol list in MUTT-1 consists of an a-list and a b-list. The b-list, in turn, consists of two sublists: the ba-list and the bb-list. When the assayer first activates the derivator, the a-list, the ba-list, and the ba-list are all empty so the derivator copies the set of symbols in the working description into the a-list.

In subsequent activations, the derivator first appends the contents of the ba-list to the bb-list and moves the contents of the a-list into the ba-list. It then derives sensible variables from the symbols in the b-list according to the rules discussed earlier in this section. There are no restrictions on the type of the variables from the ba-list. The derivator derives a variable that involves a symbol in the bb-list only when it is accompanied by a symbol from the ba-list. Variables involving just symbols in the bb-list have been previously generated.

When the cartographer first activates the derivator for the value-description of the 0/1-knapsack problem, the a-list, ba-list, and bb-list are all empty. The derivator copies the set of symbols in the working description into its a-list. This consists of the two entity variables, w() and r(), the two state variables, W() and R(), and the problem constant B. The a-list is now {R(), W(), r(), w(), K}.

When the cartographer activates the derivator for the second time, the derivator copies the ba-list into the bb-list and moves the contents of the a-list into the ba-list. Thus, the ba-list is now {R(), W(), r(), w(), K}, while the bb-list is still {}. There are no restrictions on the form of the variables derived by the derivator from the ba-list. The derivator will derive the entity variable, rΩw(), the summary variables, RW() and RR(), and the alternative variable, KΔW().

4.7.4 Tester

The tester forms a set of test patterns from the proposed rule: the positive pattern |t-condition| |d-condition$_1$| ··· |d-condition$_m$| |B(x)>B(y)|, the null pattern |t-condition| |d-condition$_1$| ··· |d-condition$_m$| |B(x)=B(y)|, and the negative pattern |t-condition| |d-condition$_1$| ··· |d-condition$_m$| |B(x)<B(y)|. Let the term *positive example* denote a node pair that matches the positive

pattern. Similarly, a *negative example* matches the negative pattern and a *null example* matches the null pattern.

Let the *antirule* be the rule formed by interchanging the symbols x and y in the conditions of a rule. The negative (positive) pattern of the antirule is identical to the positive (negative) pattern of the proposed rule. The null patterns are identical. The sets of test patterns produced from the rule and antirule are isomorphic. A positive example for any rule is a negative example for its antirule and vice-versa.

The rule, IF |SIB(x,y)| |R(x)>R(y)| THEN |B(x)≥B(y)| (rule α_2), is the first rule in map G_o^1 (Figure 4.4) selected for testing by the generator. The tester will create the positive pattern |SIB(x,y)| |R(x)>R(y)| |B(x)>B(y)|, the null pattern |SIB(x,y)| |R(x)>R(y)| |B(x)=B(y)|, and the negative pattern |SIB(x,y)| |R(x)>R(y)| |B(x)<B(y)| from α_2.

Interchanging the symbols x and y in the left-hand side of α_2 produces IF |SIB(x,y)| |R(x)<R(y)| THEN |B(x)≥B(y)|, which is the c-rule, α_3. The positive pattern for α_3 is |SIB(x,y) |R(x)<R(y)| |B(x)>B(y)|, which is identical to the negative pattern of α_2 once the symbols have been renamed. Similarly, the negative and null patterns for α_3, |SIB(x,y) |R(x)<R(y)| |B(x)<B(y)| and |SIB(x,y) |R(x)<R(y)| |B(x)=B(y)|, are respectively equivalent to the positive and null patterns of α_2.

The tester gets one or more test trees from the forester and matches the set of patterns against node pairs in the test trees. Table 4.4 lists the possible outcomes of this matching process and their implications. The presence of both positive and negative examples implies that the set of d-conditions is unable to consistently identify the node with the better solution; thus, the rule and its antirule are both unacceptable rules.

The presence of positive (negative) examples and the absence of negative (positive) examples suggests suggests possible acceptability of the rule (antirule) and unacceptability of the antirule (rule). However, any rule can only be accepted only after it has been subjected to adequate testing. MUTT-1 uses a very simple criteria: the testing is adequate if the number of positive and null examples exceeds ς_1 and the number of test trees exceeds ς_2. Suitable values for ς_1 and ς_2 are 20 and 10 respectively. The tester resumes the matching process if it finds the test set to be inadequate. This simple criteria is sufficient for a prototype program where any errors due to inadequate testing will be apparent to the human experimenter. Statistically sound criteria will have to be developed in the future.

Table 4.4 Possible outcomes of the matching process.

Positive Examples	Null Examples	Negative Examples	Implication	
			Rule	Antirule
yes	no	no	acceptable	unacceptable
yes	yes	no	acceptable	unacceptable
yes	no	yes	unacceptable	unacceptable
yes	yes	yes	unacceptable	unacceptable
no	yes	yes	unacceptable	acceptable
no	no	yes	unacceptable	acceptable
no	no	no	impossible	impossible
no	yes	no	acceptable	acceptable

If there are no examples then either the two rules are impossible or the set of test trees does not contain the proper examples. The tester uses the size of the test set to decide between these two possibilities. If the number of test trees is less than ς_3, then it resumes the matching process with new test trees. A suitable value for ς_3 is 10.

Suppose that the tester will match its set of patterns against the nodes in search tree, τ_1 (Table 4.1). (N_8,N_7), (N_{18},N_{16}), (N_{17},N_{18}), (N_{21},N_{20}), (N_{11},N_{12}), (N_{23},N_{22}), (N_{28},N_{24}), and (N_{27},N_{25}) match the positive pattern; (N_{10},N_9) matches the null pattern; and (N_2,N_1), (N_4,N_3), (N_6,N_5), (N_{12},N_{11}), and (N_{19},N_{18}) match the negative pattern. The presence of both positive and negative examples causes the tester to reject rule α_2 and its antirule α_3.

4.7.5 Scrounger

The scrounger investigates the effects of a decision on other rules. The decision made by the tester can imply decisions for other rules due to the inheritance of decision values and the parallelism between the α-graph and the β-graph.

Suppose that rule c_i is an unacceptable rule. There is at least one unacceptable rule among c_i's children, but there may also be acceptable rules or impossible rules. The scrounger attempts to identify the decision values of the children rules by matching the examples of c_i against d-conditions that connect c_i to its children in the map. Let d_j be the d-condition that connects c_i to c_j in the map. An example of c_i that satisfies a d-condition is an example of the new rule formed by adding the d-condition to c_i. The subset of example set of c_i which match the d_j forms a example set for c_j. If this subset contains both positive and negative examples, then both c_j and its antirule may be rejected.

A positive example of rule α_2 is a node pair that satisfies the conditions [SIB(x,y)], [R(x)>R(y)], and [B(x)>B(y)]. A positive example of α_2 that satisfies the condition [W(x)>W(y)] is a node pair that satisfies the conditions [SIB(x,y)], [R(x)>R(y)], [W(x)>W(y)], and [B(x)>B(y)], that is, a positive example of the resulting child rule α_5. Similarly, a null (negative) example of rule α_2 that satisfies [W(x)>W(y)] is also a null (negative) example of rule α_5.

An empty subset implies that there are no examples of both the α_j and its antirule among the test trees that generated the example set of α_i. The scrounger does not know whether or not the set of test trees used on α_i as an adequate test set. Thus, it labels α_j and its antirule as potentially impossible

rules and waits for the tester to make the final decision on the rules. Additional testing by the tester may reveal that the potentially impossible rule is actually an acceptable rule, an unacceptable rule, or an impossible rule.

Similarly, if the subset does not contain negative (positive) examples of α_i, then α_j is a potential acceptable (impossible) rule and its antirule is a potential impossible (acceptable) rule.

The generate-test-scrounge process gives a higher priority to rules which have been identified as potential impossible or acceptable rules.

For example, the rejection of α_2 causes the scrounger to reject the parallel rules B_2 and β_3. The scrounger then tries to tentatively identify the decision values of α_2's children. In tree r_1 (Table 4.1), node pairs, (N_8,N_7), (N_{16},N_{14}), (N_{17},N_{15}), (N_{21},N_{20}), (N_{11},N_{12}), (N_{23},N_{22}), (N_{26},N_{24}), and (N_{27},N_{25}) are positive examples of α_2. (N_{10},N_9) is a null example of α_2, and node pairs, (N_2,N_1), (N_4,N_3), (N_6,N_5), (N_{12},N_{11}), and (N_{19},N_{18}) are negative examples. The scrounger matches these examples against the condition [W(x)>W(y)] and finds that all of the examples successfully matches the d-condition. There are positive, null, and negative examples of α_5 in tree r_1. The scrounger rejects rule α_5 which leads to the the rejection of α_9, β_6, and β_9.

Next, the scrounger considers the condition [W(x)<W(y)]. Only (N_{11},N_{12}) satisfies this condition. This result implies that there are only positive examples of the resulting rule, α_6, in test tree, r_1. Rule α_6 is a potentially acceptable rule. Additional testing by the tester will reveal that α_6 is an acceptable rule.

Finally, the scrounger considers the condition [W(x)=W(y)]. No example satisfies this condition, so rule α_7 is a potentially impossible rule. Further testing by the tester will determine that α_7 is actually an acceptable α-rule.

The acceptance of a rule may change the order in which rules are proposed by the generator. An acceptable rule indicates the presence of a region of consistent projections in the symbol space. Priority should then be given to determining the actual boundaries of this consistent region because all rules within its boundaries are acceptable rules. This implies priority for the testing of siblings of the acceptable rule and their descendants.

4.7.6 Generator

MUTT-1 keeps a set of supporting reasons for each candidate rule. The generator always proposes the rule with the best supporting reasons. The acceptable reasons are:

(1) The rule is the rule selected by the breadth-first criterion.

(2) The rule is either a sibling of an accepted rule or a descendant of a sibling of an accepted rule.

(3) The scrounger has tentatively identified this rule as either an impossible rule or an acceptable rule.

(4) The parallel β-rule is an acceptable rule, or the parallel α-rule is either an acceptable rule or an impossible rule.

These reasons have been listed in order of increasing priority. The generator tries to select the rule whose most compelling reason has the highest priority level. In case of ties, the generator will continue looking at the priority level of the next most compelling reason until all ties are resolved. The breadth-first criterion ensures that there will always be one 'best' rule. The motivation behind reasons (2), (3), (4), and (6) have been discussed elsewhere in this chapter.

4.8 Verification

Hand-simulations have demonstrated the ability of MUTT-1 to learn dominance relations. A prototype of the searcher has also been implemented. Appendix A contains a listing of the searcher. The implemented searcher uses a static set of user-supplied test search trees as the forester was not implemented. Also, the variable descriptions must be supplied to the implemented searcher in the appropriate form as the description manager was not implemented. Nonetheless, this implementation of the searcher is useful as it demonstrates the validity of three vital aspects of the design, that is, the rule graph description of the rule space, the process of deriving new symbols, and the process of deriving new d-conditions.

When MUTT-1 is applied to the 0/1-knapsack problem, it learns the relation, IF [L(x)=L(y)] [R(x)≥R(y)] [W(x)≤W(y)] THEN [B(x)≥B(y)], which is the well-known dominance relation for the 0/1-knapsack problem. It is also able to learn the relation, IF [L(x)=L(y)] [R(x)≥R(y)] [KΔW(x)≥RW()] THEN [B(x)≥B(y)], which leads to the problem solver to include all of the unassigned items when the capacity of the knapsack exceeds the total weight of

the unassigned items. Another relation learned by MUTT-1 for the 0/1-knapsack problem is IF {SIB(x,y)} {R(x)≥RERR(y)} THEN {B(x)≥B(y)} where RERR() is defined as the sum of R() and RR(). RERR() is an upper bound on the return from any node. Appendix B shows the unmodified trace produced by our implementation of the searcher for the value-description of the 0/1-knapsack problem when supplied with the accompanying input data.

MUTT-1 has also been able to learn the well-known dominance relation found in sequencing to minimize weighted completion time, sequencing to minimize weighted lateness, a simplified inventory problem, a simplified two-machine flowshop problem, sequencing to minimize latest completion time, and sequencing to minimize task weight.

4.9 Summary

MUTT-1 is a prototype program that is able to learn dominance relations by experimentation. It uses the model-driven generate-and-test paradigm, and it models the space of possible candidates by a rule graph. MUTT-1 uses both the properties of this rule graph and semantics of the problem to limit the number of possible candidates proposed and tested by the generate-and-test-process. An integral part of this process is the derivation of new variables. MUTT-1 has been applied to several problems with known dominance relations and has been able to learn those dominance relations.

CHAPTER 5
CONCLUSIONS AND FUTURE WORK

Ways of improving the efficiency of the search process includes reducing the number of nodes searched by the process and reducing the average time to search each node. The number of nodes searched by the search process may be reduced by using an appropriate problem representation, an appropriate selection scheme, and pruning rules. The average time to search each node may be reduced by reducing the mismatch between the algorithm and the architecture. In this thesis, we have studied two of these problems: the effect of the architecture on the selection scheme and the derivation of pruning rules.

Even though the best-first B&B algorithm generally expands the least number of nodes among the different B&B algorithms, it is not always the best B&B algorithm. Most computer systems with large memories use a multi-level hierarchy of at least two levels. On a two-level memory system, the cost of accessing an item of information depends on the level in which the item is located. The secondary memory is significantly slower than the main memory. The best-first B&B algorithm has exponential space and time complexities, while the depth-first B&B algorithm has a linear space complexity and an exponential time complexity. For many problem instances, the use of the best-first B&B algorithm will require the use of both main memory and secondary memory, while the depth-first B&B algorithm often only requires the use of main memory. As a result, the average time required to expand a node in the best-first B&B algorithm is greater than that required by the depth-first B&B algorithm. The best B&B algorithm will depend on the characteristics of the problem and the characteristics of the memory hierarchy. When the disparity between the numbers of nodes expanded by the best-first and depth-first B&B algorithms is small, the depth-first B&B algorithm is generally preferable. On the other hand, when the best-first B&B algorithm expands much less nodes than the depth-first B&B algorithm, the best-first B&B algorithm should be used. For intermediate situations, the algorithm chosen will depend on the relative speeds of the different levels of the hierarchy: it is

the depth-first B&B algorithm when the secondary memory is very slow, and the modified B&B algorithm otherwise.

The modified B&B algorithm partitions the range of possible lower bounds into several distinct subranges. Nodes falling within each subrange are stored in a LIFO stack associated with the subrange. Each stack may partly reside in the secondary memory. The modified B&B strategy does not wait for nodes to be transferred from the secondary memory to the main memory but continues to expand nodes which are currently available in the main memory. The modified B&B algorithm demonstrates the effectiveness of including architectural considerations in the design of an algorithm.

Dominance relations are pruning rules between two nodes in the search tree. Even though dominance relations are problem dependent, it is possible to find them by machine learning techniques. In this thesis, we have studied machine learning of dominance relations for a number of optimization problems. A natural representation for dominance relations is the production rule. The production rule can be easily converted into either a logical implication or a frame. The representation of the optimization problems embeds semantic information into the problem description. This frame representation permits a systematic manipulation of semantic information about the problem domain and facilitates the transfer of information between domains.

Dominance relations can be learned by experimentation, deduction, and analogy. Learning by deduction only learns relations which can be deduced from facts known to the deduction program. Not all of the relevant facts are known about every problem, especially unfamiliar problems. Learning by analogy transforms a known dominance relation into a relation which may be suitable for the new problem. It is most useful when the new problem domain is similar to a domain with known dominance relations. Learning by experimentation is the most general mechanism as it can learn relations in unfamiliar problem domains and learn new types of relations. Thus, learning by experimentation is the primary learning mechanism studied in this thesis. Both analogy and deduction are also necessary as they are very effective when used in the correct context.

We have studied learning dominance relations by experimentation in greater detail and implemented a prototype program. This program uses model-driven generate-and-test due to the impossibility of identifying training instances needed to drive a data-driven method paradigm based upon heuristic constraints derived from manipulations of the standard descriptions as well as

constraints in the space of relations to reduce the number of proposed candidate relations. In addition, the program is able to automatically define and manipulate new variable symbols.

There is much left to do in machine learning of dominance relations before the goal of an integrated system incorporating multiple learning mechanisms is achieved. We have only taken a small step in the process of refining the learning mechanisms. MUTT-1 is a prototype and has to be extended to other types of number optimization problems. We do not anticipate major change in the design of MUTT-1; only modifications to the heuristic knowledge driving the assayer and derivator to reflect new problem domains. Work has also to be done on deriving the knowledge needed to drive the analogy and deduction mechanisms.

This work can be extended into a study of dominance relations for decision problems. There are no simple return variables in decision problems and it is necessary to deal with different types of heuristic measure. There is also a question of the utility of dominance relations for these problems.

In addition, there is the intriguing possibility of using learning by discovery to automate the process of finding the knowledge which drives learning by experimentation, learning by analogy, and learning by deduction.

Another possible avenue of investigation will be to study pruning heuristics comprised of candidate relations which usually give good results. Some questions in this direction are the statistical measures of a good heuristic and the effects of this relaxation on the learning mechanisms.

LIST OF REFERENCES

LIST OF REFERENCES

[AEH84] T. O. Alanko, H. H. A. Erkio, and I. J. Haikala, "Virtual Memory Behavior of some Sorting Algorithms," *IEEE Trans. on Software Engineering*, vol. SE-10, no. 4, pp. 422-431, July 1984.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[Ama86] S. Amarel, "Program Synthesis as a Theory Formation Task: Problem Representations and Solution Methods," *Machine Learning: An Artificial Intelligence Approach - Volume II*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), Morgan Kaufmann, Los Altos, 1986.

[And84] P. M. Andrae, "Constraint Limited Generalization: Acquiring Procedures from Examples," *AAAI-84*, pp. 6-10, 1984.

[AnS79] Y. Anzai and H. Simon, "The Theory of Learning by Doing," *Psychological Review*, Vol. 36, No. 2, pp. 124-140, 1979.

[Ara84] A. A. Araya, "Learning Problem Classes by Means of Experimentation and Generalization," *AAAI-84*, pp. 11-15, 1984.

[BGM70] B. S. Brown, F. G. Gustavson, and E. S. Mankin, "Sorting in a Paging Environment," *CACM*, vol. 13, no. 8, pp. 488-494, Aug. 1970.

[BaS76] J. L. Baer and G. R. Sager, "Dynamic Improvement of Locality in Virtual Memory Systems," *IEEE Trans. on Software Engineering*, vol. SE-2, no. 1, pp. 54-62, March 1976.

[BeD62] R. Bellman and S. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, NJ, 1962.

[BeM71] M. Bellmore and J. C. Malone, "Pathology of Travelling Salesman Subtour-elimination Algorithms," *Operations Research*, vol. 19, no. 19, pp. 278-307, 1971.

[BeN68] M. Bellmore and G. L. Nemhauser, "The Travelling Salesman Problem: A Review," *Operations Research*, vol. 16, pp. 538-558, 1968.

[Ber79] H. Berliner, "The B* Tree Search Algorithm: A Best-First Proof Procedure," *Artificial Intelligence*, Vol. 12, pp. 23-40, 1979.

[Bob80] D. G. Bobrow (Ed.), "Special Issue on Non-Monotonic Logic," *Artificial Intelligence*, vol. 13, 1980.

[BrP81] C. A. Brown and P. W. Purdom, Jr., "An Average Time Analysis of Backtracking," *SIAM J. on Computing*, Vol. 10, No. 3, pp. 583-593, 1981.

[BrT79] F. M. Brown and S. A. Tarnlund, "Inductive Reasoning on Recursive Equations," *Artificial Intelligence*, vol. 12, pp. 207-229, 1979.

[BuM78] B. G. Buchanan and T. M. Mitchell, "Model-Directed Learning of Production Rules," *Pattern-Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (Eds.), Academic Press, 1978.

[Bur86] M. H. Burstein, "Concept Formation by Incremental Analogical Reasoning and Debugging," *Machine Learning: An Artificial Intelligence Approach - Volume II*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), Morgan Kaufmann, Los Altos, 1986.

[CMM83] J. G. Carbonell, R. S. Michalski, and T. M. Mitchell, "An Overview of Machine Learning," *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), Tioga, 1983.

[Car83] J. G. Carbonell, "Learning by Analogy: Formulating and Generalizing Plans from Past Experiences," *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), Tioga, 1983.

[CoF82] P. R. Cohen and E. A. Feigenbaum (Eds.), *The Handbook of Artificial Intelligence*, William Kaufmann, Los Altos, 1982.

[Com79] D. Comer, "The Ubiquitous B-tree," *Computing Surveys*, vol. 11, no. 2, pp. 121-137, June 1979.

[DaL82] R. Davis and D. B. Lenat, *Knowledge-Based Systems in Artificial Intelligence*, McGraw-Hill, 1982.

[Dav78] R. Davis, "Knowledge Acquisition in Rule-based Systems: Knowledge About Representations as a Basis for System Construction and Maintenance," *Pattern-directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (Eds.), Academic Press, New York, 1978.

[DeM77] N. Dershowitz and Z. Manna, "The Evolution of Programs: Automatic Program Modification," *IEEE Trans. on Software Engineering*, vol. SE-3, no. 6, pp. 377-385, 1977.

[Den70] P. J. Denning, "Virtual Memory," *Computing Surveys*, vol. 2, pp. 153-189, 1970.

[DiM83] T. G. Dietterich and R. S. Michalski, "A Comparative Review of Selected Methods for Learning from Examples," *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), Tioga, 1983.

[Die80] T. G. Dietterich, "Applying General Induction Methods to the Card Game Eleusis," *AAAI 1*, pp. 218-220, 1980.

[Doy79] J. Doyle, "A Truth Maintenance System," *Artificial Intelligence*, vol. 12, pp. 231-272, 1979.

[DrL77] S. E. Dreyfus and A. M. Law, *The Art and Theory of Dynamic Programming*, Academic Press, New York, 1977.

[Eas58] W. L. Eastman, "A Solution to the Traveling-Salesman Problem," presented at the *American Summer Meeting of the Econometric Society*, Cambridge, Mass., Aug. 1958.

[EfR66] M. A. Efroymson and T. C. Ray, "A Branch-and-Bound Algorithm for Plant Location," *Operations Research*, Vol. 14, pp. 361-368, 1966.

[Eva68] T. G. Evans, "A Program for the Solution of Geometric Analogy Intelligence Test Questions," *Semantic Information Processing*, M. L. Minsky (Ed.), MIT Press, Cambridge, 1968.

[FHN72] R. E. Fikes, P. E. Hart, and N. J. Nilsson, "Learning and Executing Generalized Robot Plans," *Artificial Intelligence*, Vol. 3, pp. 251-288, 1972.

[Fel71] W. Feller, *An Introduction to Probability Theory and its Applications*, Vol. II, 2nd edition, John Wiley & Sons, Inc., 1971.

[Fer74] D. Ferrari, "Improving Locality by Critical Working Sets," *CACM*, vol. 17, no. 11, pp. 614-620, Nov. 1974.

[Fer75] D. Ferrari, "Tailoring Programs to Models of Program Behavior," *IBM Journal of Research and Development*, vol. 19, no. 3, pp. 244-251, May 1975.

[Fer76] D. Ferrari, "The Improvement of Program Behavior," *IEEE Computer*, pp. 39-47, Nov. 1976.

[FiP79] P. C. Fisher and R. L. Probert, "Storage Reorganization Techniques for Matrix Computation in a Paging Environment," *CACM*, vol.22, no. 7, pp. 405-415, July 1979.

[GaJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, 1979.

[GaN72] R. S. Garfinkel and G. L. Nemhauser, *Integer Programming*, Wiley, New York, 1972.

[Gar73] R. Garfinkel, "On Partitioning the Feasible Set in a Branch-and-Bound Algorithm for the Asymmetric Traveling-Salesman Problem," *Operations Research*, Vol. 21, No. 1, pp. 340-342, 1973.

[GeM72] A. M. Geoffrion and R. E. Marsten, "Integer Programming Algorithms: A Framework and State-of-the-Art Survey," *Management Science*, Vol. 18, No. 9, pp. 465-491, May 1972.

[GrH70] H. Greenberg and R. Hegerich, "A Branch Search Algorithm for the Knapsack Problem," *Management Science*, vol. 16, no. 5, pp. 327-332, 1970.

[HaG71] D. J. Hatfield and J. Gerald, "Program Restructuring for Virtual Memory," *IBM Systems Journal*, vol. 10, no. 3, pp. 168-192, 1071.

[HoS74] E. Horowitz and S. Sahni, "Computing Partitions with Applications to the Knapsack Problem," *J. ACM*, vol. 21, no. 2, pp. 277-292, 1974.

[HoS78] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, 1978.

[Iba73] T. Ibaraki, "Solvable Classes of Discrete Dynamic Programming," *J. of Math. Analysis and Applications*, vol. 43, pp. 642-693, 1973.

[Iba76a] T. Ibaraki, "Computational Efficiency of Approximate Branch-and-Bound Algorithms," *Math. of Oper. Research*, Vol. 1, No. 3, pp. 287-298, 1976.

[Iba76b] T. Ibaraki, "Theoretical Comparisons of Search Strategies in Branch-and-Bound Algorithms," *Int. Jr. of Comp. and Info. Sci.*, Vol. 5, No. 4, pp. 315-344, 1976.

[Iba77] T. Ibaraki, "The Power of Dominance Relations in Branch-and-Bound Algorithms," *JACM*, Vol. 24, No. 2, pp. 264-279, 1977.

[Iba78] T. Ibaraki, "Depth-m Search in Branch-and-Bound Algorithms," *Int. J. of Comp. and Inf. Sci.*, Vol. 7, No. 4, pp. 315-343, 1978.

[IgS65] E. Ignall and L. Schrage, "Application of the Branch and Bound Technique to Some Flow-shop Scheduling Problems," *Operations Research*, vol. 11, pp. 400-412, 1965.

[InK77] G. Ingargiola and J. Korsh, "A General Algorithm for One Dimensional Knapsack Problems," *Operations Research*, Vol. 25, No. 5, pp. 752-759, 1977.

[KaH67] R. Karp and M. Held, "Finite State Processes and Dynamic Programming," *SIAM J. on Applied Math.*, vol. 15, pp. 693-718, 1967.

[Kar76] R. M. Karp, *Algorithms and Complexity: New Direction and Recent Results*, J. F. Traub (Ed.), Academic Press, pp. 1-19, 1976.

[Kli71] R. Kling, "A Paradigm for Reasoning by Analogy," *Artificial Intelligence*, vol. 2, pp. 147-178, 1971.

[KnM75] D. E. Knuth and R. W. Moore, "An Analysis of Alpha-Beta Pruning," *Artificial Intelligence*, Vol. 6, pp. 293-326, 1975.

[KoS74] W. Kohler and K. Steiglitz, "Characterization and Theoretical Comparison of Branch-and-Bound Algorithms for Permutation Problems," *JACM*, Vol. 21, No. 1, pp. 140-156, 1974.

[Kor80] R. E. Korf, "Toward a Model of Representation Changes," *Artificial Intelligence*, Vol. 14, pp. 41-78, 1980.

[KuK83] V. Kumar and L. Kanal, "A General Branch-and-Bound Formulation for Understanding and Synthesizing And/Or Tree Search Procedures," *Artificial Intelligence*, Vol. 21, pp. 179-198, 1983.

[LBS83] P. Langley, G. L. Bradshaw, and H. A. Simon, "Rediscovering Chemistry with the BACON System," *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), Tioga, 1983.

[LHK79] D. B. Lenat, F. Hayes-Roth, and P. Klahr, "Cognitive Economy in Artificial Intelligence Systems," *IJCAI 6*, pp. 531-536, 1979.

[LaD60] A. H. Land and A. Doig, "An Automatic Method for Solving Discrete Programming Problems," *Econometrica*, Vol. 28, pp. 497-520, 1960.

[LaW66] E. L. Lawler and D. W. Wood, "Branch-and-Bound Methods: A Survey," *Operations Research*, Vol. 14, pp. 699-719, 1966.

[Lan81] P. Langley, "Data-Driven Discovery of Physical Laws," *Cognitive Science*, Vol. 5, pp. 31-54, 1981.

[Len76] J. Lenstra, "Sequencing by Enumerative Methods," *Math. Centre. Tract 69*, Mathematisch Centrum, Amsterdam, 1976.

[Len82] D. B. Lenat, "The Nature of Heuristics," *Artificial Intelligence*, Vol. 19, pp. 189-249, 1982.

[Len83a] D. B. Lenat, "Theory Formation by Heuristic Search; The Nature of Heuristics II: Background and Examples," *Artificial Intelligence*, Vol. 21, pp. 31-59, 1983.

[Len83b] D. B. Lenat, "EURISKO: A Program that Learns New Heuristics and Domain Concepts; The Nature of Heuristics III: Program Design and Results," *Artificial Intelligence*, Vol. 21, pp. 61-98, 1983.

[Li85] G.-J. Li, *Parallel Processing of Combinatorial Search Problems*, Ph. D. Thesis, Purdue University, W. Lafayette, 1985.

[LiW84] G. J. Li and B. W. Wah, "Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms," *Proc. Int'l Conf. on Parallel Processing*, Aug. 1984, pp. 473-480.

[LiW85] G.-J. Li and B. W. Wah, "MANIP-2: A Multicomputer Architecture for Evaluating Logic Programs," *Proc. Int'l Conf. on Parallel Processing*, IEEE, pp. 123-130, June 1985.

[MUB83] T. M. Mitchell, P. E. Utgoff, and R. B. Banerji, "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics," *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), Tioga, 1983.

[MaM78] A. Martelli and U. Montanari, "Optimizing Decision Trees Through Heuristically Guided Search," *CACM*, Vol. 21, pp. 1025-1039, 1978.

[MaW75] Z. Manna and R. J. Waldinger, "Knowledge and Reasoning in Program Synthesis," *Artificial Intelligence*, vol. 6, pp. 175-208, 1975.

[McC69] A. C. McKellar and E. G. Coffman, Jr., "Organizing Matrices and Matrix Operations for Paged Memory Systems," *CACM*, vol. 12, no. 3, pp. 153-164, 1969.

[McD78] D. McDermott, "Planning and Acting," *Cognitive Science*, vol. 2, pp. 71-109, 1978.

[McD79] J. McDermott, "Learning to Use Analogies," *IJCAI 6*, pp. 568-576, 1979.

[MiC80] R. S. Michalski and R. L. Chilausky, "Learning by Being Told and Learning From Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis," *Int. J. of Policy Analysis and Information Systems*, vol. 4, pp. 125-161, 1980.

[Mic83] R. S. Michalski, "A Theory and Methodology of Inductive Learning," *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), Tioga, 1983.

[Mic86] R. S. Michalski, "Understanding the Nature of Learning: Issues and Research Directions," *Machine Learning: An Artificial Intelligence Approach - Volume II*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), Morgan Kaufmann, Los Altos, 1986.

[Min75] M. Minsky, "A Framework for Representing Knowledge," *The Psychology of Computer Vision*, P. H. Winston (Ed.), McGraw-Hill, New York, 1975.

[Min84] S. Minton, "Constraint-Based Generalization: Learning Game-Playing Plans from Single Examples," *AAAI-84*, pp 251-254, 1984.

[Mit70] L. Mitten, "Branch-and-Bound Methods: General Formulation and Properties," *Operations Research*, Vol. 18, pp. 24-34, 1970.

[Mit82] T. M. Mitchell, "Generalization as Search," Artificial Intelligence Vol. 18, pp. 203-226, 1982.

[MoH79] D. J. Mostow and F. Hayes-Roth, "Operationalizing Heuristics: Some AI Methods for Assisting AI Programming," *IJCAI 6*, pp. 601-609, 1979.

[MoN73] J. A. Moore and A. Newell, "How Can MERLIN Understand?" *Knowledge and Cognition*, L. Gregg (Ed.), Erlbaum, Hillsdale, 1973.

[Mor66] J. E. Morrison, "User Program Performance in Virtual Storage Systems," *IBM Systems Journal*, vol. 5, no. 2, pp. 216-23, 1966.

[NeS56] A. Newell and H. A. Simon, "The Logic Theory Machine," *IRE Transactions on Information Theory*, vol. 2, pp. 61-79, 1956.

[Nil71] N. J. Nilsson, *Problem Solving Methods in Artificial Intelligence*, McGraw Hill, New York, 1971.

[Nil80] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Company, Palo Alto, California, 1980.

[Ohl83] S. Ohlsson, "A Constrained Mechanism for Procedural Learning" *IJCAI-83*, pp. 426-428, 1983.

[Pea84] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984.

[Rob65] J. A. Robinson, "A Machine-oriented Logic Based on the Resolution Principle," *J. ACM*, vol. 12, pp. 23-41, 1965.

[Ros70] S. M. Ross, *Applied Probability Models with Optimization Applications*, Holden-Day, San Francisco, 1970.

[Sah77] S. Sahni, "General Techniques for Combinatorial Approximation," *Operations Research*, vol. 25, no. 6, pp. 920-936, 1977.

[Sam59] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM J. Research and Development*, vol. 3, pp. 210-229, 1959.

[Sam67] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers. II - Recent Progress," *IBM J. Research and Development*, vol. 11, pp. 601-617, 1967.

[Sek81] Y. Sekiguchi, "A Unifying Framework of Combinatorial Optimization Algorithms: Tree Programming and its Validity," *J. Operations Research of Japan*, vol. 24, no. 1, 1981.

[SiL74] H. A. Simon and G. Lea, "Problem Solving and Rule Induction: A Unified View," *Knowledge and Cognition*, L. Gregg (Ed.), Lawrence Erlbaum, Hillsdale, 1974.

[Smi78] A. J. Smith, "Sequentiality and Prefetching in Database Systems," *ACM Trans. on Database Systems*, vol. 3, no. 3, pp. 223-247, Sept. 1978.

[Smi84] D. R. Smith, "Random Trees and the Analysis of Branch-and-Bound Procedures," *Journal of the ACM*, vol. 31, no. 1, pp. 163-188, 1984.

[StK83] G. C. Stockman and L. N. Kanal, "Problem Reduction Representation for the Linguistic Analysis of Waveforms," *IEEE Trans. on PAMI*, Vol. 5, No. 3, pp. 287-298, 1983.

[Sto84] H. S. Stone, "The Average Complexity of Depth-First Search," IBM Research Report RC 10717, 1984.

[Sus75] G. J. Sussman, *A Computer Model of Skill Acquisition*, American Elsevier, New York, 1971.

[Ver78] S. A. Vere, "Inductive Learning of Relational Productions," *Pattern-Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (Eds.), Academic Press, 1978.

[WLY85] B. W. Wah, G.-J. Li, and C. F. Yu, "*Multiprocessing of Combinatorial Search Problems*," *IEEE Computer*, vol. 18, no. 6, pp. 93-108, June 1985.

[WOL84] P. H. Wos, R. Overbeek, E. Lusk, and J. Boyle, *Automated Reasoning: Introduction and Applications* Prentice-Hall, New Jersey, 1984.

[WRC65] L. Wos, G. A. Robinson, and D. F. Carson, "Efficiency and Completeness of the Set of Support Strategy in Theorem Proving," *J. ACM*, vol. 12, pp. 536-541, 1965.

[WaM84] B. W. Wah and Y. W. Ma, "MANIP–A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems," *IEEE Trans. on Computers*, vol. C-33, no. 5, pp. 377-390, May 1984.

[WaY82] B. W. Wah and C. F. Yu, "Probabilistic Modeling of Branch-and-Bound Algorithms," *Proc. COMPSAC*, IEEE, pp. 647-623, 1982.

[WaY85] B. W. Wah and C. F. Yu, "Stochastic Modeling of Branch-and-Bound Algorithms with Best-First Search," *IEEE Trans. on Software Engineering*, vol.SE-11, no. 9, pp. 922-934, Sept. 1985.

[Wat70] D. A. Waterman, "Generalization Learning Techniques for Automating the Learning of Heuristics," *Artificial Intelligence*, Vol. 1, No. 1/2, pp. 121-170, 1970.

[Whi60] D. White, *Dynamic Programming*, Oliver and Boyd, Edinburgh, 1960.

[Win75] P. H. Winston, "Learning Structural Descriptions from Examples," *The Psychology of Computer Vision*, P. H. Winston (Ed.), McGraw Hill, New York 1975.

[Win79] P. H. Winston, "Learning and Reasoning by Analogy," *CACM*, Vol. 23, pp. 689-703, 1979.

[Win84] P. H. Winston, *Artificial Intelligence* Addison-Wesley, Reading, 1984.

[YuW83] C. F. Yu and B. W. Wah, "Virtual-Memory Support for Branch-and-Bound Algorithms," *Proc. of COMPSAC*, pp. 618-626, 1983.

[YuW84] C. F. Yu and B. W. Wah, "Efficient Branch-and-Bound Algorithms on a Two-Level Memory System," *Proc. of COMPSAC*, pp. 504-514, 1984.

APPENDICES

Appendix A: A Prototype of the Searcher

```
;*****************************************************************
;
; an implementation of a prototype of the searcher
; warning: this code is not USDA approved and may contain bugs
;
;*****************************************************************
;
; searcher : the searcher
;
(defun searcher (var-file inst-file)
  (prog (frame nu-list)
    (init inst-file)
    next-map
    (setq nu-list (assay var-file)) ; find new d-conditions
    (cond [(null nu-list) (return t)]) ; quit if no new d-condition
    ; i.e. assayer and derivator have exhausted their limited knowledge
    (setq Condlist (append nu-list Condlist)) ; add to known d-conditions
    (cart)   ; update alpha and beta maps
    next-frame
    (cond
      [(null Activelist) (go next-map)] ; maps completely processed
      [t (setq frame (eval (generate)))
        (tester frame)
        (scrounger frame)
        (go next-frame)]
    );cond
  );prog
);defun
;*****************************************************************
;
; function that implements the generator
;
;*****************************************************************
;
; generate : the generator
;          (assumes Activelist present)
;
(defun generate ()
  (prog (choice choicerank frame antichoice clist crule)
    (setq choice nil)
    (setq choicerank 0)
    (setq clist Activelist)
    next-crule
```



```
    (cond
      [(null clist) ; best choice has been identified
        ; delete choice from Activelist
        (setq Activelist (delete choice Activelist))
        (remprop 'Rank choice)
        ; delete antirule from Activelist if there is and antirule
        ; and it is a memebr of Activelist
        (setq frame (eval choice))
        (setq antichoice (vref frame cAnti))
        (cond [(and antichoice (member antichoice Activelist))
                (setq Activelist (delete antichoice Activelist))
                (remprop 'Rank antichoice)]
        );cond
        (prine "Generator chooses rule ") (prine choice)
        (prine " of rank ") (prine choicerank) (terpr)
        (return choice)]
      [t ; keep on comparing with other elements of Activelist
        (setq crule (car clist))
        (setq clist (cdr clist))
        (cond
          [(> (get 'Rank crule) choicerank) ; crule has higher rank
            (setq choice crule) ; make crule the new choice
            (setq choicerank (get 'Rank crule))]) ; update the rank
        (go next-crule)]
    );cond
  );prog
);defun
;*****************************************************************
;
; functions that implement the tester
;
;*****************************************************************
;
; tester : the tester
;
(defun tester (cframe)
  (prog (delta tframe treelist arule aframe crule)
    (setq delta (vref cframe cDelta))
    (setq crule (vref cframe cName))
    (prine "Testing rule ") (prine crule)
    (prine " with delta ") (prine delta) (terpr)
    (cond
      [(equal delta nil)
        (prog ()
          (setq instlist Inst-list)
          next-tree
          (cond
            [(null instlist)    ; all trees done
              (go evaluate)]    ; go evaluate test results
          );cond
          ; some more search trees left, work on these
          (setq inst (car instlist))
```

```
      (setq instlist (cdr instlist))
      (match cframe inst)
      (go next-tree)
      evaluate
      (princ "tester: rule ") (princ crule) (princ " is ")
      (princ (ieval cframe)) (terpri)
      (cond ((equal (vref cframe cDelta) 'acceptable)
             (cdump1 cframe)))
      (vset cframe cEv '(test))
      (setq arule (vref cframe cAnti))
      (cond ((null arule) (return (list crule)))) ; no antirule
      (cond
        ((equal arule crule) ; rule same as antirule
             (return (list crule))))
      (setq aframe (eval arule))
      (vset aframe cPoslst (vref cframe cNeglst))
      (vset aframe cNeglst (vref cframe cPoslst))
      (vset aframe cNullst (vref cframe cNullst))
      (vset aframe cNumpos (vref cframe cNumneg))
      (vset aframe cNumneg (vref cframe cNumpos))
      (vset aframe cNumnul (vref cframe cNumnul))
      (princ "tester: antirule ") (princ arule) (princ " is ")
      (princ (ieval aframe)) (terpr)
      (cond ((equal (vref aframe cDelta) 'acceptable)
             (cdump1 aframe)))
      (vset aframe cEv '(test anti))
      (return (list crule arule))
    );prog
   );equal
  );cond
  (cond
    ((equal delta 'tmp-acceptable) ; hook for tmp-acceptable
                          (vset cframe cDelta 'acceptable)
                          (return nil))
  );cond
  (cond
    ((equal delta 'tmp-impossible) ; hook for tmp-impossible
                          (vset cframe cDelta 'impossible)
                          (return nil))
  );cond
  (cond
    ((equal delta 'acceptable) (return nil))
  );cond
  (cond
    ((equal delta 'unacceptable) (return nil))
  );cond
  (cond
    ((equal delta 'impossible) (return nil))
  );cond
 );prog
);defun
;
```



```
; ieval : evaluate example set of c-rule and determine its delta value
;
(defun ieval (frame)
 (prog ()
  (cond
    ((> (vref frame cNumneg) 0) ; there are -ve examples
     (vset frame cDelta 'unacceptable)
     (return 'unacceptable))
  );cond
  ; no -ve examples
  (cond
    ((> (vref frame cNumpos) 0) ; there are +ve examples
     (vset frame cDelta 'acceptable)
     (return 'acceptable))
  );cond
  ; no -ve and +ve examples
  (cond
    ((> (vref frame cNumnul) 0)  ; there are null examples
     (vset frame cDelta 'acceptable)
     (return 'acceptable))
  );cond
  ; no +ve, -ve, or null examples
  (vset frame cDelta 'impossible)
  (return 'impossible)
 );prog
);cond
;
; match : find instances of c-rule in test tree
;
(defun match (cframe inst)
 (prog (nodelst lhs x y tframe tlev levlst numlev ql qt xlst clst)
  (setq tframe (eval (get inst 'tree)))
  (setq qt inst)
  (setq tlev (get inst 'levlist))
  (setq levlst (cdr (vref tframe tNodelst)))
  (setq numlev (1- (car (vref tframe tNodelst))))
  (setq lhs (vref cframe cLhs))
  nxtlev
  (cond
    ((zerop numlev) t)
    (t (setq nodelst (cdar levlst))
     (setq levlst (cdr levlst))
     (setq xlst nodelst)
     (setq ql (car tlev))
     (setq tlev (cdr tlev))
     (prog ()
      nxtx
      (cond
        ((null xlst) t)
        (t (setq x (car xlst))
         (setq xlst (cdr xlst))
         (setq ylst nodelst)
```

```
        (prog ()
          nxtnode
          (cond
            ((null ylst) t)
            (t (setq y (car ylst))
               (setq ylst (cdr ylst))
               (cond ((equal x y) (go nxtnode)))
               (prog ()
                ; test if node pair is instance
                (setq clst lhs)
                nxtcond
                (cond
                  ((null clst) t)
                  ((null (eval (car clst))) (go nxtnode))
                  (t (setq clst (cdr clst))
                     (go nxtcond)))
                ;cond
                ; node pair is instance
                ; find type of instance
                (cond
                  ((> (get x 'B) (get y 'B)) ; +ve instance
                   (vset cframe cNumpos (1 + (vref cframe cNumpos)))
                   (vset cframe cPoslst
                     (append1 (vref cframe cPoslst)
                          (list x y))))
                  ((< (get x 'B) (get y 'B)) ; -ve instance
                   (vset cframe cNumneg (1 + (vref cframe cNumneg)))
                   (vset cframe cNeglst
                     (append1 (vref cframe cNeglst)
                          (list x y))))
                  ((equal (get x 'B) (get y 'B))    ; null instance
                   (vset cframe cNumnul (1 + (vref cframe cNumnul)))
                   (vset cframe cNullst
                     (append1 (vref cframe cNullst)
                          (list x y))))
                ;cond
                (go nxtnode)
               );prog
              )t
            );cond
          );prog
          (go nxtx)  ; try next candidate for x
         )t
        );cond
       );prog
       (setq numlev (1- numlev))
       (go nxtlev)
      )t
     );cond
    );prog
);defun
;*****************************************************************
```



```
;
; functions that implement the scrounger
;
;*****************************************************************
;
; scrounger
;
(defun scrounger (cframe)
  (prog (arule alist antirule)
    (setq arule (vref cframe cName))
    (princ "Scrounger called with rule ") (princ arule) (terpr)
    (setq antirule (vref cframe cAnti))
    (cond
      ((null antirule) ; no antirule .. it happens
       (setq alist (list arule))
       (go next-arule))
    );cond
    ; there is a non-nil antirule
    (cond
      ((equal antirule arule) ; rule is its own antirule
       (setq alist (list arule))
       (go next-arule))
    );cond
    ; there is a distinct non-nil antirule
    (setq alist (list arule antirule))
    next-arule
    (cond
      ((null alist) ; all jobs done, so quit
       (return))
    );cond
    ; there are still jobs to be done, so do it
    (setq arule (car alist))
    (setq alist (cdr alist))
    (cond ((setq nu-jobs (parallel-delta arule))
           (setq alist (append alist nu-jobs))))
    );cond
    (cond ((setq nu-jobs (parent-delta arule))
           (setq alist (append alist nu-jobs))))
    );cond
    (cond ((setq nu-jobs (child-delta arule))
           (setq alist (append alist nu-jobs))))
    );cond
    (go next-arule)
  );prog
);cond
;
; parallel-delta : adjust parallel rule
;
(defun parallel-delta (arule)
  (prog (adelta aframe atype brule blist bframe)
    (setq aframe (eval arule))
    (setq brule (vref aframe cParallel))
```

```
    (cond ((null brule) (return nil))) ; no parallel rule
    ; there is a parallel rule
    (setq bframe (eval brule))
    (setq adelta (vref aframe cDelta))
    (setq atype (vref aframe cType))
    (cond ((and (equal adelta 'unacceptable) (equal atype 'alpha))
           (go set-value))
    );cond
    (cond ((and (equal adelta 'impossible) (equal atype 'beta))
           (go set-value))
    );cond
    ; remaining cases can only result in upgrading the ranking
    (cond ((vref bframe cDelta) (return nil))) ; delta of brule known
    ; delta of brule not known so upgrade its ranking
    (putprop 'Rank 4 brule)
    (princ "scrounger: ") (princ brule) (princ " is parallel to ")
    (princ adelta) (princ " c-rule ")
    (princ arule) (princ " -> increase rank of ") (princ brule) (terpri)
    (return nil) ; no new decision to scrounge on
    ;
    ; rest is code to handle setting of decision
    ;
    set-value
    (cond ((null (vref bframe cDelta)) ; delta of brule not known
      (vset bframe cDelta adelta)
      (vset bframe cEv (list 'parallel 'to arule))
      (princ "scrounger: ") (princ brule) (princ " is parallel to ")
      (princ adelta) (princ " c-rule ")
      (princ arule) (princ " -> ") (princ brule) (princ " is ")
      (princ adelta) (terpri)
      (setq Activelist (delete brule Activelist))
      (remprop 'Rank brule)
      (return (list brule)))
    );cond
    ;
    ; delta of brule known ... set if confirmed by parallelism
    ;
    (cond ((equal adelta (vref bframe cDelta)) ; confirms delta
      (return nil))
    );
    ;
    ; contradiction
    ;
    (vset bframe cEv (append (vref bframe cEv)
                     (list 'parallelism 'to arule 'implies adelta)))
    (return nil)
  );prog
);defun
;
; parent-delta : adjust parent rules
;
(defun parent-delta (arule)
```



```
  (prog (aframe brule bframe blist nu-rules)
    (setq aframe (eval arule))
    ; nothing can be deduced if adelta is not 'unacceptable
    (cond ((nequal (vref aframe cDelta) 'unacceptable) (return nil)))
    ; adelta is 'unacceptable
    (setq blist (vref aframe cParent))
    (setq nu-rules nil) ; list of jobs to be added to joblist
    next-brule
    (cond ((null blist) (return nu-rules))) ; all parents done
    (setq brule (car blist))
    (setq blist (cdr blist))
    (setq bframe (eval brule))
    (cond ((null (vref bframe cDelta)) ; delta of brule not known
      (vset bframe cDelta 'unacceptable) ; set implied decision
      (vset bframe cEv (list 'parent 'of arule))
      (princ "scrounger: ") (princ brule)
      (princ " is parent of unacceptable c-rule ")
      (princ arule) (princ " -> ") (princ brule)
      (princ " is unacceptable") (terpri)
      (setq Activelist (delete brule Activelist))
      (remprop 'Rank brule)
      (setq nu-rules (append1 nu-rules brule))
      (go next-brule))
    );cond
    ; delta of brule is known, so try to confirm
    (cond ((equal (vref bframe cDelta) 'unacceptable) ; confirms delta
      (go next-brule))
    )
    ; contradiction
    (vset bframe cEv (append (vref bframe cEv)
                     (list 'parenthood 'of arule 'implies 'unacceptable)))
    (go next-brule)
  );prog
);defun
;
; child-delta : find implied delta values of children rules
;
(defun child-delta (arule)
  (prog (aframe adelta brule blist bframe nu-rules)
    (setq aframe (eval arule))
    (setq adelta (vref aframe cDelta))
    (setq blist (vref aframe cChildren)) ; list of arule's children
    (setq nu-rules nil)
    next-brule
    (cond ((null blist) (return nu-rules))) ; all children processed
    (setq brule (car blist))
    (setq blist (cdr blist))
    (setq bframe (eval brule))
    (setq bdelta (vref bframe cDelta))
    (cond ((and (equal adelta 'impossible) (null bdelta))
      ; bdelta not known
      (vset bframe cDelta 'impossible) ; set implied decision
```

```
  (vset bframe cEv (list 'parent 'of arule))
  (princ "scrounger: ") (princ brule)
  (princ " is a child of impossible c-rule ")
  (princ arule) (princ " -> ") (princ brule)
  (princ " is impossible") (terpr)
  (setq Activelist (delete brule Activelist))
  (remprop 'Rank brule)
  (setq nu-rules (append1 nu-rules brule))
  (go next-brule)]
 );cond
 (cond [(and (equal adelta 'impossible) (equal bdelta 'impossible))
  ; decision confirmed
  (go next-brule)]
 );cond
 (cond [(equal adelta 'impossible)
  ; contradiction
  (vset bframe cEv (append (vref bframe cEv)
                           (list 'child 'of arule 'implies adelta)))
  (go next-brule)]
 );cond
 (cond [(and (equal adelta 'acceptable) (null bdelta))
  ; bdelta is nil
  (vset bframe cDelta 'acceptable) ; set implied decision
  (vset bframe cEv (list 'child 'of arule))
  (princ "scrounger: ") (princ brule)
  (princ " is a child of acceptable c-rule ")
  (princ arule) (princ " -> ") (princ brule)
  (princ " is acceptable/impossible") (terpr)
  (setq Activelist (delete brule Activelist))
  (remprop 'Rank brule)
  (setq nu-rules (append1 nu-rules brule))
  (go next-brule)]
 );cond
 (cond [(and (equal adelta 'acceptable) (member bdelta '(acceptable impossible))
  ; confirms delta
  (go next-brule)]
 );cond
 (cond [(equal adelta 'acceptable)
  ; contradiction
  (vset bframe cEv (append (vref bframe cEv)
                           (list 'child 'of arule 'implies 'acceptable 'or 'unacceptable)))
  (go next-brule)]
 );cond
 (cond [(and (equal adelta 'unacceptable)
             (null (member 'test (vref aframe cEv))))
  (go next-brule)]
 );cond
 (cond [(and (equal adelta 'unacceptable) (null bdelta))
  (vset bframe cDelta (no-divide aframe bframe))
  (vset bframe cEv (list 'child 'of arule))
  (princ "scrounger: ") (princ brule)
  (princ " is a child of unacceptable c-rule ")
  (princ arule) (princ " -> partition shows ")
  (princ brule) (princ " is ")
  (princ (vref bframe cDelta)) (terpr)
  (cond [(equal (vref bframe cDelta) 'acceptable) (cdump1 bframe)])
  (setq Activelist (delete brule Activelist))
  (remprop 'Rank brule)
  (setq nu-rules (append1 nu-rules brule))
  (go next-brule)]
 );cond
 (go next-brule)
 );prog
);defun
;
; no-divide : find instances of parent that satisfy a c-rule
;
(defun no-divide (aframe bframe)
 (prog ()
  ; find instances of arule
  (setq link (vref bframe cCond))
  (vset bframe cPoslst (cons '+ (portion link (cdr (vref aframe cPoslst)))))
  (vset bframe cNeglst (cons '- (portion link (cdr (vref aframe cNeglst)))))
  (vset bframe cNullst (cons 0 (portion link (cdr (vref aframe cNullst)))))
  (vset bframe cNumpos (1- (length (vref bframe cPoslst))))
  (vset bframe cNumneg (1- (length (vref bframe cNeglst))))
  (vset bframe cNumnul (1- (length (vref bframe cNullst))))
  ; evaluate instance set of arule
  ; not really the same as for tester
  ; include hooks
  (cond [(> (vref bframe cNumneg) 0) ; there are -ve instances
   (return 'unacceptable)]
  );cond
  ; no -ve instances
  (cond [(> (vref bframe cNumpos) 0) ; there are +ve instances
   ; code here is a kludge
   (return 'acceptable)]
  );cond
  ; no +ve and no -ve instances
  (cond [(> (vref bframe cNumnul) 0) ; there are null instances
   ; code here also a kludge
   (return 'acceptable)]
  );cond
  ; no instances of any kind
  ; again, code here is a kludge
  (return 'impossible)
 );prog
);defun
;
; portion : find subset that satisfies give condition
;
(defun portion (link eglst)
 (prog (eg x y qt ql num alist)
 ;(princ link) (terpr)
```

```
(setq nlist nil)
nxteg
(cond
  ((null eglst) t)
  (t
    (setq eg (car eglst))
    (setq eglst (cdr eglst))
    (setq x (car eg))
    (setq y (cadr eg))
    (setq qt (get x 'qt))
    (setq ql (get x 'ql))
    (cond
      ((eval link)
        (setq nlist (append1 nlist (list x y)))
      ));cond
    (go nxteg)
  );t
);cond
(return nlist)
);prog
);defun
;*****************************************************************************
;
; functions that implement the cartographer
;
;*****************************************************************************
;
; cart: cartographer
;
(defun cart ()
 (prog ()
   (princ "Cartographer") (terpr)
   (setq lev 1)
   (setq maxlev (1- Lmaxlev)) ; lev refers to level of parent
   next-lev0
   (cond
     ((and (< lev maxlev) (vref aLevlist lev))
       (makelev lev 'ca aLevlist Condlist 2)
       (setq lev (1+ lev))
       (go next-lev0))
   );cond
   (setq lev 1)
   next-lev1
   (cond
     ((and (< lev maxlev) (vref bLevlist lev))
       (makelev lev 'cb bLevlist Condlist 1)
       (setq lev (1+ lev))
       (go next-lev1))
   );cond
   (setcor) ; set the parallel rules
   (nu-anti) ; update list of condition - anti-condition pairs
```



```
   (setanti) ; set rule-antirule relationships
   (setq Cdumplist (nreverse Cdumplist))
   (mapc 'cdump0 Cdumplist)
   (setq Cdumplist nil)
 )
);defun
;
; makelev
;
(defun makelev (plev root levlist condlist rank)
 (prog (lev type)
   (cond ((equal root 'ca) (setq type 'alpha))
         ((equal root 'cb) (setq type 'beta))
         (t (error)))
   (setq plist (vref levlist plev))
   (setq lev (1+ plev))
   next-parent
   (cond
     ((null plist) t)
     (t ; some more parent nodes available
       (setq parent (eval (car plist)))
       (setq plist (cdr plist))
       ; don't generated children of 'acceptable or 'impossible rules
       (setq delta (vref parent cDelta))
       (cond
         ((equal delta 'acceptable) (go next-parent))
         ((equal delta 'impossible) (go next-parent))
       );cond
       ; rule is 'unacceptable or nil, so generate children
       (setq op-list (findlink parent condlist))
       ; create a cframe for each operator
       (prog ()
         next-op
         (cond
           ((null op-list) t) ; all done
           (t                 ; some more operators available
             (setq operator (car op-list))
             (setq op-list (cdr op-list))
             (setq lhs (append1 (vref parent cLhs) operator))
             (setq cname (cmatch lev lhs levlist))
             (cond
               ((null cname)
                 (setq cname (newsym root))
                 (set cname (new-vector cVsize))
                 (setq cframe (eval cname))
                 (vset cframe cName cname)
                 (vset cframe cType type)
                 (vset cframe cLhs lhs)
                 (vset cframe cCond operator)
                 (vset cframe cParent (list (vref parent cName)))
                 (vset cframe cPoslst '(+))
                 (vset cframe cNeglst '(-))
```

```
            (vset cframe cNullst '(0))
            (vset cframe cNumpos 0)
            (vset cframe cNumneg 0)
            (vset cframe cNumnul 0)
            (vset levlist lev (append1 (vref levlist lev) cname))
            (setq Activelist (append1 Activelist cname))
            (putprop 'Rank rank cname)
            (setq Cdumplist (cons cname Cdumplist))
           ];t
          );cond
          (vset parent cChildren
                (append1 (vref parent cChildren) cname))
          (go next-op)
         ];t
        );cond
       );prog
       (go next-parent)
      ]
    );cond
  );prog
);defun
;
; cmatch : find if c-rule is a duplicate of an existing c-rule
;          ( different generating sequences )
;
(defun cmatch (lev lhs levlist)
  (prog (nlist node nlist xlist xcon)
    (setq nlist (vref levlist lev))
    next-frame
    (cond [(null nlist) (return nil)])
    (setq node (eval (car nlist)))
    (setq nlist (cdr nlist))
    (setq xlist (vref node cLhs))
    next-cond
    (cond
      [(null xlist)
       (return (vref node cName))]
      [t (setq xcon (car xlist))
       (setq xlist (cdr xlist))
       (cond [(member xcon lhs) (go next-cond)]
             [t (go next-frame)]
       );cond
      ]
    );cond
  );prog
);defun
;
; findlink : find acceptable links for generating children nodes
;    (not part of parent's lhs and not in previously generated siblings)
;
(defun findlink (parent linklist)
  (prog (op-list link curlist curlink sib-list sibling)
```



```
    (setq op-list nil)
    next-link
    (cond [(null linklist) (return op-list)]) ; all done
    (setq link (car linklist))
    (setq linklist (cdr linklist))
    ; check if link already occurs in parent
    (setq curlist (vref parent cLhs))
    next-curlink
    (cond
      [(null curlist) t] ; link not part of parent
      [t ; check if curlink occurs in parent
       (setq curlink (car curlist))
       (setq curlist (cdr curlist))
       (cond
         [(equal (cdr link) (cdr curlink)) ; link part of parent
                 (go next-link)]    ; try next link
       ; curlink is not part of parent
       (go next-curlink)
      ]
    )
    ; link not part of parent - check if part of any sibling
    (setq sib-list (vref parent cChildren))
    next-sib
    (cond
      [(null sib-list)                  ; link not in siblings
       (setq op-list (append1 op-list link)) ; link is acceptable
       (go next-link)]                  ; try next link
      [t
       (setq sibling (eval (car sib-list)))  ; check next sibling
       (setq sib-list (cdr sib-list))
       (cond
         [(nequal link (vref sibling cCond)) (go next-sib)] ; no match
         [t (go next-link)]
       )
      ]
    )
  );prog
);defun
;
; findcor : find parallel rule
;          (implicit : lhs of rule contain at least one d-condition)
;
(defun findcor (lev levlist corlevlist rank)
  (prog (alist arule aframe alhs blist bframe bcondlist bcondition)
    (setq alist (vref levlist lev))
    next-arule
    (cond [(null alist) (return)])
    (setq arule (car alist))
    (setq aframe (eval arule))
    (setq alhs (vref aframe cLhs))
    (setq alist (cdr alist))
    (cond [(vref aframe cParallel) (go next-arule)]) ; rule known
    ; rule not known, so find parallel rule
```

```
  (setq alhs (vref aframe cLhs))
  (setq blist (vref corlevlist lev))
  next-brule
  (cond [(null blist) ; || rule not in map - either 'acceptable or 'impossible
    ; adjust ranking of rule
    (putprop 'cRank rank arule)
    (go next-arule)])
  (setq bframe (eval (car blist)))
  (setq blist (cdr blist))
  ; check if (cdr arule) matches (cdr brule)
  (setq bcondlist (cdr (vref bframe cLhs)))
  next-bcondition
  (cond
    [(null bcondlist) ; || rule found
      (vset aframe cParallel (vref bframe cName))
      (vset bframe cParallel (vref aframe cName))
      (go next-arule)
    ];null
    [t   ; try next b-condition
     (setq bcondition (car bcondlist))
     (setq bcondlist (cdr bcondlist))
     (cond [(member bcondition alhs) (go next-bcondition)]
           [t (go next-brule)])
    ];t
  ];cond
 ];prog
);cond
;
; setcor : set cParallel for newly generated rules
;         (implicit: [cParallel == nil] implies parallel rule
;                    not interesting (i.e. has delta value of either
;                                     'acceptable or 'impossible
;
(defun setcor ()
 (prog (lev maxlev)
   (setq lev 1)
   (setq maxlev Lmaxlev)
   next-lev0
   (cond [(and (< lev maxlev) (vref aLevlist lev))
              (findcor lev aLevlist bLevlist t)
              (setq lev (1+ lev))
              (go next-lev0)
         ];and
   );cond
   (setq lev 1)
   next-lev1
   (cond [(and (< lev maxlev) (vref bLevlist lev))
              (findcor lev bLevlist aLevlist t)
              (setq lev (1+ lev))
              (go next-lev1)
         ];and
   );cond
```



```
 ];prog
);defun
;
; setanti : set cAnti for all newly-generated rules
;
(defun setanti ()
 (prog (lev maxlev)
   ; set cAnti values for alpha rules
   (setq lev 1)
   (setq maxlev Lmaxlev)
   next-lev0
   (cond [(and (< lev maxlev) (vref aLevlist lev))
              ; below depth limit and
              ; there are actually rules on this level
              (findanti lev aLevlist)
              (setq lev (1+ lev))
              (go next-lev0)]
   ; set cAnti for beta rules
   (setq lev 1)
   next-lev1
   (cond [(and (< lev maxlev) (vref bLevlist lev))
              ; below depth limit and
              ; there are actually rules on this level
              (findanti lev bLevlist)
              (setq lev (1+ lev))
              (go next-lev1)]))
 ];prog
);defun
;
; findanti : find cAnti values for all new rules on given level
;
(defun findanti (lev levlist)
 (prog (alist aframe blist bframe antilhs)
   (setq alist (vref levlist lev))
   next-arule
   (cond [(null alist) (return t)]) ; all rules on lev done
   (setq aframe (eval (car alist)))
   (setq alist (cdr alist))
   (cond [(vref aframe cAnti) (go next-arule)]) ; cAnti known
   ; cAnti not known
   (setq antilhs (makeanti (cdr (vref aframe cLhs))))
   ; match antilhs against other c-rules on same level
   (setq blist (vref levlist lev))
   next-brule
   (cond [(null blist) (go next-arule)]) ; no antirule for arule
                                         ; so try next arule
   (setq bframe (eval (car blist)))
   (setq blist (cdr blist))
   (cond
     [(gmatch antilhs (cdr (vref bframe cLhs))) ; antirule found
           (vset aframe cAnti (vref bframe cName))
           (vset bframe cAnti (vref aframe cName))
```

```
          (go next-arule)]
    [t (go next-brule)] ; brule not antirule, try another brule
   );cond
  );prog
);defun
;
; gmatch : match two sets of d-conditions
;         (order of d-conditions may be scrambled)
;
(defun gmatch (alist blist)
  (prog (acond)
    (cond [(nequal (length alist) (length blist)) (return nil)])
    next-acond
    (cond [(null alist) (return t)]) ; all d-conditions in alist are
                                     ; present in blist .. match
    (setq acond (car alist))
    (setq alist (cdr alist))
    (cond
      [(member acond blist) (go next-acond)] ; so far so good
                                     ; try next acond in alist
      [t (return nil)] ; one d-condition in alist missing from blist
                     ; cannot possibly match
    );cond
  );prog
);defun
;
; makeanti : create antilhs corresponding to lhs
;          (warning : lhs must not contain L(x) or SIB(x,y)
;
(defun makeanti (lhs)
  (prog (antilhs acond cond-num)
    (setq antilhs nil)
    next-acond
    (cond [(null lhs) (return antilhs)]) ; all acond in lhs changed into
                                     ; anti-cond in antilhs
    (setq acond (car lhs))
    (setq lhs (cdr lhs))
    ; find bcond in Condlist that matches acond
    (setq blist Antilist)
    next-bcond
    (cond [(null blist) ; no match .. oh-oh
          (princ "ERROR") (terpr)
          (princ "lhs     ") (princ lhs) (terpr)
          (princ "Antilist ") (princ Antilist) (terpr)
          (error)])
    (setq bpair (car blist))
    (cond
      [(equal acond (car bpair)) ; match
       ; replace acond by its anti-cond in antilhs
       (setq antilhs (append1 antilhs (cadr bpair)))
       (go next-acond)]
      [t (setq blist (cdr blist))] ; bcond doesn't match
```



```
       (go next-bcond)] ; try next bcond
    );cond
  );prog
);defun
;
; functions to update the Anticon list
; needs to run after each update of Condlist
;
; nu-anti : update Anticon list
;
(defun nu-anti ()
  (prog (acond alst)
    (setq Antilist nil)
    (setq alist Condlist)
    next-acond
    (cond [(null alist) (return t)]) ; all done ..
    (setq ccond (car alist))
    (setq alist (cdr alist))
    (setq Antilist (append1 Antilist (list ccond (energise ccond))))
    (go next-acond)
  );prog
);defun
;
; energise : creation of anticondition .
;
(defun energise (acond)
  (prog (op0 frag1 frag2 op1 var1 prop1 op2 var2 prop2)
    (setq op0 (car acond))     ; > or = or <
    (setq frag1 (cadr acond))  ; (get ? ?)
    (setq frag2 (caddr acond)) ; (get ? ?)
    ; isolate sub-particles of frag1
    (setq op1 (car frag1))
    (setq var1 (cadr frag1))
    (setq prop1 (caddr frag1))
    ; extract sub-particles of frag2
    (setq op2 (car frag2))
    (setq var2 (cadr frag2))
    (setq prop2 (caddr frag2))
    (cond
      [(and (nequal var2 'x) (nequal var2 'y))
       ; frag2 not '(get x something) or '(get y something)
       (cond [(equal var1 'x) (setq nu-var 'y)]
             [(equal var1 'y) (setq nu-var 'x)])
       (return (list op0 (list 'get nu-var prop1) frag2))]
    );cond
    ; frag2 involves state variable (includes pseudo-state)
    (cond
      [(equal prop1 prop2)
       (cond [(equal var1 var2) (error)]) ; same var of same node
       (cond
         [(equal op0 '<) (return (list '> frag1 frag2))]
         [(equal op0 '>) (return (list '< frag1 frag2))]
```

```
      ((equal op0 'equal) (return acond))
      (t (error))
     );cond
    );equal
  );cond
  ; prop1 != prop2
  (cond
    ((equal var1 var2)
     (cond ((equal var1 'x) (setq nu-var 'y))
           ((equal var2 'y) (setq nu-var 'x))
           (t (error)))
     (return (list op0 (list 'get nu-var prop1)
                       (list 'get nu-var prop2)))
    );equal
  );cond
  ; prop1 != prop2  and  var1 != var2
  (cond
    ((equal op0 '>) (return (list '< (list 'get var1 prop2)
                                     (list 'get var2 prop1))))
  );cond
  (cond
    ((equal op0 '<) (return (list '> (list 'get var1 prop2)
                                     (list 'get var2 prop1))))
  );cond
  (cond
    ((equal op0 'equal) (return (list 'equal (list 'get var1 prop2)
                                             (list 'get var2 prop1))))
  );cond
 );prog
);defun
;*****************************************************************************
;
; functions that implement the assayer
;
;*****************************************************************************
;
; assay : the assayer
;
(defun assay (var-file)
 (prog (alist asymb blist bsymb nucond-list nu-list sec-list)
   (prine "Assayer") (terpr)
   (setq nucond-list nil)
   (setq nu-list (derive var-file))
   (setq Esymb-list (append nu-list Esymb-list))
   (setq alist nu-list)
   (setq sec-list Esymb-list)
   next-asymb
   (cond ((null alist) (return nucond-list))) ; all asymb's done
   (setq asymb (car alist))
   (setq alist (cdr alist))
   (setq sec-list (cdr sec-list)) ; do not compare a symbol with itself
   (setq blist sec-list) ; compare asymb with all baymbs in blist
```



```
   next-bsymb
   (cond ((null blist) (go next-asymb)) ; all bsymb's done
   (setq bsymb (car blist))
   (setq blist (cdr blist))
   (cond
     ((equal Prbdes 'value)
       (cond ((setq nu-cond (vfndcond asymb bsymb))
              (setq nucond-list (append nucond-list nu-cond)))) )
     ((equal Prbdes 'iden)
       (cond ((setq nu-cond (ifndcond asymb bsymb))
              (setq nucond-list (append nucond-list nu-cond)))) )
     (t (error))
   );cond
   (go next-bsymb)
 );prog
);defun
;
; gen-cond : form the three conditions involving the two symbols
;
(defmacro gen-cond (symb1 symb2)
  (list 'list (list 'list ''> symb1 symb2)
              (list 'list ''< symb1 symb2)
              (list 'list ''equal symb1 symb2)
  );list
);defmacro
;
; vfndcond : find conditions in a value description
;
(defun vfndcond (asymb bsymb)
  (prog (avar aprop bvar bprop)
    (setq avar (cadr asymb))
    (setq aprop (cadr (caddr asymb)))
    (setq bvar (cadr bsymb))
    (setq bprop (cadr (caddr bsymb)))
    (cond ((and (equal avar 'qt) (equal bvar 'qt))
      ; both asymb and bsymb are problem constants
      (return nil))
    );cond
    ; at least one of asym and bsym is not a problem constant
    (cond ((and (equal avar 'qt)
                (equal bvar 'ql))
      ; one problem constant and one level constant
      (return nil))
    );cond
    (cond ((and (equal bvar 'qt)
                (equal avar 'ql))
      ; one problem constant and one level constant
      (return nil))
    );cond
    (cond ((and (equal avar 'qt)
                (member bvar '(x y)))
      ; one problem constant and one level constant
```

```
    (return (vfnd-xv-pc bsymb asymb))]
  );cond
  (cond [(and (equal bvar 'qt)
              (member avar '(x y)))
    ; one problem constant and one level constant
    (return (vfnd-xv-pc asymb bsymb))]
  );cond
  (cond [(and (equal avar 'ql)
              (equal bvar 'ql)
              (equal Prbdes 'value) ; only in a value description
                                    ; avars use qx and qy in ides
         );and
      (return (vfnd-lv-lv asymb bsymb))]
  );cond
  (cond [(and (equal avar 'ql)
              (member bvar '(x y))
         );and
      (return (vfnd-xv-lv bsymb asymb))]
  );cond
  (cond [(and (member avar '(x y))
              (equal bvar 'ql)
         );and
      (return (vfnd-xv-lv asymb bsymb))]
  );cond
  (cond [(and (member avar '(x y))
              (member bvar '(x y))
         );and
      (return (vfnd-xv-xv asymb bsymb))]
  );cond
 );prog
);defun
;
; vfnd-xv-pc : find conditions between xvar and pconst
;
(defun vfnd-xv-pc (xsymb psymb)
 (prog (rfrag rlist xprop nu-conlst pprop)
  ; form a condition only when psymb is a limit, but not a
  ; feasibility limit (already satisfied by all nodes)
  (setq xprob (cadr (caddr xsymb)))
  (setq pprop (cadr (caddr psymb)))
  (cond [(null (equal (get xprob 'xSubt) 'xrsc)) (return nil)])
  (setq rlist (get xprop 'xConsFrag))
  next-rfrag
  (cond [(null rlist) (return nil)])
  (setq rfrag (car rlist))
  (setq rlist (cdr rlist))
  (cond [(nequal (car rfrag) pprop) (go next-rfrag)])
  (cond
    [(equal (caddr rfrag) 'fcns) (return nil)]
    [(equal (caddr rfrag) 'lim)
     (setq nu-conlst (gen-cond xsymb psymb))
     (princ "assayer: ") (princ pprop)
```



```
     (princ " is a limit of ") (princ xprop) (terpr)
     (princ "new conditions: ") (princ nu-conlst) (terpr)
     (return nu-conlst)]
    [t (error)]
  );cond
 );prog
);defun
;
; vfnd-lv-lv : find conditions between lvar and lvar
;
(defun vfnd-lv-lv (asymb bsymb)
 (prog (aprop bprop asubt bsubt nu-conlst)
   ; only when one is an eview of the other
   (setq aprop (cadr (caddr asymb)))
   (setq bprop (cadr (caddr bsymb)))
   (setq asubt (get aprop 'xSubt))
   (setq bsubt (get bprop 'xSubt))
   (cond
     [(and (nequal asubt 'eview) (nequal bsubt 'eview))
      (return nil)]
   );cond
   (cond
     [(and (equal asubt 'eview) (equal bsubt 'eview))
      (return nil)]
   );cond
   (cond [(null Prbvalue) (return nil)])
   (cond [(and (equal asubt 'eview) (equal bsubt 'eret))
     (return (vlvlv-fn0 bsymb asymb bprop aprop))]
   );cond
   (cond [(and (equal asubt 'eview) (equal bsubt 'ersc))
     (return (vlvlv-fn0 bsymb asymb bprop aprop))]
   );cond
   (cond [(and (equal asubt 'eret) (equal bsubt 'eview))
     (return (vlvlv-fn0 asymb bsymb aprop bprop))]
   );cond
   (cond [(and (equal asubt 'ersc) (equal bsubt 'eview))
     (return (vlvlv-fn0 asymb bsymb aprop bprop))]
   );cond
   (return nil)
 );prog
);defun
;
; vlvlv-fn0 : find condition between a lvar and its view variable
;
(defun vlvlv-fn0 (asymb bsymb aprop bprop)
 (prog (rlist nu-conlst)
   (setq rlist (member aprop (get bprop 'xRel)))
   (cond [(null rlist) (return nil)])
   (cond [(nequal (cadr rlist) 'xDefvar) (return nil)])
   (setq nu-conlst (gen-cond asymb bsymb))
   (princ "assayer: ") (princ asymb)
   (princ " is a view of ") (princ bprop) (terpr)
```

```
    (princ "new conditions: ") (princ nu-conlst) (terpr)
    (return nu-conlst)
  );prog
);defun
;
; vfnd-xv-lv : find conditions between xvar and lvar
;
(defun vfnd-xv-lv (xsymb lsymb)
  (prog (xprop lprop xsubt lsubt nu-conlst)
    (setq xprop (cadr (caddr xsymb)))
    (setq lprop (cadr (caddr lsymb)))
    (setq xsubt (get xprop 'xSubt))
    (setq lsubt (get lprop 'xSubt))
    (cond
      ((and (equal xsubt 'shyb)
            (equal lsubt 'esum)
       )
        (return (vxvlv-fn0 xsymb lsymb xprop lprop)))
    );cond
    (cond ((and (equal xsubt 'sref) (equal lsubt 'esum))
      (return (vxvlv-fn0 xsymb lsymb xprop lprop)))
    );cond
    (cond ((and (equal xsubt 'srac) (equal lsubt 'esum))
      (return (vxvlv-fn0 xsymb lsymb xprop lprop)))
    );cond
    (cond ((and (equal xsubt 'srac) (equal lsubt 'elim))
      (return (vxvlv-fn1 xsymb lsymb xprop lprop)))
    );cond
    (return nil)
  );prog
);defun
;
; vxvlv-fn0 : find condition between state variable and summary variable
;
(defun vxvlv-fn0 (ssymb esymb sprop eprop)
  (prog (slist elist sdef nu-conlist)
    (cond ((nequal (get sprop 'xDir) (get eprop 'xDir)) (return nil)))
    (setq slist (get sprop 'xDefvar))
    (setq elist (get eprop 'xDefvar))
    next-sdef
    (cond ((null slist) (return nil)))
    (setq sdef (car slist))
    (setq slist (cdr slist))
    (cond ((null (member sdef elist)) (go next-sdef)))
    (setq nu-conlist (gen-cond ssymb esymb))
    (princ "assayer: both summary variable ") (princ eprop)
    (prine " and state variable ") (princ sprop)
    (prine " involve ") (prine sdef) (terpr)
    (princ "new conditions: ") (princ nu-conlist) (terpr)
    (return nu-conlist)
  );prog
);defun
```



```
;
; vxvlv-fn1 : find condition between svar variable and limit evar
;
(defun vxvlv-fn1 (ssymb esymb sprop eprop)
  (prog (slist sfrag nu-conlst)
    (setq slist (get sprop 'xConsFrag))
    next-sfrag
    (cond ((null slist) (return nil)))
    (setq sfrag (car slist))
    (setq slist (cdr slist))
    (cond ((nequal (car sfrag) eprop) (go next-sfrag)))
    (cond ((equal (caddr sfrag) 'feas) (go next-sfrag)))
    (setq nu-conlst (gen-cond ssymb esymb))
    (princ "assayer: ") (princ sprop) (princ " is a limit of ")
    (princ sprop) (terpr)
    (princ "new conditions: ") (princ nu-conlst) (terpr)
    (return nu-conlst)
  );prog
);defun
;
; vfnd-xv-xv : find conditions between xvar and xvar
;
(defun vfnd-xv-xv (asymb bsymb)
  (prog (aprop bprop asubt bsubt nu-conlst adir bdir)
    (setq aprop (cadr (caddr asymb)))
    (setq bprop (cadr (caddr bsymb)))
    (setq asubt (get aprop 'xSubt))
    (setq bsubt (get bprop 'xSubt))
    (setq adir (get aprop 'xDir))
    (setq bdir (get bprop 'xDir))
    (cond
      ((and
         (equal aprop bprop)
         (equal adir bdir)
         (nequal (get aprop 'xHybtype) 'alt)
       );and
         (cond
           ((equal (cadr asymb) 'x)
             (setq nu-conlst (gen-cond asymb bsymb)))
           (t (setq nu-conlst (gen-cond bsymb asymb)))
         );cond
         (princ "assayer: ") (princ asymb) (princ " and ")
         (princ bsymb) (princ " have identical properties") (terpr)
         (princ "new conditions: ") (princ nu-conlst) (terpr)
         (return nu-conlst))
    );cond
    (cond ((and (nequal asubt 'shyb) (nequal bsubt 'shyb))
      (return nil))
    );
    (cond ((and (equal asubt 'shyb) (equal bsubt 'shyb))
      (return nil))
    );cond
```

```
    ;trap for any stray values
    (error)
  );prog
);defun
;
; ifnd-xv-pc : find condition between one state variable and one
;              problem constant
;
(defun ifnd-xv-pc (xsymb csymb)
  (prog (aprop nu-conlst cprop)
    (setq aprop (cadr (caddr xsymb)))
    (setq cprop (cadr (caddr csymb)))
    (cond ((nequal (get aprop 'xSubt) 'xrac)
      (return nil))
    );cond
    (cond ((member cprop (get aprop 'xLimVar))
      (setq nu-conlst (gen-cond xsymb csymb))
      (princ "assayer: ") (princ cprop) (princ " is a limit of ")
      (princ aprop) (terpr)
      (princ "new conditions: ") (princ nu-conlst) (terpr)
      (return nu-conlst))
    );cond
    (return nil)
  );prog
);defun
;
; ifnd-xv-xv : find condition between two state variables
;
(defun ifnd-xv-xv (xsymb bsymb)
  (prog (aprop bprop asubt bsubt nu-conlst)
    (setq aprop (cadr (caddr xsymb)))
    (setq bprop (cadr (caddr bsymb)))
    (cond
      ((and (equal aprop bprop) (equal (cadr xsymb) 'x))
        (setq nu-conlst (gen-cond xsymb bsymb))
        (princ "assayer: identical property") (terpr)
        (princ "new-conditions ") (princ nu-conlst) (terpr)
        (return nu-conlst))
    );cond
    (cond
      ((and (equal aprop bprop) (equal (cadr bsymb) 'x))
        (setq nu-conlst (gen-cond bsymb xsymb))
        (princ "assayer: identical property") (terpr)
        (princ "new conditions: ") (princ nu-conlst) (terpr)
        (return nu-conlst))
    );cond
    ; aprop not equal to bprop
    (setq asubt (get aprop 'xSubt))
    (setq bsubt (get bprop 'xSubt))
    (cond ((and (equal asubt 'aret) (equal bsubt 'shyb))
      (return (ixvxv-fn0 xsymb bsymb)))
    );cond
```



```
  (cond
    ((and (equal asubt 'shyb)
          (member bprop (get aprop 'xDefvar))
          (equal adir bdir)
          (nequal (cadr xsymb) (cadr bsymb))
      );and
      (setq nu-conlst (gen-cond bsymb xsymb))
      (princ "assayer: ") (princ aprop) (princ " is a hybrid of ")
      (princ bprop) (terpr)
      (princ "new conditions: ") (princ nu-conlst) (terpr)
      (return nu-conlst))
  );cond
  (cond
    ((and (equal bsubt 'shyb)
          (member aprop (get bprop 'xDefvar))
          (equal adir bdir)
          (nequal (cadr xsymb) (cadr bsymb))
      );and
      (setq nu-conlst (gen-cond xsymb bsymb))
      (princ "assayer: ") (princ aprop) (princ " is a hybrid of ")
      (princ bprop) (terpr)
      (princ "new conditions: ") (princ nu-conlst) (terpr)
      (return nu-conlst))
  );cond
  (return nil)
 );prog
);defun
;
; ifnd-cond : find conditions for an identity representation
; WARNING : the functions implementing the assayer for an
;           identity representation may be especially buggy
;
(defun ifnd-cond (asymb bsymb)
 (prog (avar bvar)
   (setq avar (cadr asymb))
   (setq bvar (cadr bsymb))
   (cond ((and (equal avar 'qt) (equal bvar 'qt)) ; two problem constants
     (return nil))
   );cond
   (cond ((and (equal avar 'qt) (member bvar '(x y)))
     ; one state variable and one problem constant
     (return (ifnd-xv-pc bsymb asymb)))
   );cond
   (cond ((and (equal bvar 'qt) (member avar '(x y)))
     ; one state variable and one problem constant
     (return (ifnd-xv-pc asymb bsymb)))
   );cond
   (cond
     ((and (member avar '(x y)) (member bvar '(x y)))
       ; two state variables
       (return (ifnd-xv-xv asymb bsymb)))
   );cond
```

```
    (cond [(and (equal asubt 'shyb) (equal bsubt 'sret))
      (return (ixvxv-fn0 bsymb asymb))]
    );cond
    (cond [(and (equal asubt 'srsc) (equal bsubt 'shyb))
      (return (ixvxv-fn0 asymb bsymb))]
    );cond
    (cond [(and (equal asubt 'shyb) (equal bsubt 'srsc))
      (return (ixvxv-fn0 bsymb asymb))]
    );cond
    (cond [(and (equal asubt 'srsc) (equal bsubt 'esum))
      (return (ixvxv-fn1 asymb bsymb))]
    );cond
    (cond [(and (equal asubt 'esum) (equal bsubt 'srsc))
      (return (ixvxv-fn1 bsymb asymb))]
    );cond
    (cond [(and (equal asubt 'srsc) (equal bsubt 'elim))
      (return (ixvxv-fn2 asymb bsymb))]
    );cond
    (cond [(and (equal asubt 'elim) (equal bsubt 'srsc))
      (return (ixvxv-fn2 bsymb asymb))]
    );cond
    ; all other conditions not known to yield useful conditions
    (return nil)
  );prog
);defun
;
; ixvxv-fn0 : find conditions between sret/srsc and shyb
;
(defun ixvxv-fn0 (baymb haymb)
  (prog (bprop hprop nu-conlst)
    ; only acceptable if sret/srsc is part of shyb's definition
    ; and between different nodes
    (cond [(equal (cadr bsymb) (cadr hsymb))
      (return nil)]
    );cond
    (setq bprop (cadr (caddr bsymb)))
    (setq hprop (cadr (caddr hsymb)))
    (cond [(null (member bprop (get hprop 'xDefvar)))
      (return nil)]
    );cond
    ; sret/srsc part of shyb's defn and different nodes
    (cond
      [(equal (cadr bsymb) 'x)
        (setq nu-conlst (gen-cond bsymb hsymb))
        (princ "assayer: ") (princ hprop)
        (princ " is a hybrid state variable involving ")
        (princ bprop) (terpr)
        (princ "new conditions: ") (princ nu-conlst) (terpr)
        (return nu-conlst)]
      [(equal (cadr hsymb) 'x)
        (setq nu-conlst (gen-cond hsymb bsymb))
        (princ "assayer: ") (princ hprop)
        (princ " is a hybrid state variable involving ")
        (princ bprop) (terpr)
        (princ "new conditions: ") (princ nu-conlst) (terpr)
        (return nu-conlst)]
      [t (error)]
    );cond
  );prog
);defun
;
; ixvxv-fn1 : find conditions between a srsc and a esum
;
(defun ixvxv-fn1 (rsymb asymb)
  (prog (rprop sprop rdefn sdefn nu-conlst)
    ; only if they share a common variable in their definitions,
    ; have opposite directions, and involve the same nodes
    (cond [(nequal (cadr rsymb) (cadr asymb))
      (return nil)]
    );cond
    (setq rprop (cadr (caddr rsymb)))
    (setq sprop (cadr (caddr asymb)))
    (cond [(nequal (get rprop 'xDir) (get sprop 'xDir))
      (return nil)]
    );cond
    (setq rdefn (get rprop 'xDefvar))
    (setq sdefn (get sprop 'xDefvar))
    next-rentry
    (cond [(null rdefn) (return nil)])
    (setq rentry (car rdefn))
    (setq rdefn (cdr rdefn))
    (cond [(null (member rentry sdefn))
      (go next-rentry)]
    );cond
    ; rsymb and asymb share a common variable in their defn
    (setq nu-conlst (gen-cond rsymb asymb))
    (princ "assayer: summary variable ") (princ sprop)
    (princ " and state variable ") (princ rprop)
    (princ " involves ") (princ rentry) (terpr)
    (princ "new conditions: ") (princ nu-conlst) (terpr)
    (return nu-conlst)
  );prog
);defun
;
; ixvxv-fn2 : find conditions between a srsc anf a elim
;
(defun ixvxv-fn2 (rsymb lsymb)
  (prog (rprop lprop nu-conlst)
    (cond [(nequal (cadr rsymb) (cadr lsymb))
      (return nil)]
    );
    (setq rprop (cadr (caddr rsymb)))
    (setq lprop (cadr (caddr lsymb)))
    (cond [(null (member lprop (get rprop 'xLimVar)))
```



```
      (princ " is a hybrid state variable involving ")
      (princ bprop) (terpr)
      (princ "new conditions: ") (princ nu-conlst) (terpr)
      (return nu-conlst)]
    [t (error)]
  );cond
);prog
);defun
;
; ixvxv-fn1 : find conditions between a srsc and a esum
;
(defun ixvxv-fn1 (rsymb asymb)
  (prog (rprop sprop rdefn sdefn nu-conlst)
    ; only if they share a common variable in their definitions,
    ; have opposite directions, and involve the same nodes
    (cond [(nequal (cadr rsymb) (cadr asymb))
      (return nil)]
    );cond
    (setq rprop (cadr (caddr rsymb)))
    (setq sprop (cadr (caddr asymb)))
    (cond [(nequal (get rprop 'xDir) (get sprop 'xDir))
      (return nil)]
    );cond
    (setq rdefn (get rprop 'xDefvar))
    (setq sdefn (get sprop 'xDefvar))
    next-rentry
    (cond [(null rdefn) (return nil)])
    (setq rentry (car rdefn))
    (setq rdefn (cdr rdefn))
    (cond [(null (member rentry sdefn))
      (go next-rentry)]
    );cond
    ; rsymb and asymb share a common variable in their defn
    (setq nu-conlst (gen-cond rsymb asymb))
    (princ "assayer: summary variable ") (princ sprop)
    (princ " and state variable ") (princ rprop)
    (princ " involves ") (princ rentry) (terpr)
    (princ "new conditions: ") (princ nu-conlst) (terpr)
    (return nu-conlst)
  );prog
);defun
;
; ixvxv-fn2 : find conditions between a srsc anf a elim
;
(defun ixvxv-fn2 (rsymb lsymb)
  (prog (rprop lprop nu-conlst)
    (cond [(nequal (cadr rsymb) (cadr lsymb))
      (return nil)]
    );
    (setq rprop (cadr (caddr rsymb)))
    (setq lprop (cadr (caddr lsymb)))
    (cond [(null (member lprop (get rprop 'xLimVar)))
```

```
    (return nil)]
   );cond
   ; lsymb is a limit for rsymb but not a feasibility limit
   ; (which is satisfied by all feasible nodes)
   (setq nu-conlst (gen-cond rsymb lsymb))
   (princ "assayer: ") (princ lprop) (princ " is a limit of ")
   (princ rprop) (terpr)
   (princ "new conditions: ") (princ nu-conlst) (terpr)
   (return nu-conlst)
  );prog
);defun
;*************************************************************************
;
; functions that implement the derivator
;
;*************************************************************************
;
; derive : the derivator
;
(defun derive (var-file)
  (prog (alist blist nu-list aprop aslot avalue iport num-frames num-slots)
    (princ "Derivator") (terpr)
    (cond [(or Da-list Dba-list Dbb-list) (go find-conds)])
    ; initial case -> read in basic variables
    (setq iport (infile var-file))
    (princ "derivator: reading set of initial variables") (terpr)
    (setq num-frames (read iport))
    next-frame
    (cond [(zerop num-frames) (close iport)
                              (return (grow Da-list))]
    );cond
    (setq num-frames (1- num-frames))
    (setq aprop (read iport))
    (setq Da-list (cons aprop Da-list))
    (putprop aprop aprop 'xName)
    (setq num-slots 0)
    next-slot
    (cond [(zerop num-slots) (dump-prop aprop)
                             (go next-frame)])
    (setq aslot (read iport))
    (setq avalue (read iport))
    (putprop aprop avalue aslot)
    (setq num-slots (1- num-slots))
    (go next-slot)
    ;
    ; post-initial case -> try to derive
    ;
    find-conds
    (setq Dbb-list (append Dbb-list Dba-list))
    (setq Dba-list Da-list)
    (setq Da-list nil)
    (setq alist Dba-list)
```



```
    (setq blist (cdr (append Dba-list Dbb-list)))
    next-asymb
    (cond [(null alist) (calc-nuprop Da-list)
                        (return (grow Da-list))])
    (self-def (car alist))
    (cross-def (car alist) blist)
    (setq alist (cdr alist))
    (setq blist (cdr blist))
    (go next-asymb)
   );prog
);cond
;
; self-def : derive variable from one existing variable
;          (i.e. summary variable / view variable )
;
(defun self-def (aprop)
  (prog (atype asubt)
    (cond
      [(member (get aprop 'xSubt) '(eret esc))
       (make-sum aprop)
       (cond [(and (equal Prbdes 'value) Prbview)
         (make-view aprop)]
       );cond
      ]
    );cond
  );prog
);defun
;
; makesum : create a new summary variable
;
(defun make-sum (aprop)
  (prog (nu-name)
    (setq nu-name (concat 'R aprop))
    ; fill in its properties
    (putprop nu-name nu-name 'xName)
    (putprop nu-name 'evar 'xType) ; type evar (entity variable)
    (putprop nu-name 'esum 'xSubt) ; subtype esum (summary variable)
    (putprop nu-name 'fore 'xDir)   ; direction fore (future decisions)
    (putprop
      nu-name
      (list '- (list 'get 'qp (list 'quote nu-name))
               (list 'get 'QVar (list 'quote aprop)))
      'xDef) ; defining code
    (putprop nu-name (list nu-name aprop) 'xDefvar) ; variables in defn
    (putprop
      nu-name
      (list 'nsum (list 'get 'qt (list 'quote aprop)))
      'xInit) ; initialization code
    (putprop nu-name (list aprop 'xDefvar) 'xRel) ; related variables
    (setq Da-list (cons nu-name Da-list)) ; add to list of properties
    (princ "derivator: new summary variable from ")
    (princ aprop) (terpr)
```

```
    (dump-prop nu-name)
    (return nu-name)
  );prog
);defun
;
; makeview : create a new view variable
;
(defun make-view (aprop)
  (prog (nu-name)
    (setq nu-name (concat 'V aprop))
    ; fill in its properties
    (putprop nu-name nu-name 'xName)
    (putprop nu-name 'evar 'xType) ; type evar (entity variable)
    (putprop nu-name 'eview 'xSubt) ; subtype eview (view variable)
    (putprop nu-name 'fore 'xDir)   ; direction fore (future decision)
    (putprop
      nu-name
      (list 'nth (list '1 + (list 'get 'ql ''xLev))
            (list 'get 'qt (list 'quote aprop)))
      'xDef) ; defining code
    (putprop nu-name (list aprop) 'xDefvar) ; only variables in formula
                                    ; no implied variables
    (putprop nu-name
      (list  'cadr (list 'get 'qt (list 'quote aprop)))
      'xInit) ; initialization code
    (putprop nu-name (list aprop 'xDefvar) 'xRel)
    (setq Da-list (cons nu-name Da-list))
    (princ "derivator: new view variable from ") (princ aprop) (terpr)
    (dump-prop nu-name)
    (return nu-name)
  );prog
);defun
;
; grow : expand list of variables into instances
;       variable R -> (get x (quote R)), (get y (quote R))
;       constant K -> K
;
(defun grow (alist)
  (prog (nu-list asymb)
    (setq nu-list nil)
    next-asymb
    (cond ((null alist) (return nu-list)))
    (setq asymb (car alist))
    (setq alist (cdr alist))
    (cond
      ((equal (get asymb 'xType) 'pconst) ; asymb is constant
        (setq nu-list (append1 nu-list
                  (list 'get 'qt (list 'quote asymb)))))
      ((and (equal (get asymb 'xType) 'evar) (equal Prbdes 'value))
        (setq nu-list (append1 nu-list
                        (list 'get 'ql (list 'quote asymb)))))
      (t                            ; asymb is a variable
```



```
        (setq nu-list (append1 nu-list
                  (list 'get 'x (list 'quote asymb))))
        (setq nu-list (append1 nu-list
                  (list 'get 'y (list 'quote asymb)))))
      );cond
      (go next-asymb)
    );prog
);defun
;
; calc-nuprop : compute values of new properties for the search nodes
;
(defun calc-nuprop (plist)
  (prog (prop instlist inst node-list tframe tlev qt ql qx formula
         iformula lev-index root node qp)
    (princ "Computing values of new symbols for problem instances")
    (terpr)
    next-prop
    (cond ((null plist) (return t)))
    (setq prop (car plist))
    (setq plist (cdr plist))
    (setq formula (get prop 'xDef))
    (setq iformula (get prop 'xInit))
    (setq instlist Inst-lst)
    next-inst
    (cond ((null instlist) (go next-prop)))
    (setq inst (car instlist))
    (setq instlist (cdr instlist))
    (putprop inst (cons prop (get inst 'proplist)) 'proplist)
    (setq qt inst)
    (setq tlev (get inst 'levlist))
    (cond
      ((or (nequal (get inst 'type) 'value)
           (nequal (get prop 'xType) 'evar)
       )
       (go process-var))
    );cond
    (setq ql (car tlev))
    (setq tlev (cdr tlev))
    (putprop ql (eval iformula) prop)
    next-ql
    (cond ((null tlev) (go next-inst)))
    (setq qp ql)
    (setq ql (car tlev))
    (setq tlev (cdr tlev))
    (putprop ql (eval formula) prop)
    (go next-ql)
    ;
    ; augment variable set of each node with new variable
    ;
    process-var
    (setq tframe (eval (get inst 'tree)))
    (setq lev-index (cdr (vref tframe tNodelst)))
```

```
;
; specially process the root of the search tree
;
(setq node-list (car lev-index))
(setq lev-index (cdr lev-index))
(setq root (cadr node-list))
(setq qx root)
(setq ql (car tlev))
(setq tlev (cdr tlev))
(putprop root (eval ilformula) prop)
;
; process other levels in search tree
;
next-lev
(cond ((null lev-index) (go next-inst)))
(setq node-list (cdar lev-index))
(setq lev-index (cdr lev-index))
(setq ql (car tlev))
(setq tlev (cdr tlev))
; process nodes on this level
next-node
(cond ((null node-list) (go next-lev)))
(setq node (car node-list))
(setq node-list (cdr node-list))
(setq qx node)
(setq qp (get node 'Pa))
(putprop node (eval formula) prop)
(go next-node)
);prog
);defun
;
; cross-def : define new property by combining two properties
;
(defun cross-def (aprop blist)
 (prog (bprop atype btype)
  (setq atype (get aprop 'xType))
  next-bprop
  (cond ((null blist) (return t)))
  (setq bprop (car blist))
  (setq blist (cdr blist))
  (setq btype (get bprop 'xType))
  (cond ((and (equal atype 'pconst) (equal btype 'svar))
    (sp-make bprop aprop)
    (go next-bprop))
  );cond
  (cond ((and (equal atype 'svar) (equal btype 'svar))
    (ee-make aprop bprop)
    (go next-bprop))
  );cond
  (cond ((and (equal atype 'svar) (equal btype 'svar))
    (se-make bprop aprop)
    (go next-bprop))
```



```
  );cond
  (cond ((and (equal atype 'svar) (equal btype 'pconst))
    (sp-make aprop bprop)
    (go next-bprop))
  );cond
  (cond ((and (equal atype 'svar) (equal btype 'svar))
    (se-make aprop bprop)
    (go next-bprop))
  );cond
  (go next-bprop)
 );prog
);defun
;
; sp-make : make new property, if any,
;         between a svar and a pconst
;
(defun sp-make (sprop cprop)
 (prog ()
  ; nothing between a pconst and a srst/shyb
  (cond ((equal (get sprop 'xSubt) 'srsc)
    (return nil))
  );cond
  ; sprop is a srsc
  (cond ((member cprop (get sprop 'xRel))
    (return (sp-fn0 sprop cprop)))
  );cond
  (return nil)
 );prog
);defun
;
; sp-fn0 : actually create the variable between the pconst
;          and the svar
;
(defun sp-fn0 (sprop cprop)
 (prog (nu-name)
  (setq nu-name (concat cprop '- sprop))
  (putprop nu-name nu-name 'xName)
  (putprop nu-name 'svar 'xType)
  (putprop nu-name 'shyb 'xSubt)
  (putprop nu-name 'alt 'xHybtype)
  (putprop nu-name 'fore 'xDir)
  (putprop nu-name
   (list '- (list 'get 'qt (list 'quote cprop))
            (list 'get 'qx (list 'quote sprop)))
   'xDef
  )
  (putprop nu-name (fnd-defvar sprop cprop) 'xDefvar)
  (putprop nu-name nil 'xInit) ; a hack that works
  ; no xOptFrag - don't know how to do this
  ; no xConsFrag - don't know how to do this, also
  (putprop nu-name (list sprop 'xDefvar cprop 'xDefvar) 'xRel)
  (dump-prop nu-name)
```

```
    (setq Da-list (cons nu-name Da-list))
    (return nu-name)
  );prog
);defun
;
; fnd-defvar : returns the list of variables involved in the definition
;              of the new variable
;
(defun fnd-defvar (aprop bprop)
  (prog (dlist dprop plist)
    (setq dlist (list aprop bprop))
    ; now add those variables present in xDefvar of aprop and bprop
    ; if not already present in dlist
    (setq plist (get aprop 'xDefvar))
    nextp0
    (cond ((null plist) (go done)))
    (setq dprop (car plist))
    (setq plist (cdr plist))
    (cond ((member dprop dlist) t)
          (t (setq dlist (cons dprop dlist))))
    );cond
    (go nextp0)
    done
    (setq plist (get bprop 'xDefvar))
    nextp1
    (cond ((null plist) (return dlist)))
    (setq dprop (car plist))
    (setq plist (cdr plist))
    (cond ((member dprop dlist) t)
          (t (setq dlist (cons dprop dlist))))
    );cond
    (go nextp1)
  );prog
);defun
;
; fnd-rel : copy related variables in xDefvar, xOptFrag, and xConsFrag
;           into xRel and annotate with source
;
(defun fnd-rel (nu-prop)
  (prog (rel-list)
    (setq rel-list nil)
    (mapc
      '(lambda (cand-prop)
        (cond
          ((member cand-prop rel-list) t)
          (t (setq rel-list (cons 'xDefvar rel-list))
             (setq rel-list (cons cand-prop rel-list))))
        );cond
      );lambda
      (get nu-prop 'xDefvar)
    );mapc
    ; derived vars do not have OptFrag and ConsFrag right now
```



```
    (return rel-list)
  );prog
);defun
;
; se-make : make new variables, if any, between a svar and a evar
;
(defun se-make (sprop eprop)
  (prog (ssubt esubt)
    (setq ssubt (get sprop 'xSubt))
    (setq esubt (get eprop 'xSubt))
    (cond ((and (equal ssubt 'sret) (equal esubt 'esum))
      (return (se-fn0 sprop eprop))))
    );cond
    (cond ((and (equal ssubt 'srsc) (equal esubt 'esum))
      (return (se-fn0 sprop eprop))))
    );cond
    (cond ((and (equal ssubt 'srsc) (equal esubt 'elim))
      (return (se-fn0 sprop eprop 'xConsFrag))))
    );cond
    (return nil)
  );prog
);defun
;
; se-fn0 : create new variable between the svar and the evar
;
(defun se-fn0 (svar-prop esum-prop)
  (prog (slist elist sentry nu-name)
    (setq slist (get svar-prop 'xDefvar))
    (setq elist (get esum-prop 'xDefvar))
    next-sentry
    (cond ((null slist) (return nil)))
    (setq sentry (car slist))
    (setq slist (cdr slist))
    (cond ((null (member sentry elist)) (go next-sentry)))
    (setq nu-name (concat svar-prop '+ esum-prop))
    (putprop nu-name nu-name 'xName)
    (putprop nu-name 'svar 'xType)
    (putprop nu-name 'shyb 'xSubt)
    (cond
      ((equal Prbdes 'value) (putprop nu-name 'alt 'xHybtype))
      ((equal Prbdes 'iden) (putprop nu-name nil 'xHybtype))
      (t (princ "Prbdes ") (princ Prbdes) (terpr) (error)))
    );cond
    (putprop nu-name 'aft 'xDir) ;  sprop = sret/srsc (aft direction)
    (putprop nu-name
      (list ' + (list 'get 'qx (list 'quote svar-prop))
                (list 'get 'QVar (list 'quote esum-prop)))
      'xDef)
    (putprop nu-name (fnd-defvar svar-prop esum-prop) 'xDefvar)
    (putprop nu-name nil 'xInit)
    (putprop nu-name (fnd-rel nu-name) 'xRel)
    (setq Da-list (cons nu-name Da-list))
```

```
  (dump-prop nu-name)
  (return nu-name)
 );prog
);defun
;
; ee-make : make new variables, if any, between two evar's
;
(defun ee-make (aprop bprop)
 (prog (asubt bsubt nu-prop nu-list)
  (setq asubt (get aprop 'xSubt))
  (setq bsubt (get bprop 'xSubt))
  (cond
   ((and (equal asubt 'eret) (equal bsubt 'eret))
    (return (ee-fn0 aprop bprop 'xOptFrag))
   )
  );cond
  (cond
   ((and (equal asubt 'eret) (equal bsubt 'ersc))
    (return (ee-fn1 aprop bprop))
   )
  );cond
  (cond
   ((and (equal asubt 'ersc) (equal bsubt 'eret))
    (return (ee-fn1 bprop aprop))
   )
  );cond
  (cond
   ((and (equal asubt 'ersc) (equal bsubt 'ersc))
    (return (ee-fn0 aprop bprop 'xConsFrag))
   )
  );cond
  (cond
   ((and (equal asubt 'ersc) (equal bsubt 'elim))
    (return (ee-fn0 aprop bprop 'xConsFrag))
   )
  );cond
  (cond
   ((and (equal asubt 'elim) (equal bsubt 'ersc))
    (return (ee-fn0 bprop aprop 'xConsFrag))
   )
  );cond
  (cond
   ((and (equal asubt 'etwo) (equal bsubt 'etwo)) ;etwo = eret & eret
    (ee-fn0 aprop bprop 'xOptFrag)
    (ee-fn0 aprop bprop 'xConsFrag)
    (ee-fn1 bprop aprop)
    (return)
   )
  );cond
  (cond
   ((and (equal asubt 'etwo) (equal bsubt 'eret))
    (ee-fn0 aprop bprop 'xOptFrag)
```



```
    (ee-fn1 bprop aprop)
    (return)
   )
  );cond
  (cond
   ((and (equal asubt 'etwo) (equal bsubt 'ersc))
    (ee-fn0 aprop bprop 'xConsFrag)
    (ee-fn1 bprop aprop)
    (return)
   )
  );cond
  (cond
   ((and (equal asubt 'eret) (equal bsubt 'etwo))
    (ee-fn0 aprop bprop 'xOptFrag)
    (ee-fn1 aprop bprop)
    (return)
   )
  );cond
  (cond
   ((and (equal asubt 'ersc) (equal bsubt 'etwo))
    (ee-fn0 aprop bprop 'xConsFrag)
    (ee-fn1 aprop bprop)
    (return)
   )
  );cond
  (cond
   ((and (equal asubt 'etwo) (equal bsubt 'elim))
    (ee-fn0 aprop bprop 'xConsFrag)
    (return)
   )
  );cond
  (cond
   ((and (equal asubt 'elim) (equal bsubt 'etwo))
    (ee-fn0 bprop aprop 'xConsFrag)
    (return)
   )
  );cond
 );prog
);defun
;
; ee-fn0
;
(defun ee-fn0 (aprop bprop slot)
 (prog (alist afrag comb)
  (setq alist (get aprop slot))
  next-afrag
  (cond ((null alist) (return nil)))
  (setq afrag (car alist))
  (setq alist (cdr alist))
  (cond ((null (member bprop afrag)) (go next-afrag)))
  (setq comb (car afrag))
  (setq nu-name (concat aprop comb bprop))
```

```
  (putprop nu-name nu-name 'xName)
  (putprop nu-name 'evar 'xType)
  (cond ((equal (get aprop 'xSubt) (get bprop 'xSubt))
         (putprop nu-name (get aprop 'xSubt) 'xSubt))
        (t (putprop nu-name 'chyb 'xSubt))
  );cond
  (putprop nu-name
    (list comb (list 'get QVar (list 'quote aprop))
               (list 'get QVar (list 'quote bprop)))
    'xDef)
  (putprop nu-name (list aprop bprop) 'xDefvar) ; from 2 basic vars
  (putprop nu-name nil 'xInit)
  ; don't know how to derive ConsFrag and OptFrag for these
  (putprop nu-name (list aprop 'xDefvar bprop 'xDefvar) 'xRel) ; 2 basic vars
  (setq Da-list (cons nu-name Da-list))
  (dump-prop nu-name)
  (return nu-name)
 );prog
);defun
;
; ee-fn1
;
(defun ee-fn1 (ret-prop rsc-prop)
 (prog (nu-name)
  ; hard-coded to produce the ratio
  (setq nu-name (concat ret-prop '/ rsc-prop))
  (putprop nu-name nu-name 'xName)
  (putprop nu-name 'evar 'xType)
  (putprop nu-name 'chyb 'xSubt)
  (putprop nu-name
    (list 'quotient (list 'get QVar (list 'quote ret-prop))
                    (list 'get QVar (list 'quote rsc-prop)))
    'xDef)
  (putprop nu-name (list ret-prop rsc-prop) 'xDefvar) ; 2 basic vars
  (putprop nu-name nil 'xInit)
  ; no initialization
  ; no OptFrag nor ConsFrag
  (putprop nu-name (list ret-prop 'xDefvar rsc-prop 'xDefvar) 'xRel) ; 2 basic vars
  (setq Da-list (cons nu-name Da-list))
  (dump-prop nu-name)
  (return nu-name)
 );prog
);defun
;
; dump-prop : dump out a property frame
;
(defun dump-prop (aprop)
 (princ (get aprop 'xName)) (tab 10)
 (princ "xType ") (princ (get aprop 'xType)) (tab 40)
 (princ "xSubt ") (princ (get aprop 'xSubt)) (terpr)
 (tab 10) (princ "xDef ") (princ (get aprop 'xDef)) (tab 65)
 (princ "xInit ") (princ (get aprop 'xInit)) (terpr)
```



```
 (tab 10) (princ "xOptFrag ") (princ (get aprop 'xOptFrag)) (tab 50)
 (princ "xConFrag ") (princ (get aprop 'xConsFrag)) (terpr)
 (tab 10) (princ "xDefvar ") (princ (get aprop 'xDefvar)) (tab 50)
 (princ "xRel ") (princ (get aprop 'xRel)) (terpr)
);defun
;*****************************************************************
;
; initialization routines
;
;*****************************************************************
;
; init : carry out initialization
;
(defun init (desc-file)
 (princ "Initialization") (terpr)
 ; set size and elements of cframe (a c-rule)
 (setq cVsize 16)
 (setq cName 0)
 (setq cType 1)
 (setq cLhs 2)
 (setq cCond 3)
 (setq cParallel 4)
 (setq cAnti 5)
 (setq cParent 6)
 (setq cChildren 7)
 (setq cDelta 8)
 (setq cPoslst 9)
 (setq cNeglst 10)
 (setq cNullst 11)
 (setq cNumpos 12)
 (setq cNumneg 13)
 (setq cNumnul 14)
 (setq cEv 15)
 ; set elements of a tframe (tree frame)
 (setq tName 0)
 (setq tNodelst 1)
 ;
 (inst-form desc-file)
 ;
 ;
 ; kludge to generate root of rule graphs and to ensure that the
 ; c-rules on the first level only involve the return variables
 ; IMPORTANT : return state variable must be named R()
 ;
 (setq Condlist nil)
 (setq oneCondlist '((> (get x 'R) (get y 'R))
                     (< (get x 'R) (get y 'R))
                     (equal (get x 'R) (get y 'R))))
 ;
 (setq ca0 (new-vector 16))
 (vset ca0 cName 'ca0)
 (vset ca0 cType 'alpha)
```

```
(vset ca0 cLhs '((Sibling x y)))
(vset ca0 cCond nil)
(vset ca0 cParallel 'cb0)
(vset ca0 cAnti 'ca0)
(vset ca0 cParent nil)
(vset ca0 cChildren nil)
(vset ca0 cDelta 'no)
(vset ca0 cPoslst '( + ))
(vset ca0 cNeglst '(-))
(vset ca0 cNullst '(0))
(vset ca0 cNumpos 0)
(vset ca0 cNumneg 0)
(vset ca0 cNumnul 0)
;
(setq cb0 (new-vector 16))
(vset cb0 cName 'cb0)
(vset cb0 cType 'beta)
(vset cb0 cLhs '((equal (get x 'L) (get y 'L))))
(vset cb0 cCond nil)
(vset cb0 cParallel 'ca0)
(vset cb0 cAnti 'cb0)
(vset cb0 cParent nil)
(vset cb0 cChildren nil)
(vset cb0 cDelta 'no)
(vset cb0 cPoslst '( + ))
(vset cb0 cNeglst '(-))
(vset cb0 cNullst '(0))
(vset cb0 cNumpos 0)
(vset cb0 cNumneg 0)
(vset cb0 cNumnul 0)
;
; initialize assorted global variables
;
(setq Activelist nil)
(setq aLevlist (new-vector 128))
(vset aLevlist 0 '(ca0))
(initsym '(ca 0))
(setq Cdumplist nil)
(makelev 0 'ca aLevlist oneCondlist 2)
(setq bLevlist (new-vector 128))
(vset bLevlist 0 '(cb0))
(initsym '(cb 0))
(makelev 0 'cb bLevlist oneCondlist 1)
;
(setq Lmaxlev 3)
(setq Antilist nil)
;
(setq Da-list nil)
(setq Dba-list nil)
(setq Dbb-list nil)
(setq Prbdes 'value)
(setq Prbview nil)
```



```
 (setq QVar 'qi)
 (setq Exymb-list nil)
)
;
; cdump0 : print summary of a c-rule
;
(defun cdump0 (cname)
 (prog (cframe)
  (setq cframe (eval cname))
  (princ (vref cframe cName))
  (tab 7) (princ "LHS : ") (princ (vref cframe cLhs)) (terpr)
  (tab 7) (princ "Parallel : ") (princ (vref cframe cParallel))
  (tab 23) (princ "Anti : ") (princ (vref cframe cAnti))
  (tab 42) (princ "Parent : ") (princ (vref cframe cParent))
  (tab 60) (princ "Children : ") (princ (vref cframe cChildren)) (terpr)
 )
)
;
; cdump1 : print a different summary of a c-rule
;
(defun cdump1 (cframe)
 (tab 8) (princ "LHS : ") (princ (vref cframe cLhs)) (terpr)
 (tab 8) (princ "#{+ve ex} : ") (princ (vref cframe cNumpos))
 (tab 28) (princ "#{-ve ex} : ") (princ (vref cframe cNumneg))
 (tab 48) (princ "#{null ex} : ") (princ (vref cframe cNumnul))
 (terpr)
)
;
; inst-form : create the frames for the problem instances
;
;          assumes data file of form (illustrated by 0/1-knapsack):
;            1  ; # of problem instances in data file
;            i0 ; name of problem instance
;            7  ; # of properties of problem instance
;            r    (1 4 3 2 5) ; property #1 and its value
;            w    (3 4 2 1 4) ; #2
;            K    8           ; #3
;            numitems  5  ; #4 - #entities in problem
;            proplist  (r w) ; #5 - entity properties
;            type value  ; #6 - description type
;            tree  t0    ; #7 - name of file holding search tree
;
(defun inst-form (desc-file)
 (prog (iport numinst instname numslots iprop ivalue numitems
        llist instlist inst tframe tree)
  (setq Inst-lst nil)
  (setq iport (infile desc-file))
  (setq numinst (read iport))
  next-inst0
  (cond ((zerop numinst) (go ingest-inst)))
  (setq numinst (1- numinst))
  (setq instname (read iport))
```

```
    (princ "reading instance ") (princ instname) (terpr)
    (setq inst-lst (cons instname inst-lst))
    (setq numslots (read iport))
    (putprop instname instname 'name)
    next-slot
    (cond ((zerop numslots) (go cleanup-inst))
    (setq iprop (read iport))
    (setq ivalue (read iport))
    (putprop instname ivalue iprop)
    (setq numslots (1- numslots))
    (go next-slot)
    cleanup-inst
    (setq numitems (get instname 'numitems))
    (setq llist nil)
    next-item
    (cond ((minusp numitems) (putprop instname llist 'levlist)
                             (go next-inst0))
    );cond
    (setq llist (cons (concat instname 'l numitems) llist))
    (setq numitems (1- numitems))
    (go next-item)
    ;
    ; transform these frames into working data structures
    ;
    ingest-inst
    (setq instlist inst-lst)
    next-inst1
    (cond ((null instlist) (return t))
    (setq inst (car instlist))
    (setq instlist (cdr instlist))
    ;
    ; first, create tree frame and read in the search tree
    ;
    (setq tree (get inst 'tree))
    (set tree (new-vector 4))
    (setq tframe (eval tree))
    (vset tframe tName tree)
    (princ "reading search tree ") (princ tree) (terpr)
    (r3 inst tframe)
    ;
    ; now process the entity variables
    ;
    (cond
      ((equal (get inst 'type) 'value)
        (value-ingest inst))
      ((equal (get inst 'type) 'iden)
        (iden-ingest inst))
      (t (error))
    );cond
    (go next-inst1)
  );prog
);defun
```



```
;
; value-ingest : ingest the information in the instance frame into
;                the working data structures
;
(defun value-ingest (inst)
  (prog (levlist levnum proplist prop)
    (setq levlist (get inst 'levlist))
    (setq levname (car levlist))
    (setq levlist (cdr levlist))
    (putprop levname levname 'name)
    (putprop levname 0 'level)
    (mapc
      '(lambda (x)
        (putprop levname nil x)
      );lambda
      (get inst 'proplist)
    );mapc
    (setq levnum 0)
    next-levname
    (cond ((null levlist) (return t)))
    (setq levname (car levlist))
    (setq levlist (cdr levlist))
    (setq proplist (get inst 'proplist))
    (putprop levname levname 'name)
    (putprop levname (1 + levnum) 'level)
    next-prop
    (cond
      ((null proplist)
        (setq levnum (1 + levnum))
        (go next-levname))
    );cond
    (setq prop (car proplist))
    (setq proplist (cdr proplist))
    (putprop levname (nth levnum (get inst prop)) prop)
    (go next-prop)
  );prog
);defun
;
; iden-ingest : digest the entity variables in an identity description
;               augment variables in each node
;
(defun iden-ingest (inst)
  (prog (tframe)
    (setq tframe (eval (get inst 'tree)))
    (mapc
      '(lambda (property)
        (prog (tlist)
          (setq tlist (cddr (vref tframe tNodelst))) ; skips root
          next-set
          (cond ((null tlist) (return t)))
          (mapc
            '(lambda (node)
```

```
        (putprop node (get (get node 'Op) property) property)
       );lambda
       (cdar tlist)
      );mapc
      (setq tlist (cdr tlist))
      (go next-set)
     );prog
    );lambda
    (get inst 'proplist)
   );mapc
  );prog
);cond
;
; nsum : utility function to find sm of numbers in a list
;
(defun nsum (alist)
 (prog (sum entry)
  (setq sum 0)
  next-entry
  (cond [(null alist) (return sum)])
  (setq entry (car alist))
  (setq alist (cdr alist))
  (setq sum ( + sum entry))
  (go next-entry)
 );prog
);defun
;
; r3 : read in search tree
;
(defun r3 (inst tframe)
 (prog (iport num name nodelst levlst llist ql plist)
  (setq lev 0)
  (setq name (vref tframe tName))
  (setq llist (get name 'levlist))
  (setq iport (infile name))
  (setq plist (read iport)) ; set of variables for each node
  (setq nodelst nil)
  nxtlev
  (setq num (read iport))   ; # of nodes in current level
                            ; also first element of list
  (cond
   [(null num) t]           ; at eof
   [t (setq levlst (list lev))
    (setq ql (car llist))
    (setq llist (cdr llist))
    (prog ()
     nxtnode
     (cond
      [(equal num 0) t]     ; all nodes on level k read
      [t (setq node (read iport))
       (putprop node node 'N)
       (mapc
```



```
        '(lambda (x)
          (putprop node (read iport) x)
         );lambda
         plist
        );mapc
        (putprop node inst 'qt)
        (putprop node ql 'ql)
        (setq num (1- num))
        (setq levlst (append1 levlst node))
        (go nxtnode)            ; go to next node
       ]
      );cond
      ; all nodes on level lev read in
      (setq nodelst (append1 nodelst levlst))
      (setq lev (1 + lev))
      (go nxtlev)               ; go to next level
     );prog
    ];t
   );cond
   (close iport)            ; all done
   (vset tframe tNodelst (append (list lev) nodelst))
  );prog
);defun
;
;
(defun Sibling (x y)
 (prog ()
  (cond [(equal x (get y 'Si)) (return t)]
        [t (return nil)])
 )
);defun
```

Appendix B : A Trace for the 0/1-Knapsack Problem

This appendix describes one data set supplied to the searcher and the resulting trace produced by the searcher. The data supplied to the searcher is distributed over four files: knap, des, t0, and t1. The file knap (Figure A1) describes the symbols in the value-description of the 0/1-knapsack problem. The first entry in this file is the number of symbols. Each symbol has ten entries. The first entry for each symbol is its name. Each of the remaining nine entries describe a slot name and its corresponding slot filler.

The file des (Figure A2) describes the 0/1-knapsack problem instances. The first entry in this file is the number of problem instances. Each problem instance is described by a variable number of entries. Two entries must be present for each problem instance: the name of the problem instance and the number of properties that its possesses. Each remaining entry describes a slot name and its slot filler.

The file t0 (Figure A3) describes the search tree for problem instance i0 The first entry in t0 is a list which serves as a template for the property values of each node. The nodes are described by levels. Each set of entries for the set of equilevel nodes consists of a count of the number of nodes on that level and the appropriate entries describing the individual nodes. Similarly, Figure A4 shows t1, the search tree for problem instance i1.

The searcher is then invoked with arguments giving the names of the files containing the symbol descriptions and the file. The actual trace produced by the searcher is as follows:

```
->(searcher 'knap 'ides)
Initialization
reading instance i1
reading instance i0
reading search tree t0
reading search tree t1
Assayer
Derivator
derivator: reading set of initial variables
w       xType evar              xSubt ersc
        xDef (get ql 'w)                    xInit nil
        xOptFrag nil                xConFrag ((K - feas))
```



```
$
w
xType       evar
xSubt       ersc
xDef        (get ql 'w)
xDir        nil
xDefvar     (w)
xInit       nil
xConsFrag   ((K-feas))
xOptFrag    nil
xRel        (w xDefvar K xConsFrag)
r
xType       evar
xSubt       eret
xDef        (get ql 'r)
xDir        nil
xDefvar     (w)
xInit       nil
xConsFrag   nil
xOptFrag    nil
xRel        (r xDefvar)
W
xType       avar
xSubt       arsc
xDef        (+ (get qp 'W) (* ql (get ql 'w)))
xDefvar     (Ww)
xDir        aft
xInit       0
xConsFrag   ((K-feas))
xOptFrag    nil
xRel        (W xDefvar w xDefvar K xConsFrag)
R
xType       avar
xSubt       aret
xDef        (+ (get qp 'R) (* ql (get ql 'r)))
xDefvar     (Rr)
xDir        aft
xInit       0
xConsFrag   0
xOptFrag    nil
xRel        (R xDefvar r xDefvar)
K
xType       pconst
xSubt       nil
xDef        (get qk 'K)
xDefvar     (K)
xDir        nil
xConsFrag   ((W-feas)(w-feas))
xOptFrag    nil
xRel        (W xConsFrag w xConsFrag)
```

Figure A1 Listing of file knap

```
2
i1
?
type        value
numitems    5
proplist    (r w)
r           (1 2 3 1 2)
w           (2 1 2 1 2)
K           5
tree        t1
i0
?
type        value
numitems    5
proplist    (r w)
r           (1 4 3 2 5)
w           (3 4 2 1 4)
K           8
tree        t0
```

Figure A2 Listing of file des.

(	L	R	W	B	Si	Pa	Ch	Op)
1								
aa0	0	0	0	10	aa4	nil	(aa1 aa2)	nil
2								
aa1	1	0	0	10	aa2	aa0	(aa3 aa4)	0
aa2	1	1	3	8	aa3	aa0	(aa5 aa6)	1
4								
aa3	2	0	0	10	aa4	aa1	(aa7 aa8)	0
aa4	2	1	1	9	aa3	aa1	(aa9 aa10)	1
aa5	2	1	3	8	aa6	aa2	(aa11 aa12)	0
aa6	2	3	7	7	aa5	aa2	(aa13)	1
7								
aa7	3	0	0	7	aa8	aa3	(aa14 aa15)	0
aa8	3	3	2	10	aa7	aa3	(aa16 aa17)	1
aa9	3	4	4	9	aa10	aa4	(aa18 aa19)	0
aa10	3	7	6	9	aa9	aa4	(aa20 aa21)	1
aa11	3	1	3	8	aa12	aa5	(aa22 aa23)	0
aa12	3	4	5	8	aa11	aa5	(aa24 aa25)	1
aa13	3	5	7	7	nil	aa6	(aa26 aa27)	0
14								
aa14	4	0	0	5	aa15	aa7	(aa28 aa29)	0
aa15	4	2	1	7	aa14	aa7	(aa30 aa31)	1
aa16	4	3	2	8	aa17	aa8	(aa32 aa33)	0
aa17	4	5	3	10	aa16	aa8	(aa34 aa35)	1
aa18	4	4	4	9	aa19	aa9	(aa36 aa37)	0
aa19	4	6	5	8	aa18	aa9	(aa38)	1
aa20	4	7	6	7	aa21	aa10	(aa39)	0
aa21	4	8	7	9	aa20	aa10	(aa40)	1
aa22	4	1	3	6	aa23	aa11	(aa41 aa42)	0
aa23	4	3	4	8	aa22	aa11	(aa43 aa44)	1
aa24	4	4	5	4	aa25	aa12	(aa45)	0
aa25	4	6	6	6	aa24	aa12	(aa46)	1
aa26	4	5	7	5	aa27	aa13	(aa47)	0
aa27	4	7	8	7	aa26	aa13	(aa48)	1
21								
aa28	5	0	0	0	aa29	aa14	nil	0
aa29	5	5	4	5	aa28	aa14	nil	1
aa30	5	2	1	2	aa31	aa15	nil	0
aa31	5	7	5	7	aa30	aa15	nil	1
aa32	5	3	2	3	aa33	aa16	nil	0
aa33	5	8	6	8	aa32	aa16	nil	1
aa34	5	5	3	5	aa35	aa17	nil	0
aa35	5	10	7	10	aa34	aa17	nil	1
aa36	5	4	4	4	aa37	aa18	nil	0
aa37	5	9	8	9	aa36	aa18	nil	1
aa38	5	6	5	6	nil	aa19	nil	0
aa39	5	7	6	7	nil	aa20	nil	0
aa40	5	8	7	8	nil	aa21	nil	0
aa41	5	1	3	1	aa42	aa22	nil	0
aa42	5	6	7	6	aa41	aa22	nil	1
aa43	5	3	4	3	aa44	aa23	nil	0
aa44	5	8	8	8	aa43	aa23	nil	1
aa45	5	4	5	4	nil	aa24	nil	0
aa46	5	6	6	6	nil	aa25	nil	0
aa47	5	5	7	5	nil	aa26	nil	0
aa48	5	7	8	7	nil	aa27	nil	0

Figure A3 Listing of file t0.

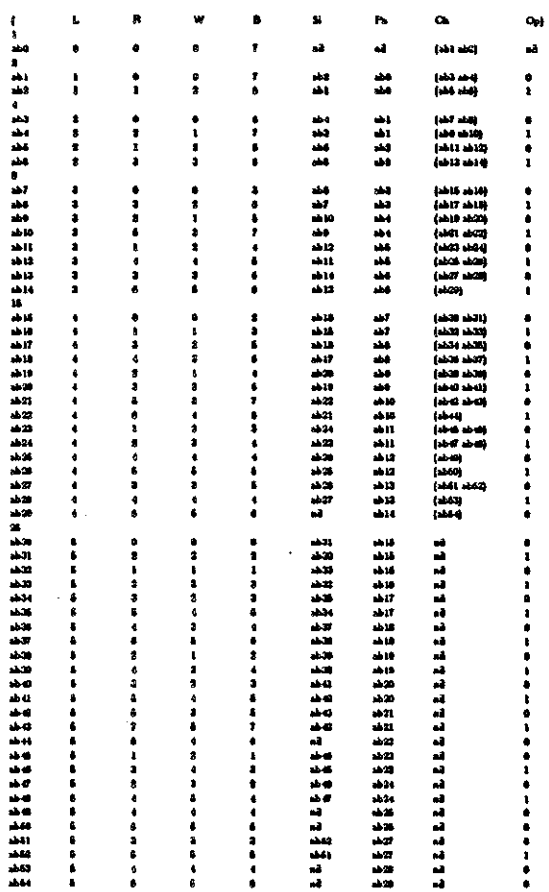


Figure A4 Listing of file t1.


```
        xDefvar (w)                    xRel (w xDefvar K xConsFrag)
r       xType evar              xSubt eret
        xDef (get ql 'r)                       xInit nil
        xOptFrag nil                   xConFrag nil
        xDefvar (w)                    xRel (r xDefvar)
W       xType svar              xSubt srec
        xDef (+ (get qp 'W) (* qd (get ql 'w)))        xInit 0
        xOptFrag nil                   xConFrag ((K - feas))
        xDefvar (W w)                  xRel (W xDefvar w xDefvar K xConsFrag)
R       xType svar              xSubt sret
        xDef (+ (get qp 'R) (* qd (get ql 'r)))        xInit 0
        xOptFrag nil                   xConFrag 0
        xDefvar (R r)                  xRel (R xDefvar r xDefvar)
K       xType pconst            xSubt nil
        xDef (get qt 'K)                       xInit nil
        xOptFrag nil                   xConFrag ((W - feas) (w - feas))
        xDefvar (K)                    xRel (W xConsFrag w xConsFrag)
assayer: (get x 'R) and (get y 'R) have identical properties
new conditions: ((> (get x 'R) (get y 'R)) (< (get x 'R) (get y 'R)) (equal (get x 'R) (get y 'R)))
assayer: (get x 'W) and (get y 'W) have identical properties
new conditions: ((> (get x 'W) (get y 'W)) (< (get x 'W) (get y 'W)) (equal (get x 'W) (get y 'W)))
Cartographer
ca1   LHS : ((Sibling x y) (> (get x 'R) (get y 'R)))
      Parallel : cb1  Anti : ca2        Parent : (ca0)    Children : (ca4 ca5 ca6)
ca2   LHS : ((Sibling x y) (< (get x 'R) (get y 'R)))
      Parallel : cb2  Anti : ca1        Parent : (ca0)    Children : (ca7 ca8 ca9)
ca3   LHS : ((Sibling x y) (equal (get x 'R) (get y 'R)))
      Parallel : cb3  Anti : ca3        Parent : (ca0)    Children : (ca10 ca11 ca12)
cb1   LHS : ((equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)))
      Parallel : ca1  Anti : cb2        Parent : (cb0)    Children : (cb4 cb5 cb6)
cb2   LHS : ((equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)))
      Parallel : ca2  Anti : cb1        Parent : (cb0)    Children : (cb7 cb8 cb9)
cb3   LHS : ((equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)))
      Parallel : ca3  Anti : cb3        Parens : (cb0)    Children : (cb10 cb11 cb12)
ca4   LHS : ((Sibling x y) (> (get x 'R) (get y 'R)) (> (get x 'W) (get y 'W)))
      Parallel : cb4  Anti : ca8        Parent : (ca1)    Children : nil
ca5   LHS : ((Sibling x y) (> (get x 'R) (get y 'R)) (< (get x 'W) (get y 'W)))
      Parallel : cb5  Anti : ca7        Parent : (ca1)    Children : nil
ca6   LHS : ((Sibling x y) (> (get x 'R) (get y 'R)) (equal (get x 'W) (get y 'W)))
      Parallel : cb6  Anti : ca9        Parent : (ca1)    Children : nil
ca7   LHS : ((Sibling x y) (< (get x 'R) (get y 'R)) (> (get x 'W) (get y 'W)))
      Parallel : cb7  Anti : ca5        Parent : (ca2)    Children : nil
ca8   LHS : ((Sibling x y) (< (get x 'R) (get y 'R)) (< (get x 'W) (get y 'W)))
      Parallel : cb8  Anti : ca4        Parent : (ca2)    Children : nil
ca9   LHS : ((Sibling x y) (< (get x 'R) (get y 'R)) (equal (get x 'W) (get y 'W)))
      Parallel : cb9  Anti : ca6        Parent : (ca2)    Children : nil
ca10  LHS : ((Sibling x y) (equal (get x 'R) (get y 'R)) (> (get x 'W) (get y 'W)))
      Parallel : cb10 Anti : ca11       Parent : (ca3)    Children : nil
ca11  LHS : ((Sibling x y) (equal (get x 'R) (get y 'R)) (< (get x 'W) (get y 'W)))
      Parallel : cb11 Anti : ca10       Parent : (ca3)    Children : nil
ca12  LHS : ((Sibling x y) (equal (get x 'R) (get y 'R)) (equal (get x 'W) (get y 'W)))
      Parallel : cb12 Anti : ca12       Parent : (ca3)    Children : nil
```

```
cb4   LHS : ((equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (> (get x 'W) (get y 'W)))
      Parallel : ca4  Anti : cb8        Parent : (cb1)    Children : nil
cb5   LHS : ((equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (< (get x 'W) (get y 'W)))
      Parallel : ca5  Anti : cb7        Parent : (cb1)    Children : nil
cb6   LHS : ((equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (equal (get x 'W) (get y 'W)))
      Parallel : ca6  Anti : cb9        Parent : (cb1)    Children : nil
cb7   LHS : ((equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (> (get x 'W) (get y 'W)))
      Parallel : ca7  Anti : cb5        Parent : (cb2)    Children : nil
cb8   LHS : ((equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (< (get x 'W) (get y 'W)))
      Parallel : ca8  Anti : cb4        Parent : (cb2)    Children : nil
cb9   LHS : ((equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (equal (get x 'W) (get y 'W)))
      Parallel : ca9  Anti : cb6        Parent : (cb2)    Children : nil
cb10  LHS : ((equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (> (get x 'W) (get y 'W)))
      Parallel : ca10 Anti : cb11       Parent : (cb3)    Children : nil
cb11  LHS : ((equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (< (get x 'W) (get y 'W)))
      Parallel : ca11 Anti : cb10       Parent : (cb3)    Children : nil
cb12  LHS : ((equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (equal (get x 'W) (get y 'W)))
      Parallel : ca12 Anti : cb12       Parent : (cb3)    Children : nil
Generator chooses rule ca1 of rank 2
Testing rule ca1 with delta nil
tester: rule ca1 is unacceptable
tester: antirule ca2 is unacceptable
Scrounger called with rule ca1
scrounger: cb1 is parallel to unacceptable c-rule ca1 -> cb1 is unacceptable
scrounger: ca4 is a child of unacceptable c-rule ca1 -> partition shows ca4 is unacceptable
scrounger: ca5 is a child of unacceptable c-rule ca1 -> partition shows ca5 is impossible
scrounger: ca6 is a child of unacceptable c-rule ca1 -> partition shows ca6 is impossible
scrounger: cb2 is parallel to unacceptable c-rule ca2 -> cb2 is unacceptable
scrounger: ca7 is a child of unacceptable c-rule ca2 -> partition shows ca7 is unacceptable
scrounger: ca8 is a child of unacceptable c-rule ca2 -> partition shows ca8 is impossible
scrounger: ca9 is a child of unacceptable c-rule ca2 -> partition shows ca9 is impossible
scrounger: cb4 is parallel to unacceptable c-rule ca4 -> cb4 is unacceptable
scrounger: cb5 is parallel to impossible c-rule ca5 -> increase rank of cb5
scrounger: cb6 is parallel to impossible c-rule ca6 -> increase rank of cb6
scrounger: cb7 is parallel to unacceptable c-rule ca7 -> cb7 is unacceptable
scrounger: cb8 is parallel to impossible c-rule ca8 -> increase rank of cb8
scrounger: cb9 is parallel to impossible c-rule ca9 -> increase rank of cb9
Generator chooses rule cb5 of rank 4
Testing rule cb5 with delta nil
tester: rule cb5 is acceptable
      LHS : ((equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (< (get x 'W) (get y 'W)))
      #{ +ve ex) : 20     #{-ve ex} : 0     #{null ex} : 0
tester: antirule cb7 is unacceptable
Scrounger called with rule cb5
Generator chooses rule cb8 of rank 4
Testing rule cb8 with delta nil
tester: rule cb8 is acceptable
      LHS : ((equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (equal (get x 'W) (get y 'W)))
      #{ +ve ex} : 18     #{-ve ex} : 0     #{null ex) : 0
tester: antirule cb9 is unacceptable
Scrounger called with rule cb8
Generator chooses rule cb8 of rank 4
```



```
Testing rule cb8 with delta nil
tester: rule cb8 is unacceptable
tester: antirule cb4 is unacceptable
Scrounger called with rule cb8
Generator chooses rule ca3 of rank 2
Testing rule ca3 with delta nil
tester: rule ca3 is impossible
Scrounger called with rule ca3
scrounger: cb3 is parallel to impossible c-rule ca3 -> increase rank of cb3
scrounger: ca10 is a child of impossible c-rule ca3 -> ca10 is impossible
scrounger: ca11 is a child of impossible c-rule ca3 -> ca11 is impossible
scrounger: ca12 is a child of impossible c-rule ca3 -> ca12 is impossible
scrounger: cb10 is parallel to impossible c-rule ca10 -> increase rank of cb10
scrounger: cb11 is parallel to impossible c-rule ca11 -> increase rank of cb11
scrounger: cb12 is parallel to impossible c-rule ca12 -> increase rank of cb12
Generator chooses rule cb3 of rank 4
Testing rule cb3 with delta nil
tester: rule cb3 is unacceptable
Scrounger called with rule cb3
scrounger: cb10 is a child of unacceptable c-rule cb3 -> partition shows cb10 is unacceptable
scrounger: cb11 is a child of unacceptable c-rule cb3 -> partition shows cb11 is acceptable
      LHS : ((equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (< (get x 'W) (get y 'W)))
      #{ +ve ex) : 7     #{-ve ex} : 0     #{null ex} : 8
scrounger: cb12 is a child of unacceptable c-rule cb3 -> partition shows cb12 is acceptable
      LHS : ((equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (equal (get x 'W) (get y 'W)))
      #{ +ve ex} : 0     #{-ve ex} : 0     #{null ex) : 4
Assayer
Derivator
K-W     xType svar              xSubt shyb
      xDef (- (get qt 'K) (get qx 'W))              xInit nil
      xOptFrag nil                    xConFrag nil
      xDefvar (w W K)                 xRel (W xDefvar K xDefvar)
derivator: new summary variable from r
Rr      xType svar              xSubt ssum
      xDef (- (get qp 'Rr) (get ql 'r))             xInit (nsum (get qt 'r))
      xOptFrag nil                    xConFrag nil
      xDefvar (Rr r)                  xRel (r xDefvar)
r/w     xType svar              xSubt chyb
      xDef (quotient (get ql 'r) (get ql 'w))       xInit nil
      xOptFrag nil                    xConFrag nil
      xDefvar (r w)                   xRel (r xDefvar w xDefvar)
derivator: new summary variable from w
Rw      xType svar              xSubt ssum
      xDef (- (get qp 'Rw) (get ql 'w))             xInit (nsum (get qt 'w))
      xOptFrag nil                    xConFrag nil
      xDefvar (Rw w)                  xRel (w xDefvar)
Computing values of new symbols for problem instances
assayer: both summary variable Rw and state variable K-W involve w
new conditions: ((> (get x 'K-W) (get ql 'Rw)) (< (get x 'K-W) (get ql 'Rw)) (equal (get x 'K-W) (get ql 'Rw)))
assayer: both summary variable Rw and state variable K-W involve w
new conditions: ((> (get y 'K-W) (get ql 'Rw)) (< (get y 'K-W) (get ql 'Rw)) (equal (get y 'K-W) (get ql 'Rw)))
Cartographer
```

ca13 LHS : ((Sibling x y) (> (get x 'R) (get y 'R)) (> (get x 'K-W) (get ql 'Rw)))
Parallel : cb13 Anti : ca22 Parent : (ca1) Children : nil
ca14 LHS : ((Sibling x y) (> (get x 'R) (get y 'R)) (< (get x 'K-W) (get ql 'Rw)))
Parallel : cb14 Anti : ca23 Parent : (ca1) Children : nil
ca15 LHS : ((Sibling x y) (> (get x 'R) (get y 'R)) (equal (get x 'K-W) (get ql 'Rw)))
Parallel : cb15 Anti : ca24 Parent : (ca1) Children : nil
ca16 LHS : ((Sibling x y) (> (get x 'R) (get y 'R)) (> (get y 'K-W) (get ql 'Rw)))
Parallel : cb16 Anti : ca19 Parent : (ca1) Children : nil
ca17 LHS : ((Sibling x y) (> (get x 'R) (get y 'R)) (< (get y 'K-W) (get ql 'Rw)))
Parallel : cb17 Anti : ca20 Parent : (ca1) Children : nil
ca18 LHS : ((Sibling x y) (> (get x 'R) (get y 'R)) (equal (get y 'K-W) (get ql 'Rw)))
Parallel : cb18 Anti : ca21 Parent : (ca1) Children : nil
ca19 LHS : ((Sibling x y) (< (get x 'R) (get y 'R)) (> (get x 'K-W) (get ql 'Rw)))
Parallel : cb19 Anti : ca16 Parent : (ca2) Children : nil
ca20 LHS : ((Sibling x y) (< (get x 'R) (get y 'R)) (< (get x 'K-W) (get ql 'Rw)))
Parallel : cb20 Anti : ca17 Parent : (ca2) Children : nil
ca21 LHS : ((Sibling x y) (< (get x 'R) (get y 'R)) (equal (get x 'K-W) (get ql 'Rw)))
Parallel : cb21 Anti : ca18 Parent : (ca2) Children : nil
ca22 LHS : ((Sibling x y) (< (get x 'R) (get y 'R)) (> (get y 'K-W) (get ql 'Rw)))
Parallel : cb22 Anti : ca13 Parent : (ca2) Children : nil
ca23 LHS : ((Sibling x y) (< (get x 'R) (get y 'R)) (< (get y 'K-W) (get ql 'Rw)))
Parallel : cb23 Anti : ca14 Parent : (ca2) Children : nil
ca24 LHS : ((Sibling x y) (< (get x 'R) (get y 'R)) (equal (get y 'K-W) (get ql 'Rw)))
Parallel : cb24 Anti : ca15 Parent : (ca2) Children : nil
cb13 LHS : ((equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (> (get x 'K-W) (get ql 'Rw)))
Parallel : ca13 Anti : cb22 Parent : (cb1) Children : nil
cb14 LHS : ((equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (< (get x 'K-W) (get ql 'Rw)))
Parallel : ca14 Anti : cb23 Parent : (cb1) Children : nil
cb15 LHS : ((equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (equal (get x 'K-W) (get ql 'Rw)))
Parallel : ca15 Anti : cb24 Parent : (cb1) Children : nil
cb16 LHS : ((equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (> (get y 'K-W) (get ql 'Rw)))
Parallel : ca16 Anti : cb19 Parent : (cb1) Children : nil
cb17 LHS : ((equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (< (get y 'K-W) (get ql 'Rw)))
Parallel : ca17 Anti : cb20 Parent : (cb1) Children : nil
cb18 LHS : ((equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (equal (get y 'K-W) (get ql 'Rw)))
Parallel : ca18 Anti : cb21 Parent : (cb1) Children : nil
cb19 LHS : ((equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (> (get x 'K-W) (get ql 'Rw)))
Parallel : ca19 Anti : cb16 Parent : (cb2) Children : nil
cb20 LHS : ((equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (< (get x 'K-W) (get ql 'Rw)))
Parallel : ca20 Anti : cb17 Parent : (cb2) Children : nil
cb21 LHS : ((equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (equal (get x 'K-W) (get ql 'Rw)))
Parallel : ca21 Anti : cb18 Parent : (cb2) Children : nil
cb22 LHS : ((equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (> (get y 'K-W) (get ql 'Rw)))
Parallel : ca22 Anti : cb13 Parent : (cb2) Children : nil
cb23 LHS : ((equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (< (get y 'K-W) (get ql 'Rw)))
Parallel : ca23 Anti : cb14 Parent : (cb2) Children : nil
cb24 LHS : ((equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (equal (get y 'K-W) (get ql 'Rw)))
Parallel : ca24 Anti : cb15 Parent : (cb2) Children : nil
cb25 LHS : ((equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (> (get x 'K-W) (get ql 'Rw)))
Parallel : nil Anti : cb28 Parent : (cb3) Children : nil
cb26 LHS : ((equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (< (get x 'K-W) (get ql 'Rw)))
Parallel : nil Anti : cb29 Parent : (cb3) Children : nil

cb27 LHS : ((equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (equal (get x 'K-W) (get ql 'Rw)))
Parallel : nil Anti : cb30 Parent : (cb3) Children : nil
cb28 LHS : ((equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (> (get y 'K-W) (get ql 'Rw)))
Parallel : nil Anti : cb25 Parent : (cb3) Children : nil
cb29 LHS : ((equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (< (get y 'K-W) (get ql 'Rw)))
Parallel : nil Anti : cb26 Parent : (cb3) Children : nil
cb30 LHS : ((equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (equal (get y 'K-W) (get ql 'Rw)))
Parallel : nil Anti : cb27 Parent : (cb3) Children : nil
Generator chooses rule ca13 of rank 2
Testing rule ca13 with delta nil
tester: rule ca13 is acceptable
LHS : ((Sibling x y) (> (get x 'R) (get y 'R)) (> (get x 'K-W) (get ql 'Rw)))
#{+ve ex} : 5 #{-ve ex} : 0 #{null ex} : 0
tester: antirule ca22 is unacceptable
Scrounger called with rule ca13
scrounger: cb13 is parallel to acceptable c-rule ca13 -> increase rank of cb13
scrounger: cb22 is parallel to unacceptable c-rule ca22 -> cb22 is unacceptable
Generator chooses rule cb13 of rank 4
Testing rule cb13 with delta nil
tester: rule cb13 is acceptable
LHS : ((equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (> (get x 'K-W) (get ql 'Rw)))
#{+ve ex} : 32 #{-ve ex} : 0 #{null ex} : 0
tester: antirule cb22 is unacceptable
Scrounger called with rule cb13
Generator chooses rule ca14 of rank 2
Testing rule ca14 with delta nil
tester: rule ca14 is unacceptable
tester: antirule ca23 is unacceptable
Scrounger called with rule ca14
scrounger: cb14 is parallel to unacceptable c-rule ca14 -> cb14 is unacceptable
scrounger: cb23 is parallel to unacceptable c-rule ca23 -> cb23 is unacceptable
Generator chooses rule ca15 of rank 2
Testing rule ca15 with delta nil
tester: rule ca15 is acceptable
LHS : ((Sibling x y) (> (get x 'R) (get y 'R)) (equal (get x 'K-W) (get ql 'Rw)))
#{+ve ex} : 4 #{-ve ex} : 0 #{null ex} : 0
tester: antirule ca24 is unacceptable
Scrounger called with rule ca15
scrounger: cb15 is parallel to acceptable c-rule ca15 -> increase rank of cb15
scrounger: cb24 is parallel to unacceptable c-rule ca24 -> cb24 is unacceptable
Generator chooses rule cb15 of rank 4
Testing rule cb15 with delta nil
tester: rule cb15 is acceptable
LHS : ((equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (equal (get x 'K-W) (get ql 'Rw)))
#{+ve ex} : 40 #{-ve ex} : 0 #{null ex} : 0
tester: antirule cb24 is unacceptable
Scrounger called with rule cb15
Generator chooses rule ca16 of rank 2
Testing rule ca16 with delta nil
tester: rule ca16 is unacceptable
tester: antirule ca19 is unacceptable
Scrounger called with rule ca16

```
scrounger: cb16 is parallel to unacceptable c-rule ca16 -> cb16 is unacceptable
scrounger: cb19 is parallel to unacceptable c-rule ca19 -> cb19 is unacceptable
Generator chooses rule ca17 of rank 2
Testing rule ca17 with delta nil
tester: rule ca17 is unacceptable
tester: antirule ca20 is unacceptable
Scrounger called with rule ca17
scrounger: cb17 is parallel to unacceptable c-rule ca17 -> cb17 is unacceptable
scrounger: cb20 is parallel to unacceptable c-rule ca20 -> cb20 is unacceptable
Generator chooses rule ca18 of rank 2
Testing rule ca18 with delta nil
tester: rule ca18 is unacceptable
tester: antirule ca21 is unacceptable
Scrounger called with rule ca18
scrounger: cb18 is parallel to unacceptable c-rule ca18 -> cb18 is unacceptable
scrounger: cb21 is parallel to unacceptable c-rule ca21 -> cb21 is unacceptable
Generator chooses rule cb25 of rank 1
Testing rule cb25 with delta nil
tester: rule cb25 is acceptable
      LHS : {(equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (> (get x 'K-W) (get ql 'R w))}
      #{+ve ex} : 1      #{-ve ex} : 0      #{null ex} : 6
tester: antirule cb28 is unacceptable
Scrounger called with rule cb25
Generator chooses rule cb28 of rank 1
Testing rule cb28 with delta nil
tester: rule cb28 is unacceptable
tester: antirule cb28 is unacceptable
Scrounger called with rule cb28
Generator chooses rule cb27 of rank 1
Testing rule cb27 with delta nil
tester: rule cb27 is acceptable
      LHS : {(equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (equal (get x 'K-W) (get ql 'R w))}
      #{+ve ex} : 5      #{-ve ex} : 0      #{null ex} : 4
tester: antirule cb30 is unacceptable
Scrounger called with rule cb27
Assayer
Derivator
W + R w     xType svar                xSubt shyb
      xDef [+ (get qx 'W) (get ql 'R w)]                xInit nil
      xOptFrag nil                      xConFrag nil
      xDefvar {w W R w}                 xRel {R w xDefvar W xDefvar w xDefvar}
R + Rr     xType svar                xSubt shyb
      xDef {+ (get qx 'R) (get ql 'Rr)}                 xInit nil
      xOptFrag nil                      xConFrag nil
      xDefvar {r R Rr}                  xRel {Rr xDefvar R xDefvar r xDefvar}
Computing values of new symbols for problem instances
assayer: R + Rr is a hybrid of R
new conditions: {{> (get y 'R) (get x 'R + Rr)} {< (get y 'R) (get x 'R + Rr)} (equal (get y 'R) (get x 'R + Rr))}
assayer: R + Rr is a hybrid of R
new conditions: {{> (get x 'R) (get y 'R + Rr)} {< (get x 'R) (get y 'R + Rr)} (equal (get x 'R) (get y 'R + Rr))}
assayer: W + R w is a hybrid of W
new conditions: {{> (get y 'W) (get x 'W + R w)} {< (get y 'W) (get x 'W + R w)} (equal (get y 'W) (get x 'W + R w))}
```



```
assayer: W + R w is a hybrid of W
new conditions: {{> (get x 'W) (get y 'W + R w)} {< (get x 'W) (get y 'W + R w)} (equal (get x 'W) (get y 'W + R w))}
Cartographer
ca25  LHS : {(Sibling x y) (> (get x 'R) (get y 'R)) (> (get y 'R) (get x 'R + Rr))}
      Parallel : cb31 Anti : nil        Parent : {ca1}    Children : nil
ca26  LHS : {(Sibling x y) (> (get x 'R) (get y 'R)) (< (get y 'R) (get x 'R + Rr))}
      Parallel : cb32 Anti : nil        Parent : {ca1}    Children : nil
ca27  LHS : {(Sibling x y) (> (get x 'R) (get y 'R)) (equal (get y 'R) (get x 'R + Rr))}
      Parallel : cb33 Anti : nil        Parent : {ca1}    Children : nil
ca28  LHS : {(Sibling x y) (> (get x 'R) (get y 'R)) (> (get x 'R) (get y 'R + Rr))}
      Parallel : cb34 Anti : nil        Parent : {ca1}    Children : nil
ca29  LHS : {(Sibling x y) (> (get x 'R) (get y 'R)) (< (get x 'R) (get y 'R + Rr))}
      Parallel : cb35 Anti : nil        Parent : {ca1}    Children : nil
ca30  LHS : {(Sibling x y) (> (get x 'R) (get y 'R)) (equal (get x 'R) (get y 'R + Rr))}
      Parallel : cb36 Anti : nil        Parent : {ca1}    Children : nil
ca31  LHS : {(Sibling x y) (> (get x 'R) (get y 'R)) (> (get y 'W) (get x 'W + R w))}
      Parallel : cb37 Anti : nil        Parent : {ca1}    Children : nil
ca32  LHS : {(Sibling x y) (> (get x 'R) (get y 'R)) (< (get y 'W) (get x 'W + R w))}
      Parallel : cb38 Anti : nil        Parent : {ca1}    Children : nil
ca33  LHS : {(Sibling x y) (> (get x 'R) (get y 'R)) (equal (get y 'W) (get x 'W + R w))}
      Parallel : cb39 Anti : nil        Parent : {ca1}    Children : nil
ca34  LHS : {(Sibling x y) (> (get x 'R) (get y 'R)) (> (get x 'W) (get y 'W + R w))}
      Parallel : cb40 Anti : nil        Parent : {ca1}    Children : nil
ca35  LHS : {(Sibling x y) (> (get x 'R) (get y 'R)) (< (get x 'W) (get y 'W + R w))}
      Parallel : cb41 Anti : nil        Parent : {ca1}    Children : nil
ca36  LHS : {(Sibling x y) (> (get x 'R) (get y 'R)) (equal (get x 'W) (get y 'W + R w))}
      Parallel : cb42 Anti : nil        Parent : {ca1}    Children : nil
ca37  LHS : {(Sibling x y) (< (get x 'R) (get y 'R)) (> (get y 'R) (get x 'R + Rr))}
      Parallel : cb43 Anti : nil        Parent : {ca2}    Children : nil
ca38  LHS : {(Sibling x y) (< (get x 'R) (get y 'R)) (< (get y 'R) (get x 'R + Rr))}
      Parallel : cb44 Anti : nil        Parent : {ca2}    Children : nil
ca39  LHS : {(Sibling x y) (< (get x 'R) (get y 'R)) (equal (get y 'R) (get x 'R + Rr))}
      Parallel : cb45 Anti : nil        Parent : {ca2}    Children : nil
ca40  LHS : {(Sibling x y) (< (get x 'R) (get y 'R)) (> (get x 'R) (get y 'R + Rr))}
      Parallel : cb46 Anti : nil        Parent : {ca2}    Children : nil
ca41  LHS : {(Sibling x y) (< (get x 'R) (get y 'R)) (< (get x 'R) (get y 'R + Rr))}
      Parallel : cb47 Anti : nil        Parent : {ca2}    Children : nil
ca42  LHS : {(Sibling x y) (< (get x 'R) (get y 'R)) (equal (get x 'R) (get y 'R + Rr))}
      Parallel : cb48 Anti : nil        Parent : {ca2}    Children : nil
ca43  LHS : {(Sibling x y) (< (get x 'R) (get y 'R)) (> (get y 'W) (get x 'W + R w))}
      Parallel : cb49 Anti : nil        Parent : {ca2}    Children : nil
ca44  LHS : {(Sibling x y) (< (get x 'R) (get y 'R)) (< (get y 'W) (get x 'W + R w))}
      Parallel : cb50 Anti : nil        Parent : {ca2}    Children : nil
ca45  LHS : {(Sibling x y) (< (get x 'R) (get y 'R)) (equal (get y 'W) (get x 'W + R w))}
      Parallel : cb51 Anti : nil        Parent : {ca2}    Children : nil
ca46  LHS : {(Sibling x y) (< (get x 'R) (get y 'R)) (> (get x 'W) (get y 'W + R w))}
      Parallel : cb52 Anti : nil        Parent : {ca2}    Children : nil
ca47  LHS : {(Sibling x y) (< (get x 'R) (get y 'R)) (< (get x 'W) (get y 'W + R w))}
      Parallel : cb53 Anti : nil        Parent : {ca2}    Children : nil
ca48  LHS : {(Sibling x y) (< (get x 'R) (get y 'R)) (equal (get x 'W) (get y 'W + R w))}
      Parallel : cb54 Anti : nil        Parent : {ca2}    Children : nil
cb31  LHS : {(equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (> (get y 'R) (get x 'R + Rr))}
```

Parallel : ca25 Anti : nil Parent : {cb1} Children : nil
cb32 LHS : {(equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (< (get y 'R) (get x 'R +Rr))}
Parallel : ca26 Anti : nil Parent : {cb1} Children : nil
cb33 LHS : {(equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (equal (get y 'R) (get x 'R +Rr))}
Parallel : ca27 Anti : nil Parent : {cb1} Children : nil
cb34 LHS : {(equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (> (get x 'R) (get y 'R +Rr))}
Parallel : ca28 Anti : nil Parent : {cb1} Children : nil
cb35 LHS : {(equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (< (get x 'R) (get y 'R +Rr))}
Parallel : ca29 Anti : nil Parent : {cb1} Children : nil
cb36 LHS : {(equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (equal (get x 'R) (get y 'R +Rr))}
Parallel : ca30 Anti : nil Parent : {cb1} Children : nil
cb37 LHS : {(equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (> (get y 'W) (get x 'W +Rw))}
Parallel : ca31 Anti : nil Parent : {cb1} Children : nil
cb38 LHS : {(equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (< (get y 'W) (get x 'W +Rw))}
Parallel : ca32 Anti : nil Parent : {cb1} Children : nil
cb39 LHS : {(equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (equal (get y 'W) (get x 'W +Rw))}
Parallel : ca33 Anti : nil Parent : {cb1} Children : nil
cb40 LHS : {(equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (> (get x 'W) (get y 'W +Rw))}
Parallel : ca34 Anti : nil Parent : {cb1} Children : nil
cb41 LHS : {(equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (< (get x 'W) (get y 'W +Rw))}
Parallel : ca35 Anti : nil Parent : {cb1} Children : nil
cb42 LHS : {(equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (equal (get x 'W) (get y 'W +Rw))}
Parallel : ca36 Anti : nil Parent : {cb1} Children : nil
cb43 LHS : {(equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (> (get y 'R) (get x 'R +Rr))}
Parallel : ca37 Anti : nil Parent : {cb2} Children : nil
cb44 LHS : {(equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (< (get y 'R) (get x 'R +Rr))}
Parallel : ca38 Anti : nil Parent : {cb2} Children : nil
cb45 LHS : {(equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (equal (get y 'R) (get x 'R +Rr))}
Parallel : ca39 Anti : nil Parent : {cb2} Children : nil
cb46 LHS : {(equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (> (get x 'R) (get y 'R +Rr))}
Parallel : ca40 Anti : nil Parent : {cb2} Children : nil
cb47 LHS : {(equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (< (get x 'R) (get y 'R +Rr))}
Parallel : ca41 Anti : nil Parent : {cb2} Children : nil
cb48 LHS : {(equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (equal (get x 'R) (get y 'R +Rr))}
Parallel : ca42 Anti : nil Parent : {cb2} Children : nil
cb49 LHS : {(equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (> (get y 'W) (get x 'W +Rw))}
Parallel : ca43 Anti : nil Parent : {cb2} Children : nil
cb50 LHS : {(equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (< (get y 'W) (get x 'W +Rw))}
Parallel : ca44 Anti : nil Parent : {cb2} Children : nil
cb51 LHS : {(equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (equal (get y 'W) (get x 'W +Rw))}
Parallel : ca45 Anti : nil Parent : {cb2} Children : nil
cb52 LHS : {(equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (> (get x 'W) (get y 'W +Rw))}
Parallel : ca46 Anti : nil Parent : {cb2} Children : nil
cb53 LHS : {(equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (< (get x 'W) (get y 'W +Rw))}
Parallel : ca47 Anti : nil Parent : {cb2} Children : nil
cb54 LHS : {(equal (get x 'L) (get y 'L)) (< (get x 'R) (get y 'R)) (equal (get x 'W) (get y 'W +Rw))}
Parallel : ca48 Anti : nil Parent : {cb2} Children : nil
cb55 LHS : {(equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (> (get y 'R) (get x 'R +Rr))}
Parallel : nil Anti : nil Parent : {cb3} Children : nil
cb56 LHS : {(equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (< (get y 'R) (get x 'R +Rr))}
Parallel : nil Anti : nil Parent : {cb3} Children : nil
cb57 LHS : {(equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (equal (get y 'R) (get x 'R +Rr))}

Parallel : nil Anti : nil Parent : {cb3} Children : nil
cb58 LHS : {(equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (> (get x 'R) (get y 'R +Rr))}
Parallel : nil Anti : nil Parent : {cb3} Children : nil
cb59 LHS : {(equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (< (get x 'R) (get y 'R +Rr))}
Parallel : nil Anti : nil Parent : {cb3} Children : nil
cb60 LHS : {(equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (equal (get x 'R) (get y 'R +Rr))}
Parallel : nil Anti : nil Parent : {cb3} Children : nil
cb61 LHS : {(equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (> (get y 'W) (get x 'W +Rw))}
Parallel : nil Anti : nil Parent : {cb3} Children : nil
cb62 LHS : {(equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (< (get y 'W) (get x 'W +Rw))}
Parallel : nil Anti : nil Parent : {cb3} Children : nil
cb63 LHS : {(equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (equal (get y 'W) (get x 'W +Rw))}
Parallel : nil Anti : nil Parent : {cb3} Children : nil
cb64 LHS : {(equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (> (get x 'W) (get y 'W +Rw))}
Parallel : nil Anti : nil Parent : {cb3} Children : nil
cb65 LHS : {(equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (< (get x 'W) (get y 'W +Rw))}
Parallel : nil Anti : nil Parent : {cb3} Children : nil
cb66 LHS : {(equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (equal (get x 'W) (get y 'W +Rw))}
Parallel : nil Anti : nil Parent : {cb3} Children : nil
Generator chooses rule ca25 of rank 2
Testing rule ca25 with delta nil
tester: rule ca25 is impossible
Scrounger called with rule ca25
scrounger: cb31 is parallel to impossible c-rule ca25 -> increase rank of cb31
Generator chooses rule cb31 of rank 4
Testing rule cb31 with delta nil
tester: rule cb31 is impossible
Scrounger called with rule cb31
Generator chooses rule ca26 of rank 2
Testing rule ca26 with delta nil
tester: rule ca26 is unacceptable
Scrounger called with rule ca26
scrounger: cb32 is parallel to unacceptable c-rule ca26 -> cb32 is unacceptable
Generator chooses rule ca27 of rank 2
Testing rule ca27 with delta nil
tester: rule ca27 is impossible
Scrounger called with rule ca27
scrounger: cb33 is parallel to impossible c-rule ca27 -> increase rank of cb33
Generator chooses rule cb33 of rank 4
Testing rule cb33 with delta nil
tester: rule cb33 is impossible
Scrounger called with rule cb33
Generator chooses rule ca28 of rank 2
Testing rule ca28 with delta nil
tester: rule ca28 is impossible
Scrounger called with rule ca28
scrounger: cb34 is parallel to impossible c-rule ca28 -> increase rank of cb34
Generator chooses rule cb34 of rank 4
Testing rule cb34 with delta nil
tester: rule cb34 is acceptable
LHS : {(equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (> (get x 'R) (get y 'R +Rr))}
#(+ve ex) : 55 #(-ve ex) : 0 #(null ex) : 0

Scrounger called with rule cb34
Generator chooses rule ca29 of rank 2
Testing rule ca29 with delta nil
tester: rule ca29 is unacceptable
Scrounger called with rule ca29
scrounger: cb35 is parallel to unacceptable c-rule ca29 -> cb35 is unacceptable
Generator chooses rule ca30 of rank 2
Testing rule ca30 with delta nil
tester: rule ca30 is acceptable
LHS : ((Sibling x y) (> (get x 'R) (get y 'R)) (equal (get x 'R) (get y 'R+Rr)))
#(+ve ex) : 4 #(-ve ex) : 0 #(null ex) : 0
Scrounger called with rule ca30
scrounger: cb38 is parallel to acceptable c-rule ca30 -> increase rank of cb36
Generator chooses rule cb36 of rank 4
Testing rule cb36 with delta nil
tester: rule cb36 is acceptable
LHS : ((equal (get x 'L) (get y 'L)) (> (get x 'R) (get y 'R)) (equal (get x 'R) (get y 'R+Rr)))
#(+ve ex) : 25 #(-ve ex) : 0 #(null ex) : 16
Scrounger called with rule cb36
Generator chooses rule ca31 of rank 2
Testing rule ca31 with delta nil
tester: rule ca31 is impossible
Scrounger called with rule ca31
scrounger: cb37 is parallel to impossible c-rule ca31 -> increase rank of cb37
Generator chooses rule cb37 of rank 4
Testing rule cb37 with delta nil
tester: rule cb37 is impossible
Scrounger called with rule cb37
Generator chooses rule ca32 of rank 2
Testing rule ca32 with delta nil
tester: rule ca32 is unacceptable
Scrounger called with rule ca32
scrounger: cb38 is parallel to unacceptable c-rule ca32 -> cb38 is unacceptable
Generator chooses rule ca33 of rank 2
Testing rule ca33 with delta nil
tester: rule ca33 is impossible
Scrounger called with rule ca33
scrounger: cb39 is parallel to impossible c-rule ca33 -> increase rank of cb39
Generator chooses rule cb39 of rank 4
Testing rule cb39 with delta nil
tester: rule cb39 is impossible
Scrounger called with rule cb39
Generator chooses rule ca34 of rank 2
Testing rule ca34 with delta nil
tester: rule ca34 is impossible
Scrounger called with rule ca34
scrounger: cb40 is parallel to impossible c-rule ca34 -> increase rank of cb40
Generator chooses rule cb40 of rank 4
Testing rule cb40 with delta nil
tester: rule cb40 is unacceptable
Scrounger called with rule cb40
Generator chooses rule ca35 of rank 2

Testing rule ca35 with delta nil
tester: rule ca35 is unacceptable
Scrounger called with rule ca35
scrounger: cb41 is parallel to unacceptable c-rule ca35 -> cb41 is unacceptable
Generator chooses rule ca36 of rank 2
Testing rule ca36 with delta nil
tester: rule ca36 is impossible
Scrounger called with rule ca36
scrounger: cb42 is parallel to impossible c-rule ca36 -> increase rank of cb42
Generator chooses rule cb42 of rank 4
Testing rule cb42 with delta nil
tester: rule cb42 is unacceptable
Scrounger called with rule cb42
Generator chooses rule ca37 of rank 2
Testing rule ca37 with delta nil
tester: rule ca37 is impossible
Scrounger called with rule ca37
scrounger: cb43 is parallel to impossible c-rule ca37 -> increase rank of cb43
Generator chooses rule cb43 of rank 4
Testing rule cb43 with delta nil
tester: rule cb43 is unacceptable
Scrounger called with rule cb43
Generator chooses rule ca38 of rank 2
Testing rule ca38 with delta nil
tester: rule ca38 is unacceptable
Scrounger called with rule ca38
scrounger: cb44 is parallel to unacceptable c-rule ca38 -> cb44 is unacceptable
Generator chooses rule ca39 of rank 2
Testing rule ca39 with delta nil
tester: rule ca39 is unacceptable
Scrounger called with rule ca39
scrounger: cb45 is parallel to unacceptable c-rule ca39 -> cb45 is unacceptable
Generator chooses rule ca40 of rank 2
Testing rule ca40 with delta nil
tester: rule ca40 is impossible
Scrounger called with rule ca40
scrounger: cb46 is parallel to impossible c-rule ca40 -> increase rank of cb46
Generator chooses rule cb46 of rank 4
Testing rule cb46 with delta nil
tester: rule cb46 is impossible
Scrounger called with rule cb46
Generator chooses rule ca41 of rank 2
Testing rule ca41 with delta nil
tester: rule ca41 is unacceptable
Scrounger called with rule ca41
scrounger: cb47 is parallel to unacceptable c-rule ca41 -> cb47 is unacceptable
Generator chooses rule ca42 of rank 2
Testing rule ca42 with delta nil
tester: rule ca42 is impossible
Scrounger called with rule ca42
scrounger: cb48 is parallel to impossible c-rule ca42 -> increase rank of cb48
Generator chooses rule cb48 of rank 4

```
Testing rule cb48 with delta nil
tester: rule cb48 is impossible
Scrounger called with rule cb48
Generator chooses rule ca43 of rank 2
Testing rule ca43 with delta nil
tester: rule ca43 is impossible
Scrounger called with rule ca43
scrounger: cb49 is parallel to impossible c-rule ca43 -> increase rank of cb49
Generator chooses rule cb49 of rank 4
Testing rule cb49 with delta nil
tester: rule cb49 is unacceptable
Scrounger called with rule cb49
Generator chooses rule ca44 of rank 2
Testing rule ca44 with delta nil
tester: rule ca44 is unacceptable
Scrounger called with rule ca44
scrounger: cb50 is parallel to unacceptable c-rule ca44 -> cb50 is unacceptable
Generator chooses rule ca45 of rank 2
Testing rule ca45 with delta nil
tester: rule ca45 is impossible
Scrounger called with rule ca45
scrounger: cb51 is parallel to impossible c-rule ca45 -> increase rank of cb51
Generator chooses rule cb51 of rank 4
Testing rule cb51 with delta nil
tester: rule cb51 is unacceptable
Scrounger called with rule cb51
Generator chooses rule ca46 of rank 2
Testing rule ca46 with delta nil
tester: rule ca46 is impossible
Scrounger called with rule ca46
scrounger: cb52 is parallel to impossible c-rule ca46 -> increase rank of cb52
Generator chooses rule cb52 of rank 4
Testing rule cb52 with delta nil
tester: rule cb52 is impossible
Scrounger called with rule cb52
Generator chooses rule ca47 of rank 2
Testing rule ca47 with delta nil
tester: rule ca47 is unacceptable
Scrounger called with rule ca47
scrounger: cb53 is parallel to unacceptable c-rule ca47 -> cb53 is unacceptable
Generator chooses rule ca48 of rank 2
Testing rule ca48 with delta nil
tester: rule ca48 is impossible
Scrounger called with rule ca48
scrounger: cb54 is parallel to impossible c-rule ca48 -> increase rank of cb54
Generator chooses rule cb54 of rank 4
Testing rule cb54 with delta nil
tester: rule cb54 is impossible
Scrounger called with rule cb54
Generator chooses rule cb55 of rank 1
Testing rule cb55 with delta nil
tester: rule cb55 is impossible
```



```
Scrounger called with rule cb55
Generator chooses rule cb56 of rank 1
Testing rule cb56 with delta nil
tester: rule cb56 is unacceptable
Scrounger called with rule cb56
Generator chooses rule cb57 of rank 1
Testing rule cb57 with delta nil
tester: rule cb57 is impossible
Scrounger called with rule cb57
Generator chooses rule cb58 of rank 1
Testing rule cb58 with delta nil
tester: rule cb58 is impossible
Scrounger called with rule cb58
Generator chooses rule cb59 of rank 1
Testing rule cb59 with delta nil
tester: rule cb59 is unacceptable
Scrounger called with rule cb59
Generator chooses rule cb60 of rank 1
Testing rule cb60 with delta nil
tester: rule cb60 is impossible
Scrounger called with rule cb60
Generator chooses rule cb61 of rank 1
Testing rule cb61 with delta nil
tester: rule cb61 is impossible
Scrounger called with rule cb61
Generator chooses rule cb62 of rank 1
Testing rule cb62 with delta nil
tester: rule cb62 is unacceptable
Scrounger called with rule cb62
Generator chooses rule cb63 of rank 1
Testing rule cb63 with delta nil
tester: rule cb63 is acceptable
      LHS : ((equal (get x 'L) (get y 'L)) (equal (get x 'R) (get y 'R)) (equal (get y 'W) (get x 'W +Rw)))
      #( +ve ex) : 2      #(-ve ex) : 0      #(null ex) : 1
Scrounger called with rule cb63
Generator chooses rule cb64 of rank 1
Testing rule cb64 with delta nil
tester: rule cb64 is impossible
Scrounger called with rule cb64
Generator chooses rule cb65 of rank 1
Testing rule cb65 with delta nil
tester: rule cb65 is unacceptable
Scrounger called with rule cb65
Generator chooses rule cb66 of rank 1
Testing rule cb66 with delta nil
tester: rule cb66 is unacceptable
Scrounger called with rule cb66
Assayer
Derivator
Trace ends
```

VITA