Learning Dominance Relations in Combinatorial Search Problems

CHEE-FEN YU, MEMBER, IEEE, AND BENJAMIN W. WAH, SENIOR MEMBER, IEEE

Abstract—**Dominance relations are used to prune unnecessary nodes in search graphs. However, they are problem dependent and cannot be derived by a general procedure. In this paper, machine learning of dominance heuristics is presented. A classification of dominance relations and the applicable learning mechanisms are identified. A study of learning dominance heuristics using learning by experimentation is described. The design of TEACHER-1 (Techniques using Experimentation for Acquiring and Creating HEuRistics) is shown. This system has been able to learn dominance heuristics for the 0/1-knapsack problem, an inventory problem, the reliability-by-replication problem, the two-machine flow shop problem, a number of single-machine scheduling problems, and a two-machine scheduling problem. The proof that the dominance heuristics learned are true dominance relations is not part of TEACHER-1 and is not automated. The same methodology can be extended to learn dominance heuristics in general.**

Index Terms—**Combinatorial search, dominance relations, heuristics learning, learning by experimentation.**

I. INTRODUCTION

COMBINATORIAL search problems require the search of one or more solutions in a defined problem space. They are divided into decision problems and optimization problems. A *decision problem* seeks any solution that satisfies the given set of constraints, while an *optimization problem* seeks the solution that optimizes a given objective function while satisfying the set of constraints. A *feasible solution* is any solution that satisfies the given set of constraints. 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, and 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. Links in a search tree are called *branches*. It is common to discuss trees 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 with children, then that number is called the *branching factor*. Each *branch* in a search tree represents the assignment of a value to an unassigned parameter. Generating 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 called the *active list*.

Each nonterminal node in a search tree is either an AND node or an OR node. An AND node implies a problem reduction from the problem represented by the node to multiple smaller subproblems represented by its children. The problem represented by an AND node is solved when all the subproblems represented by its children are solved. An example of an AND node is one that adds the solutions from all the subtrees expanded from it. In contrast, an OR node implies a state-space approach that transforms the state represented by the node, by operators represented by the arcs, to states represented by its children. An OR node is solved if any of its children are solved.

A pruning scheme identifies and eliminates *ineffective nodes* known not to lead to a solution better than an existing node. One pruning scheme eliminates one of the two nodes when both have identical states. Another pruning scheme involves dominance relations [2], [8], [20], [21]. 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 because a solution in T_i no worse than the best solution in T_j does not always 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 cannot be solved in polynomial time.

The lower-bound test used in branch-and-bound algorithms is a dominance relation. The pruning power of the lower-bound tests depends on the quality of the lower-

Manuscript received February 15, 1986; revised June 30, 1987. This research was supported in part by the National Science Foundation under Grant MIPS 85-19649 and in part by the National Aeronautics and Space Administration under Grant NCC2-481.

C.-F. Yu is with the Artificial Intelligence Laboratory, Intel Corporation, Santa Clara, CA 95052.

B. W. Wah is with the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana-Champagin, Urbana, IL 61801.

IEEE Log Number 8822013.

bound function [45]. In the lower-bound test, the best solution found so far dominates all nodes with lower bounds not greater than its solution value. Although more general dominance relations have been used [20], there is no systematic procedure to derive dominance relations because they are problem dependent. Identifying and encoding problem-specific dominance relations in 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.

In this paper, we present results on machine learning of dominance heuristics. Note that a machine learning system can hypothesize dominance heuristics but does not prove that the heuristics found are true dominance relations. The proof is to be carried out by an expert or a theorem prover. The methodology used in our work follows the outline suggested by Winston [49].

1) First, identify the problem to be solved.

2) Then, devise appropriate representations for dominance relations and search problems.

3) Next, expose constraints on the permissible forms of the dominance relations.

4) Only then devise the learning mechanisms. This is done in two phases: an *initial phase* and a *refinement phase*. In the initial phase, we seek an overall description of the applicable learning mechanisms. These include the aspects of the problem to which each learning mechanism may be applied, the important issues in the mechanisms, and the interaction among the different mechanisms. In the refinement phase, the knowledge needed to drive the individual learning mechanisms is derived. Based upon the results obtained, it may be necessary to refine the problem domain classification and the overall description.

5) Finally, verify via experiments.

Sections II–VI describe the results of applying the steps of this methodology. Section VII describes a prototype program that learns dominance heuristics by experimentation.

II. Problem Definition

As indicated in Section I, combinatorial search problems may be classified into decision problems and optimization problems. Two functions may be associated with an optimization problem I [19]: 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 $\text{Max}[I] \leq p(\text{Length}[I])$. A similar definition holds for the *number decision problem*. The solution of a combinatorial search problem requires *pseudopolynomial time* if its time complexity function is bounded from above by a polynomial function of Length[I] and Max[I]. Most problems with known dominance relations can be solved by dynamic programming methods and can be evaluated in pseudopolynomial time.

Learning something new usually requires knowing much about the problem already, and it is prudent to start with a problem domain known to be rich in dominance relations. Many problems that are known to have dominance relations are number optimization problems. Since dominance relations do not change the worst case time complexity of the problem, they are generally more effective for problems with pseudopolynomial-time solutions than for problems with exponential-time solutions.

III. Representation

A dominance relation is a statement that if two nodes satisfy the conditions described in the dominance relation, then one node contains a solution no worse than that in the other node. This statement is easily expressed by an IF–THEN rule (production rule). The left-hand side establishes the conditions that the two nodes must satisfy, and the right-hand side establishes the relationship between the best solutions contained in these two nodes. This production rule is easily transformed into an implication in first-order predicate logic, and vice versa. This transformation permits switching between production rules and predicate logic as the situation dictates.

A good representation for number optimization problems exposes aspects of the problem relevant to learning dominance heuristics by incorporating relevant problem domain knowledge. A good representation also permits techniques developed for one problem to be reused for related problems.

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

$$\text{minimize } C_0(x) \qquad x \in X$$

$$\text{subject to } g_i(x) \leq 0 \qquad i = 1, 2, \cdots, m.$$

This description is inadequate for learning dominance relations as it lacks the necessary domain information. The needed domain information is often related to the subject matter of the problem. One classification of combinatorial search problems [19] lists 12 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 can further be divided into several minor subject areas. The number optimization problems can come from any of these subject areas except logic.

A comprehensive study of all these subject areas is probably too massive in scope for an initial study of learning dominance relations. We have chosen to investigate some of these subject areas in detail instead of making a superficial study of all the subject areas. The research in this paper uses problems in mathematical programming and in sequencing and scheduling. Problems in these two areas are somewhat related, and we have substantial experience with problems in these areas. The mathematical programming problems studied in this paper are the 0/1-

knapsack problem, the inventory problem, and the machine-reliability-by-duplication problem. The problems in sequencing and scheduling studied in this paper are sequencing to minimize weighted completion time, sequencing to minimize the latest completion time, sequencing to minimize weighted lateness, sequencing to minimize task weights, sequencing to minimize tardy task weights, and the two-machine flow shop 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 capacity K. Each item $u \in U$ has a weight $w(u)$ and a value $r(u)$. The desired solution is the subset $U' \subseteq U$ which maximizes $\Sigma_{u \in U'} r(u)$ and satisfies $\Sigma_{u \in U'} w(u) \leq K$.

In sequencing to minimize weighted completion time, there is a set T of n tasks. Each task $t \in T$ has a length $l(t)$, a weight $w(t)$, and a deadline $d(t)$. The desired solution is the one-processor schedule σ for T which minimizes $\Sigma_{t \in T}(\sigma(t) + l(t))\, w(t)$ and satisfies $(\sigma(t) + l(t)) \leq d(t)$ 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 $\Sigma_{i=1}^{n} [x(m_i) \times s(m_i) - y(m_i) \times b(m_i)]$. There can at most be B units of the item at any time, so $v + \Sigma_{i=1}^{j} [x(m_i) - y(m_i)] \leq B$ for all $j \in \{1, \cdots, n\}$. Amounts purchased are delivered at the end of each period, so the amount sold in each period must not exceed the stock at the end of the previous period, and $y(m_i) \leq v + \Sigma_{j=1}^{i-1} [x(m_j) - y(m_j)]$ for all $i \in \{1, \cdots, n\}$.

Although these three problems involve quite different domains, they are rather similar in form (see Table I). Each problem description contains a set of entities. A set of decisions must be made on this set of entities. A solution is found when decisions have been made for all of the entities. Each description also contains constraints that define the allowable set of decisions and an optimization criterion that defines the desired optimal solution. The optimization criterion is either the maximization or the minimization of an objective expression. Both 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, and limit properties establish the limits. An entity variable may 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.

TABLE I
SIMILARITIES AMONG THE PROBLEM 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), for all m∈ **M**
Constraints	$\sum_{u \in U'} w(u) \leq K$	$[\sigma(t)+\ell(t)] \leq d(t)$, for all t∈ T	$v+\sum_{i=1}^{j}[x(m_i)-y(m_i)] \leq B$, $y(m_i) \leq v+\sum_{j=1}^{i-1}[x(m_j)-y(m_j)]$, for all i,j∈ {1,...,n}
Objective Criteria	maximize $\sum_{u \in U'} r(u)$	minimize $\sum_{t \in T}[\sigma(t)+\ell(t)]w(t)$	maximize $\sum_{i=1}^{n}[x(m_i)s(m_i) - y(m_i)b(m_i)]$
Symbols			
Decision Symbols	**U'**	σ(t)	x(m), y(m)
Entity Properties	r(u), w(u)	ℓ(t), w(t), d(t)	b(m), s(m)
System Constants	K, n	n	v, n
Working Decision	x(u)	ρ(t)	x(m), y(m)

An efficient search method requires a bookkeeping scheme that allows efficient orderly exploration of the possible solutions. Serial methods often require the transformation of the set of decisions into a form suitable for efficient sequential decision making. Let this new set of decisions 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 are no waits between consecutive tasks in the optimal schedule.

Problem solvers make working decisions by assigning values to *decision variables* using a *working description* of the problem. State variables record the effects of previous decisions in working descriptions and assist in the assignment of values to the decision variables.

A working decision can be described by 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. The decision variable v_j in a *value description* specifies the *v*-tuple. Value descriptions usually assume some fixed order of the entities $\{E_1, E_2, \cdots, E_{nE}\}$, and the decision corresponding to variable v_j is (E_j, v_j). This results in a fixed-height search tree where all solution nodes are at a fixed distance from the root. In contrast, the de-

ENTITY	$u_1, ..., u_n$
DECISION VARIABLE	$x(u_i) = \{0,1\}$, $i \in \{1, ..., n\}$
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) ≤ K

(a)

ENTITY	$u_1, ..., u_n$
DECISION VARIABLE	$x(u_i) = \{u_1, ..., u_n\}$ $i \in \{1,...,n\}$
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) ≤ K

(b)

Fig. 1. Description frames for the 0/1-knapsack problem. (a) Value description. (b) Identity description.

cision variable i_j in an *identity description* specifies only the value of the e-tuple. One common type of identity description assumes a constant value for the v-tuple of the named decision and another 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 search tree in an identity description may be either 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 has only 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 period $j/2$. When j is odd, v_j is the amount sold in period $\lfloor j/2 \rfloor$.

The similarities between the different problem descriptions suggest a domain-independent structure of the problem descriptions. The problem descriptions have the same set of domain-independent elements: the entities, decision variables, state variables, entity variables, problem constants, optimization function, and constraints. Furthermore, this set of elements is complete.

This structure can be exposed by a frame representation [34]. The slots in the frame represent the different elements of the problem description. The slot fillers describe the actual values of these elements for the particular problem described. Fig. 1 shows the description frames for the 0/1-knapsack problem. The problem description frames for the other problems studied in this paper are similar.

IV. Constraints in the Problem Domain

Dominance relations may be classified by the possible positions of the dominating and dominated nodes. Each type of dominance relation 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 a terminal or nonterminal node when T_1 is *all*.

The value of T_2 can be *same*, *specific*, or *all*. If T_2 has the value *same*, then 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 assume a value of *sib* only when T_2 has a value of *same*; otherwise, the resulting combination does not make sense.

Table II lists examples of the different types of dominance relations. A relation of type {*term*, *same*, *sib/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 is known, so it is trivial to determine which is the better solution.

The lower-bound test, used in branch-and-bound 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 can dominate any node which has a worse lower bound.

A greedy algorithm always chooses the locally 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

TABLE II
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	trivial relations
	all	all	lower-bound test
all	same	sib	greedy algorithm
		all	dominance relations in dynamic programming
	specific	all	trivial relations
	all	all	upper-bound test

TABLE III
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	$O(\lceil$ active terminal nodes$\rceil)$	$O(\lceil$ nodes on level specific$\rceil)$
	all	all	$O(\lceil$ active terminal nodes$\rceil)$	O(active nodes)
all	same	sib	O(siblings)	O(siblings)
		all	$O(\lceil$ nodes on the same level $\rceil)$	$O(\lceil$ nodes on the same level $\rceil)$
	spec.	all	O(nodes on level {LEV(A) − specific})	O(nodes on level {LEV(A) + specific})
	all	all	O(active nodes)	O(active nodes)

well-behaved relations in other descriptions. For example, each search node 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. Dominance relations of this form may be ignored.

Another constraint may be derived from a study of the implementation costs of dominance relations. The implementation cost can be measured by the number of nodes to which each newly generated node must be compared to determine if it either dominates another active node or is dominated by an active node (see Table III). 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 proceeds. The implementation cost of such dominance relations is likely to exceed the savings resulting from the nodes they eliminate. Dominance relations of types $\{all, all, all\}$ may, therefore, be ignored.

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

V. APPLICABLE LEARNING MECHANISMS

Learning is a very general term denoting the way people (and computers) increase their knowledge and improve their skills. Fig. 2 shows a simplified model of a machine-learning system [7]. 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 provides feedback to the learning element.

The learning element bridges the gap between the level of information provided by the environment and the level required by the performance element. The learning element does not know in advance exactly how to fill this gap. 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.

The level of information provided by the environment determines the kinds of hypotheses that the system must generate and determines the type of learning strategy used [7], [11], [33]. Several basic strategies have been distinguished: *rote learning*, *learning by instruction*, *learning by deduction*, *learning by analogy*, *learning by induction*, and explanation-based learning. The latter is subdivided into *learning from examples* and *learning by observation and discovery*. These strategies are ordered by increasing complexity of the hypothesis (inference) and reflect increasing effort on the part of the learning element and correspondingly decreasing effort on the part of the environment (teacher).

In rote learning [25], [42], [43], the environment supplies knowledge in a form that can be used directly. The learning system need not understand or interpret the environmental information, but memorize the information for later use. One major concern here is the indexing of stored knowledge for future retrieval. Rote learning can be applied to learning known dominance relations and cannot learn new dominance heuristics or 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. The system must understand and interpret the high-level knowledge and relate it to what it already knows. Work in this area includes McCarthy's proposal for an "advice taker," the TEIRESIAS program [13], and the program FOO [38].

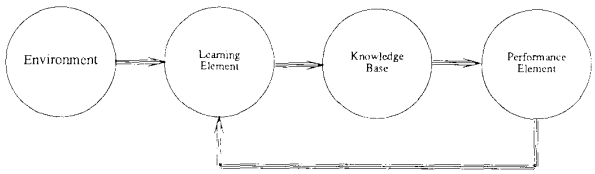


Fig. 2. A simplified model of a machine-learning system.

At present, there is neither an organized source of knowledge about dominance relations nor a 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 macrooperators, caching, chunking, equivalence-preserving operationalization, and other truth-preserving transformations.

Learning by deduction includes aspects of automatic deduction or mechanical theorem proving [39], [51]. An important early advance in automatic deduction was the development of *resolution procedures* based on the *resolution principle* [41]. The search space generated by resolution grows exponentially with the number of formulas used to describe a problem. Domain-independent heuristics, such as *set of support* [50], have been shown to be too weak to produce satisfactory results. In this case, domain-specific guidance is needed.

Nonetheless, there is an important class of problems that resist solution by methods other than deduction. Deduction permits logically complex queries of a knowledge base containing generalizations. Recent work in deduction includes work on commonsense reasoning [16], [30] and *nonmonotonic logic* [1]. Nonmonotonic logic can retract a conclusion in the face of further evidence.

Dominance relations can be 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; hence, if a candidate relation with the prescribed form can be proved to be logically true, then it must be a dominance relation for the problem under study. Auotmated deduction is very good at finding proofs. Section V-A 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 transforms relevant knowledge in the knowledge base of the related task into a form suitable for use by its performance element. Learning by analogy has been mainly used in automated problem solving [9], [10], [12], [17], [22], [31], [37], [48] and to guide the modification of programs [3], [14], [29].

Analogy can be used to learn dominance heuristics. Given a set of problems and their dominance relations, the new problem domain and its set of known problem domains. The common substructure between the two descriptions can then guide the transformation of 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 relations are likely to be dominance relations in the new domain. Section V-B describes learning dominance relations by analogy in greater detail.

Learning by induction involves the generalization of input information and the 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 [4], [15], [18], [32], [35], [36], [44], [47] or the environment on which the student performes experiments and from which it receives feedback [5], [6], [40], [46]. 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 the help of a teacher for regularities and general rules explaining all or at least most of the observations. This form of learning includes *conceptual clustering* (object classes describable by simple concepts), constructing classifications, fitting equations to data, discovering laws to explain a set of observations [23], [24], and formulating theories to account for the behavior of a system [13], [26]–[28].

Learning by examples can be used to learn dominance heuristics. A weak problem solver can generate the complete search tree of any problem instance. The best solution that can be derived from each search node in a complete search tree is always known. These search nodes 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 search 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 V-C considers learning dominance heuristics by experimentation in greater detail.

Explanation-based learning [54], [55] is a method of generalization that uses domain knowledge and knowledge of the concept being learned to produce a valid generalization from a single example. It uses an explanation of how the example is a member of the concept being learned to extract the features relevant to the target con-

cept. The inability to generate examples of a dominance heuristic without an explicit statement of the heuristic prevents this powerful method from being used to learn dominance heuristics.

A. Learning Dominance Relations by Deduction

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

As an example, 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 decisions that 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

$$C_1\colon =\big(L(x_1), L(y_1)\big) \quad C_3\colon <\big(W(x_3), W(y_3)\big)$$

$$C_2\colon >\big(R(x_2), R(y_2)\big) \quad C_4\colon \neg >\big(B(x_4), B(y_4)\big).$$

The set of axioms consists of clauses that describe pertinent aspects of problem solving and the problem descriptions. 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, via clause $C_5\colon =(B(x_5), \mathrm{SUM}(R(x_5), H(x_5)))$. Equilevel nodes in a value description 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 that includes the subset $D' \subseteq D$ in the knapsack there is a solution derived from u that includes the subset $D'' \subseteq D$ in the knapsack such that $D' \subseteq D''$. This implies that $H(u) > H(v)$ when $W(u) < W(v)$ and $L(u) = L(v)$, which is described by the following clause:

$$C_6\colon \neg =\big(L(x_6), L(y_6)\big) \,\big|\, \neg <\big(W(x_6), W(y_6)\big) \,\big|\, > \big(H(x_6), H(y_6)\big).$$

Furthermore, clause $C_7\colon \neg >(t_1, t_2) \,|\, \neg >(t_3, t_4) \,|\, > (\mathrm{SUM}(t_1, t_3), \mathrm{SUM}(t_2, t_4))$ describes a pertinent property of the sum of two quantities, and clause $C_8\colon = (>(u, v), \neg <(u, v) \,|\, \neg =(u, v))$ describes a relationship between the functions $>(u, v)$, $=(u, v)$, and $<(u, v)$. Fig. 3 shows the resulting resolution tree for the goal wff's. The contradiction of the negated goal wff proves the goal wff, so the candidate relation is a dominance relation.

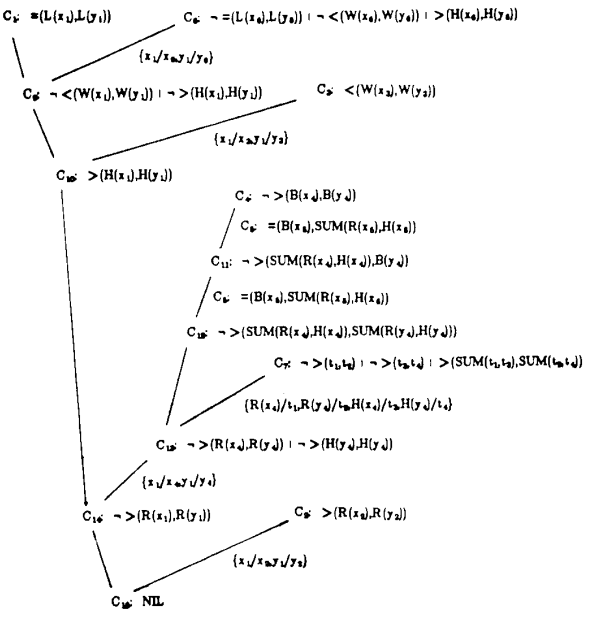


Fig. 3. Resolution tree for $=(L(x), L(y)) \;\&\; >(R(x), R(y)) \;\&\; <(W(x), W(y)) \rightarrow >(B(x), B(y))$.

In contrast, automated deduction is unable to derive the empty clause for the slightly different candidate relation $=(L(x), L(y)) \;\&\; >(R(x), R(y)) \;\&\; >(W(x), W(y)) \rightarrow \geq(B(x), B(y))$. The automated deduction system concludes that either this candidate relation is not a dominance relation or we are ignorant of some step in its proof.

An automated deduction program with sound inference rules will derive only relations that are always true when the premises are true. If the set of premises contains 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 the required deduction steps and all necessary information about the problem. This can expose questionable assumptions. More importantly, it makes explicit the reasons why a particular relation is a dominance relation and the processes by which the conclusion is reached. Such knowledge 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 detailed information can often be a handicap. The absence of a single necessary axiom will prevent the theorem prover from finding a proof of a **true** goal wff. An automated deduction system cannot modify the set of axioms supplied to it. Further, it is restricted to whatever knowledge the program designer has about dominance relations 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 insufficient by

itself to solve the problem of finding dominance relations. Automated deduction requires external mechanisms to supply it with candidate relations and initial axioms. These mechanisms should be able to innovate and derive new symbols not present in the initial problem description. The existing problem descriptions are often optimized towards state-space search methods 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 rules is well defined and well known. The axioms must include the explicit problem description as well as implicit knowledge about the problem domain and the search algorithm. The strategy is determined by the preferences assigned to the different axioms and inference rules. These preferences are problem dependent.

B. Learning Dominance Heuristics 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 cumulative lateness involves a set of n tasks: $t_1, \cdots, t_n$. 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 $\Sigma_{i=1}^{n} [\sigma(t_i) + l(t_i) - d(t_i)]$.

Each task t_i': $i \in \{1, \cdots, n\}$ 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 $\Sigma_{i=1}^{n} [\sigma'(t_i') + l'(t_i')] \times w'(t_i')$.

Table IV shows three identity descriptions for these two problems. The decision variable ρ_i is the identity of the ith 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 the completion time of the most recently scheduled task in the partial schedule represented by node N, and $L(N)$ 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 times 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 IV leads to the following analogical mapping:

$$t_i \rightarrow t_i'$$

$$\rho_i \rightarrow \rho_i'$$

$$T(N) \rightarrow T'(N)$$

$$L(N) \rightarrow C'(N)$$

TABLE IV
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	$t_1, \ldots, t_n$	$t_1', \ldots, t_n'$
DECISION VARIABLE	$\rho_i = \{t_1, \ldots, t_n\}$ i∈ {1,...,n}	$\rho_i' = \{t_1', \ldots, t_n'\}$ i∈ {1,...,n}
STATE VARIABLES	$T(P) = T(N) + \emptyset(\rho_i)$ where $P = \text{apply}(\rho_i, N)$ $L(P) = L(N) - [T(P) + d(\rho_i)]$ where $P = \text{apply}(\rho_i, N)$	$T'(P) = T'(N) + \emptyset'(\rho_i')$ where $P = \text{apply}(\rho_i', N)$ $C'(P) = C'(N) + [T'(P) * w'(\rho_i')]$ where $P = \text{apply}(\rho_i', N)$
ENTITY VARIABLES	$\emptyset(t_i), d(t_i)$	$\emptyset'(t_i'), w'(t_i')$
PROBLEM CONSTANTS	n	n
OPTIMIZATION	minimize L(S)	minimize C'(S)
CONSTRAINT	none	none

$$l(t_i) \rightarrow l'(t_i')$$

$$d(t_i) \rightarrow w'(t_i').$$

A greedy solution for scheduling to minimize cumulative lateness always schedules the unscheduled task with the least lateness when there is only one task with this value of lateness. The corresponding dominance relation is "IF [SIB(x, y)] [$L(x) < L(y)$] THEN [$B(x) \leq B(y)$]" 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. Application of this analogical mapping leads to the replacement of $L(x)$ by $C'(x)$ and the rule "IF [SIB(x, y)] [$C'(x) < C'(y)$] THEN [$B(x) \leq B(y)$]." This rule is another way of stating the known greedy scheduling algorithm that minimizes weighted completion times when only one unscheduled task has the smallest weighted completion time.

Analogy can be a very powerful and efficient method when there is a good match between the target domain and the base domain. The known dominance relation in the base domain and the analogical mapping provide powerful heuristic knowledge that drastically reduces the amount of search required.

One important factor in 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 will be 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. Heuristic knowledge is used to find the "best" base domain and transform the relations in the base domain. Testing the derived rules with sample problem instances can increase our confidence in the derived rules; however, it is impossible to test all possible problem instances.

Analogy cannot derive rules that are unrelated to known dominance relations, so it cannot derive dominance heuristics dissimilar to known dominance relations. More-

over, analogy does not work when there is no suitable base domain. Analogy is not useful in domains that are different from the domains with known dominance relations.

A system that learns dominance heuristics 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. The issues of both similarities between domains and transformations between domains will have to be addressed before analogy can be used to derive dominance relations.

C. Learning Dominance Heuristics 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 heuristics to the domain under study. Thus, learning by experimentation is not constrained to domains and heuristics that are similar to known domains and known dominance relations, although it is likely to be more efficient in familiar domains.

To learn dominance heuristics by experimentation, the system must be able to generate sample problem instances, solve these sample problem instances, and induce possible dominance heuristics from instances in the search space. A weak problem solver can solve the problem instances. The other two topics are open questions and must be solved before experimentation can learn dominance heuristics.

As indicated in Section V, in the absence of an oracle, it is impossible to identify examples of dominance heuristics among the nodes in the search tree without explicit knowledge of the proposed relation. This restriction rules out the use of data-driven methods. A suitable method is model-driven generate-and-test.

Clearly, heuristic methods have to be used to generate sample problem instances as well as to find dominance heuristics in the sample search trees. Any heuristics derived by experimentation is true only for the set of sample problem instances used in its derivation. There will always be some doubts about whether or not these heuristics are true dominance relations. Other method of proving the correctness of the dominance heuristics will be necessary.

D. 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 useful only in problem domains similar to domains with known dominance relations. Deduction is useful only in well-defined problems. A learning system may have to work in an unfamiliar environment. Only experimentation works well in these situations; hence, it must be the primary mechanism that drives a system for learning and hypothesizing new dominance heuristics. Analogy and deductions are secondary mechanisms that can be very effective in the proper situations.

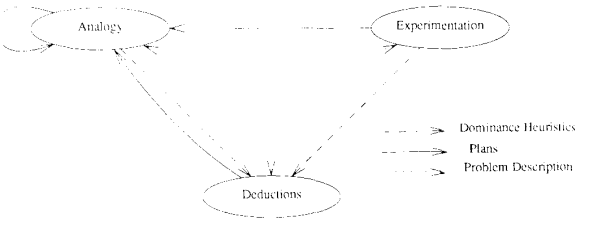


Fig. 4. Interactions between the learning mechanisms.

Fig. 4 shows the different learning mechanisms. Both analogy and experimentation derive dominance heuristics. Deduction derives dominance relations. Any relation proved by deduction to follow logically 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 it. 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 mechanism may be viewed as the execution of a plan by the learning mechanism. The efficiency of this process can be enhanced by augmenting its general plan with domain-specific knowledge. One way is to use analogy to 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 that there is an adequate pool of plans.

VI. TEACHER-1: A System that Learns Dominance Relations with Techniques using Experimentation for Acquiring and Creating Heuristics

Fig. 5 shows the organization of TEACHER-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 the relation behaves like a dominance relation during tests with sample search trees. The forester supplies the appropriate sample search trees.

The searcher tests 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 either 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 switch a description before restarting the searcher.

Details of the individual components are found in the following sections. Section VI-A describes the scheduler, Section VI-B describes the description manager, and Section VI-C describes the forester. The design of the searcher depends on the representation of the candidate

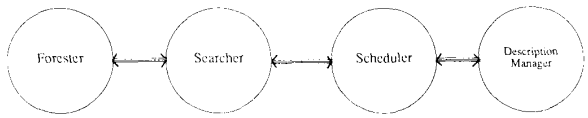


Fig. 5. Organization of TEACHER-1.

relations, so we discuss the representation of candidate relations in Section VI-D before describing the searcher in Section VI-E. These descriptions are illustrated with examples drawn mainly from the application of TEACHER-1 to the 0/1-knapsack problem. Examples from other problems are used only when there are no appropriate examples for the 0/1 knapsack problem, and these cases will be explicitly noted in the descriptions.

A. Scheduler

The scheduler coordinates the activities of the system components and incorporates the top-level control structures of the system. It decides when to stop the search of dominance heuristics. When TEACHER-1 discovers a dominance heuristic, the scheduler becomes active and tests whether or not the dominance heuristic is a complete solution; that is, it corresponds to a greedy solution of the problem. An affirmative answer causes TEACHER-1 to stop; otherwise, the search for dominance heuristics continues.

When the scheduler is activated by the exhaustion of the searcher's resource allocation, it makes a decision based 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. A preset threshold τ_0 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 decides to continue searching for dominance heuristics. If τ is greater than or equal to τ_0, the scheduler queries the user for guidance on whether to continue or stop. Values between 10 and 100 s are suitable.

If the searcher decides to continue searching, it uses the description manager to determine a proper description and allocate a new resource slice to the searcher. TEACHER-1 uses a fixed-size 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 $\kappa = \tau_0/(\eta \times \kappa_0)$. κ_0 represents the average number of evaluations for each working description during a time period of duration τ_0. 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 TEACHER-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.

B. Description Manager

The ideal description manager derives the optimal working description from the problem statement. This capability requires deep causal knowledge about the problem descriptions and the problem domains and is a major open research topic in artifical intelligence. TEACHER-1 does not address this issue; it uses a naive description manager. The user must supply the appropriate working descriptions and ensure that these descriptions are correct and complete.

The user supplies a working description in the form of a *w*-frame. The *w*-frame contains the basic problem description as well as information that assists TEACHER-1 in manipulating the problem description. Fig. 6 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-Functions* 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 that 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 following formula:

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

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

C. Forester

The forester supplies test trees to the searcher. A test tree may be used to test several relations; so the forester retains all test trees in a tree pool. The forester always tries to satisfy a request from its tree pool and generates new trees only if either the pool has been exhausted or the trees in the pool do not meet the searcher's demands.

The first step in generating a tree is to generate a problem instance 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 the problem instance

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: $R(N_t) = R(N_p) + x_i \cdot r(u_i)$
        $N_t = \text{Apply}(x_i, u_i, N_p)$
      Code: (lispfn1)
    Initial Value: 0
    Constraints: nil
    Value:
    Code:
  State Variable 2:
    Name: W()
    Definition:
      Value: $W(N_t) = W(N_p) + x_i \cdot w(u_i)$
        $N_t = \text{Apply}(x_i, u_i, N_p)$
      Code: (lispfn2)
    Initial Value: 0
    Constraints:
    Value: $W(N) \le K$
    Code: (lispfn3)
Optimization:
  Expression: $R(N_{soln})$
  Extreme Value: maximum
  Code: (lispfn4)
Decision-Variable:
  Name: x_i
  Value: [0,1]
LISP Functions: lispfn0, lispfn1, lispfn2, lispfn3, lispfn4

Fig. 6. Frame describing the value description of the 0/1-knapsack problem.

and the method of filling in these slots. The forester derives the *p*-frame from the *w*-frame by extracting the relevant subslots. The *p*-frame specifies an assignment of values to the slots of a *p*-frame instance. Fig. 7(a) shows the *p*-frame for the 0/1-knapsack problem. Fig. 7(b) shows an instance of the *p*-frame representing the 0/1-knapsack instance Π_1.

The forester then uses a weak problem solver to solve the problem instance. Table V shows the search tree τ_1 generated by the solution of the problem instance Π_1 shown in Fig. 7(b). This search tree is stored in the Search-Tree slot of Π_1.

D. Representation of Candidate Relations

TEACHER-1 represents candidate relations by *c*-rules that are production rules of the following form:

IF ⟨*t*-condition⟩ ⟨*d*-condition$_1$⟩ $\cdots$ ⟨*d*-condition$_m$⟩
THEN *x* dominates *y*

A candidate relation is a hypothesis such that if two nodes satisfy the conditions of the relation then it is always possible to identify the node with the better solution. This hypothesis can be accepted only if all nodes that satisfy its conditions agree with its identification of the better node.

The left-hand side of the *c*-rule identifies the conditions that must be satisfied by nodes *x* and *y* in order for node

Name: p-frame	Name: Π_1
Type: concept	Type: p-frame
Problem: 0/1-knapsack	Problem: 0/1-knapsack
n: Value:	n: 5
Code: (+1 (* (ran) 10))	
r: Value:	r: {1 4 3 2 5}
Code: lispfn10	
w: Value:	w: {3 4 2 1 4}
Code: lispfn11	
K: Value:	K: 8
Code: lispfn12	
Search-Tree: nil	Search-Tree:
Lisp Functions: lispfn10, lispfn11, lispfn12	
(a)	(b)

Fig. 7. (a) The *p*-frame for the 0/1-knapsack problem. (b) The *p*-frame instance for the 0/1-knapsack instance Π_1.

TABLE V
SEARCH TREE τ_1

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

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, 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 is true if and only if x and y are siblings. The rule representing a relation of type {all, same, sib} has [SIB(x, y)] as its t-condition. The term α-rule will denote 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 that 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 specify both the state variable and the node. Hence, the two state variables $R(\ )$ and $W(\ )$ result in the symbols $R(x)$, $R(y)$, $W(x)$, and $W(y)$. An entity variable is included 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 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, i.e., $B(x)$ and $B(y)$. By definition, $B(x)$ is smaller than or equal to $B(y)$ in a minimization problem; that is, $[B(x) \leq B(y)]$.

A c-rule is an *acceptable rule* if it represents a dominance heuristic. An *unacceptable rule* is a c-rule whose left-hand side cannot consistently identify the dominating and dominated nodes. An *impossible rule* is a 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.

Each c-rule has a δ-value that reflects its suitability as a dominance relation. Initially, the δ values of the c-rules are unknown. The searcher finds the δ-values of c-rules by testing search trees. If tests find only positive instances of the c-rule, then the c-rule has a δ-value of *acceptable*. If tests find one or more negative instances of the c-rule, then the c-rule has a δ-value of *unacceptable*. If tests fail 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 Fig. 8). (N_2, N_1) in search tree τ_1 (Table V) 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 causes a δ-value of unacceptable for α_2'. The left-hand side of a c-rule with a δ-value of unacceptable cannot consistently identify the node with the better solution, so it is an unacceptable c-rule.

Tests will not find any instances of the following c-rule: IF [SIB(x, y)] $[R(x) = R(y)]$ THEN $[B(x) \geq B(y)]$ (c-rule α_4'); hence, 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 or an unacceptable rule. The ambiguity in the rules results from the use of a finite number of search trees during testing. An example of this is the acceptable c-rule IF $[L(x) = L(y)]$ $[R(x) = R(y)]$ $[W(x) = W(y)]$ THEN $[B(x) \geq B(y)]$ (rule β_{13}'). Search tree τ_1 contains no instance of this rule.

The rule IF $[L(x) = L(y)]$ $[R(x) > R(y)$ $[W(x) < W(y)]$ THEN $[B(x) \geq B(y)]$ (rule β_9') has a δ-value of acceptable. β_9' consistently identifies the node with the better solution in the test set of search trees. It is most likely to be an acceptable c-rule, but may also be an unacceptable rule because of its finite test set.

Learning by experimentation can find only the δ-values of the c-rules. We assume in the design of TEACHER-1 that an adequate sample set of test trees is a 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 (respectively, unacceptable, impossible) rule and rule with a δ-value of acceptable (respectively, unacceptable, impossible).

E. 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. Fig. 8 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 $[B(x) \geq B(y)]$ 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, so there is inheritance of δ-values by descendant and ancestor c-rules. Section VI-E1) discusses this topic in greater detail.

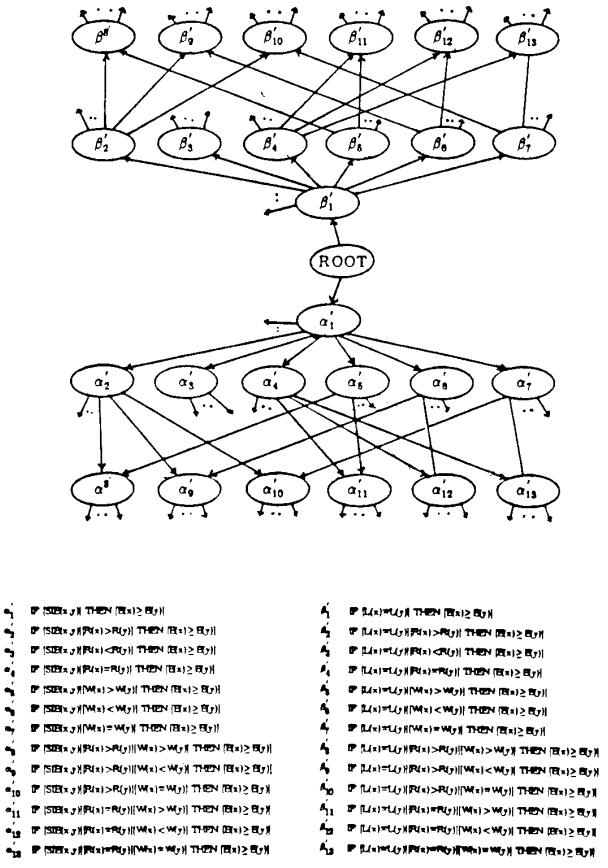


Fig. 8. A fragment of the rule graph for the 0/1-knapsack problem.

The rule graph contains both α-rules and β-rules. The α-rule is a c-rule with $[\mathrm{SIB}(x, y)]$ 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 Fig. 8 have been labeled such that rule α_i' and rule β_i' are parallel rules. Section VI-E2) explores parallelism in greater detail.

1) Value Inheritance: The parent of a c-rule is a generalization of the c-rule. Eliminating a d-condition from a c-rule yields one of its parent rules. For instance, eliminating condition $[R(x) > R(y)]$ from the c-rule α_8' produces the c-rule α_3', while eliminating 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 $\{[\mathrm{SIB}(N_i, N_j)] \& [R(N_i) > R(N_j)] \& [W(N_i) > W(N_j)] \& [B(N_i) \geq B(N_j)]\}$ 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 $\{[\mathrm{SIB}(N_i, N_j)] \& [R(N_i) > R(N_j)] \& [B(N_i) \geq B(N_j)]\}$. 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'. Every positive (respectively, negative) instance of a c-rule will be a positive (respectively, negative) instance of its parent c-rule.

By definition, there are no instances of an impossible rule. For this condition to be satisfied, none of its children can have any instances. Hence, each child of an impossible rule is an impossible rule.

An acceptable rule has positive instances, but no negative instance. 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 are negative instances of an unacceptable rule. One or more of its children must 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. This means that a child of an unacceptable rule can be another unacceptable rule, an acceptable rule, or an impossible rule.

2) Parallelism: Siblings are nodes on the same level with the same parent. $[\mathrm{SIB}(N_i, N_j)]$ implies $[L(N_i) = L(N_j)]$. Parallel rules have the same set of d-conditions and right-hand side. A positive (respectively, negative) instance of an α-rule is also a positive (respectively, 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 $\{[\mathrm{SIB}(N_2, N_1)] [R(N_2) > R(N_1)] [B(N_2) >= B(N_1)]\}$ is a true clause. Replacing $[\mathrm{SIB}(N_2), (N_1)]$ 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 *nonsibling pair* if N_i and N_j are not siblings. Siblings have a common parent and exhibit greater correlation in the values of their state variables than nonsiblings. Hence, the absence of positive (respectively, negative) instances among the sibling pairs does not imply the absence of positive (respectively, negative) instances among the nonsibling pairs. For example, there are no instances of the rule IF $[\mathrm{SIB}(x, y)] [R(x) = R(y)]$ THEN $[B(x) \geq B(y)]$ (rule α_4'), but there are positive and negative instances of the rule IF $[L(x) = L(y)] [R(x) = R(y)]$ THEN $[B(x) \geq B(y)]$ (rule β_4').

Nonetheless, it is still possible to deduce something about the possible instances of a c-rule from the instances of its parallel c-rule (see Tables VI and VII). Let α_i' and β_i' be a pair of parallel rules. 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 is an unacceptable c-rule, so β_i' is an unacceptable rule.

TABLE VI
δ VALUES OF PARALLEL β-RULE

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

TABLE VII
δ VALUES OF PARALLEL α-RULE

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

Suppose that α_i' is an acceptable c-rule. There are positive instances of α_i', but 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 be either an acceptable c-rule or an unacceptable c-rule.

Suppose that α_i is an impossible c-rule. There are neither positive nor negative instances of α_i'. These facts do not provide any useful information about 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.

F. Searcher

As indicated in Section VI, the searcher implements the generate-and-test search of 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.

It can be shown that the number of unique c-rules with m d-conditions in which the symbols in the d-conditions are chosen from a pool of n different symbols is $O(m^n)$. This implies the need for a informed generator which uses possibility-limiting information to restrict the relations it proposes.

The searcher (see Fig. 9) uses a hierarchical approach to limit the number of proposed relations. The cartographer identifies a "promising" region in rule space with the aid of the derivator and assayer and draws a map of this region. The generate–test–scrounge process explores the individual c-rules in this map.

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 that deduces the implications of the tester's decision for other

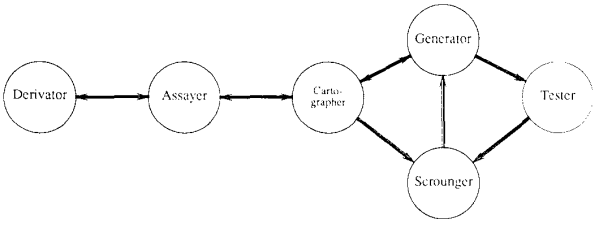


Fig. 9. Organization for the searcher.

c-rules, thereby permitting multiple decisions in a generate–test–scrounge cycle.

The generate–test–scrounge process proposes and evaluates c-rules until it has completely explored the current maps. The cartographer updates the maps by adding new c-rules, and the generate–test–scrounge process proceeds to evaluate the new c-rules. As indicated in Section VI, the searcher works under a time limit imposed by the scheduler. The searcher checks for violation of this limit only when any component of the searcher deactivates itself.

Sections VI-F1)–VI-F6) describe in detail the individual components of the searcher.

1) Cartographer: The cartographer finds regions in the α-graph (β-graph) likely to contain acceptable c-rules and produces an α-map (β-map) that describes these regions. It operates in an incremental fashion: each time the cartographer is activated, it updates the α-map (β-map) by adding new "promising" c-rules.

An acceptable c-rule must be a semantically sensible construct, and its d-conditions must be semantically sensible comparisons between semantically sensible symbols. In TEACHER-1, the cartographer uses the assayer to find new d-conditions. The assayer, in turn, uses the derivator to find new variables. Heuristic rules in the derivator discard nonsensical symbols, while heuristic rules in the assayer generate only sensible d-conditions from the symbols acceptable to the derivator.

TEACHER-1 stops when the assayer fails to find any new d-conditions; otherwise, the cartographer augments its pool of operators with the operators associated with these new d-conditions. Operators in the maps represent the addition of a d-condition to a c-rule. The cartographer then applies 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. The cartographer excludes all descendants of any c-rule with a δ-value of either impossible or acceptable. Every descendant of an impossible c-rule is an impossible c-rule. Every descendant of an acceptable c-rule has a δ-value of either acceptable or impossible. However, the descendant of a c-rule is a specialization of the c-rule and can never lead to a more powerful dominance heuristic.

A dominance relation is a relation between the best solutions in a pair of nodes. 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 $H(u)$. An acceptable rule between nodes x and y must describe the relationship

between $R(x)$ and $R(y)$ as well as the relationship between $H(x)$ and $H(y)$. β-Rules must express the relationship between $R(x)$ and $R(y)$ explicitly. α-Rules may express the relationship either explicitly or implicitly by d-conditions between other state variables. However, we can make explicit the d-condition between $R(x)$ and $R(y)$ without altering the meaning of the set of d-conditions. Hence, the maps should include only c-rules that contain a d-condition between $R(x)$ and $R(y)$. A simple way of implementing this restriction is to apply only 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 cartographer imposes a preset upper limit on the number of d-conditions in a c-rule.

The initial α-map G_α^0 in a value description of the 0/1-knapsack problem consists of the c-rule α_0: IF [SIB(x, y)] THEN [$B(x) \geq B(y)$]. The initial β-map G_β^0 consists of the c-rule β_0: IF [$L(x) = L(y)$] THEN [$B(x) \geq B(y)$]. Suppose that the initial call to the assayer returns the set of d-conditions $\{[R(x) < R(y)], [R(x) = R(y)], [R(x) > R(y)], [W(x) < W(y)], [W(x) = W(y)], [W(x) > W(y)]\}$. The resulting maps G_α^1 and G_β^1 are shown in Figs. 10 and 11.

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. Entity variables are introduced into d-conditions by associating a node with the entity involved in the decision leading to the node. Equilevel nodes may be associated with different entities in an identity description, but are always associated with the same entity in a value description. Thus, instances of entity variables are variable symbols in an identity description and 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 four types of nontrivial 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.

In the 0/1-knapsack problem, both $R(\ )$ and $W(\ )$ have semantic meaning. $R(\ )$ is the return from the items included in the knapsack, and $W(\ )$ is the weight of these items. The difference between K and $W(\ )$, $K\Delta W(\ )$, is the capacity remaining in the knapsack. In contrast, the variable $K\Sigma W(\ )$, the sum of K and $W(\ )$, has no semantic meaning.

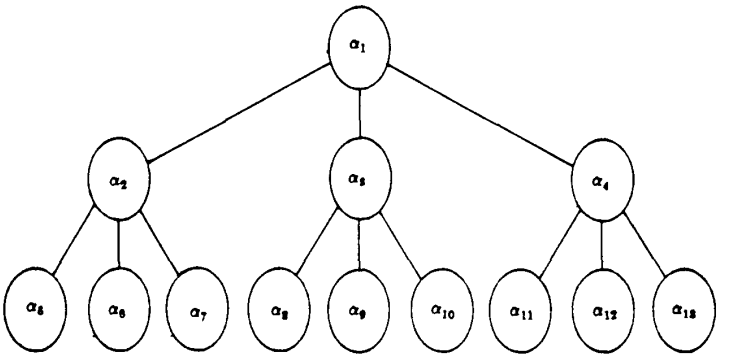
D-conditions of type T_2 and T_3 compare instances of different variables, and d-conditions of type T_4 compare an instance of a variable to a constant symbol. The assayer generates a condition from one of these three types only when it is able to deduce from the working description that the symbols in the d-conditions are semantically related by 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 problem constant K by 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 derived state variable in the 0/1-knapsack problem is $RW(\ )$, the sum of the weights of the unassigned items. Its formula is $RW(N_r) = RW(N_p) - w(\theta)$ where N_r is the search node generated by applying operator θ to node N_p. $RW(\ )$ is semantically related to $W(\ )$, as the entity variable $w(\ )$ appears in both of their definitions. Recall that the definition of $W(\ )$ is $W(N_r) = W(N_p) + w(\theta)$. Another derived state variable is $K\Delta W(\ )$, the capacity remaining in the knapsack, whose formula is $K\Delta W(N) = K - W(N)$. $RW(\ )$ is semantically related to $K\Delta W(\ )$ since both of them are related to $W(\ )$.

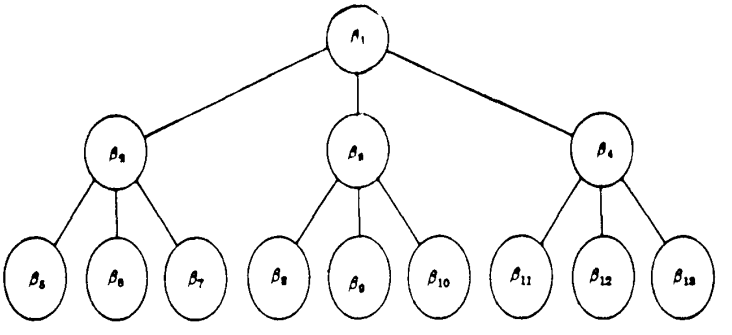
Whenever the assayer becomes active, it calls upon the derivator to find new symbols. The symbol list in the assayer consists of two sublists: the a-list and the b-list. The a-list holds the symbols generated during the most recent activation of the derivator, and the b-list holds the symbols generated earlier. D-conditions between two symbols from the b-list have already been considered by previous activations of the assayer and are not generated by the assayer.

Both the a-list and b-list are initially empty in a value description of the 0/1-knapsack problem. The derivator puts the symbols $r(\ )$, $w(\ )$, $R(\ )$, $W(\ )$, and K in the a-list. The assayer considers $r(\ )$, $w(\ )$, and K to be constant symbols and will not generate any d-conditions between them. Moreover, $r(\ )$ and $w(\ )$ do not appear in any constraint with either $R(\ )$ or $W(\ )$, and 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 generate only the d-conditions: $[R(x) < R(y)]$, $[R(x) = R(y)]$, $[R(x) > R(y)]$, $[W(x) < W(y)]$, $[W(x) = W(y)]$, and $[W(x) > W(y)]$.



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)|

Fig. 10. 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)|

Fig. 11. G_{β}^{1}.

During the second activation of the assayer, the set of symbols known to the derivator will be augmented by entity variable $r\Omega w(\ )$ and state variables $RR(\ )$, $RW(\ )$, and $K\Delta W(\ )$. $r\Omega w(\ )$ 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 $r\Omega w(\ )$, as it is not related to any variable symbol. As indicated previously, $W(\ )$, $RW(\ )$, and $K\Delta W(\ )$ are all semantically related. $RW(\ )$ and $K\Delta W(\ )$ are forward variables, and $W(\ )$ is a backward variable, so the assayer prefers $RW(\ )$ and $K\Delta W(\ )$ to $RW(\ )$ 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 and $RR(\ )$ is a forward variable. Equilevel nodes have identical values for $RR(\ )$ and $RW(\ )$, and the assayer does not generate d-conditions between $RR(\ )$ and $RW(\ )$.

Derivator: The derivator generates new variables by combining existing variables and constants. New entity variables are derived by the formula $E_0 = E_1 \# E_1$ where E_0 is the new entity variable, E_1 and E_2 are existing entity variables, and # is one of the arithmetic operators + (plus), − (minus), × (times), or ÷ (divide). Only arithmetic operators are needed, as the domain of TEACHER-1 is number optimization problems. The derivator combines two entity variables only when it heuristically finds evidence in the problem description that they are related.

In Section III, 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 commonly shared constructs are the objective function and the constraints. When two or more constraints have a common variable, the derivator also considers the composite constraints 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 return property and a limit property.

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

Another candidate for a semantically meaningful entity variable is the ratio of a return property and resource property. This new variable measures the per-unit return of this entity. In the 0/1-knapsack problem, $r(\ )$ is a return property, and $w(\ )$ is a resource property. The new entity variable, $r\Omega w(\ )$, the ratio of $r(\ )$ to $w(\ )$, is the return per-unit weight of the entity.

When the derivator combines two entity variables of the same type, the new variable has the same type as the variables used in its definition. The derivator generates a *hybrid property* when the two entity variables are of different types. Entity variables $r\Omega w(\ )$ is a hybrid property. Sensible combination of hybrid variables often requires background knowlege not present in the problem description. At present, TEACHER-1 does not combine hybrid properties with other entity variables.

A state is a data structure that captures a snapshot of the problem-solving process at some stage of the solution, and the state variables are elements of this data structure. The state variable summarizes some aspect of the decision sequence leading to the node. The derivator's heuristic rules are based on our experience in working with search algorithms and can produce some "promising" state variables.

Summary variables 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 $w(\ )$. Summary variables $RR(\ )$ and $RW(\ )$ 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.

An *alternative variable* is an alternative view of a state variable. Although the two variables are equivalent, one variable is often more suitable than the other for expressing certain facts. For 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 $W(\ ) \le K$. The new variable $K\Delta W(\ )$, where $K\Delta W(\ ) = K - W(\ )$, represents the unused capacity in the knapsack and is an alternative view of $W(\ )$. However, $K\Delta W(\ )$ is more convenient than $W(\ )$ for expressing the condition that all unassigned items can fit into the knapsack. This condition is $[K\Delta W(x) \ge RW(x)]$ when $K\Delta W(\ )$ is used and $[\mathrm{SUM}(W(x), RW(x)) \le K]$ when $W(\ )$ is used. The latter form is more awkward and is more difficult to learn.

As indicated in the previous section, the symbol list in TEACHER-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 *bb*-list are all empty. 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 drives sensible variables from the symbols in the *b*-list, while making sure not to generate variables with symbols just in the *bb*-list since they have been previously generated.

In the first activation of 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.

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 $\{R(\ ), W(\ ), r(\ ), w(\ ), K\}$, and the *bb*-list is still $\{\ \}$. The derivator will derive the entity variable $r\Omega w(\ )$, the summary variables $RW(\ )$ and $RR(\ )$, and the alternative variable $K\Delta W(\ )$.

4) Tester: The tester forms a set of test patterns from the proposed rule: the positive pattern [*t*-condition] [*d*-condition$_1$] $\cdots$ [*d*-condition$_m$] $[B(x) > B(y)]$, the null pattern [*t*-condition] [*d*-condition$_1$] $\cdots$ [*d*-condition$_m$] $[B(x) = B(y)]$, and the negative pattern [*t*-condition] [*d*-condition$_1$] $\cdots$ [*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 an *antirule* be the rule formed by interchanging the symbols x and y in the conditions of a rule. The negative (respectively, positive) pattern of the antirule is identical to the positive (respectively, 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 corresponding antirule, and vice versa.

The rule IF $[\mathrm{SIB}(x, y)]\ [R(x) > R(y)]$ THEN $[B(x) \geq B(y)]$ (rule α_2) is the first rule in map G_α^1 selected for testing by the generator. The tester will create the positive pattern $[\mathrm{SIB}(x, y)]\ [R(x) > R(y)]\ [B(x) > B(y)]$, the null pattern $[\mathrm{SIB}(x, y)]\ [R(x) > R(y)]\ [B(x) = B(y)]$, and the negative pattern $[\mathrm{SIB}(x, y)]\ [R(x) > R(y)]\ [B(x) < B(y)]$ from α_2.

Interchanging the symbols x and y on the left-hand side of α_2 produces IF $[\mathrm{SIB}(x, y)]\ [R(x) < R(y)]$ THEN $[B(x) \geq B(y)]$, which is the *c*-rule α_3. The positive pattern for α_3 is $[\mathrm{SIB}(x, y)]\ [R(x) < R(y)]\ [B(x) > B(y)]$, 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, $[\mathrm{SIB}(x, y)]\ [R(x) < R(y)]\ [B(x) < B(y)]$ and $[\mathrm{SIB}(x, y)]\ [R(x) < R(y)]\ [B(x) = B(y)]$, are, respectively, equivalent to the positive and null patterns of α_2.

TABLE VIII
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

The tester matches the set of patterns against node pairs in the test trees. Table VIII 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 identify consistently the node with the better solution; thus, the rule and its antirule are both unacceptable rules.

The presence of positive (respectively, negative) examples and the absence of negative (respectively, positive) examples suggest the possible acceptability of the rule (respectively, the antirule) and unacceptability of the antirule (respectively, the rule). However, any rule can be accepted only after adequate testing. TEACHER-1 uses a very simple criterion: the testing is adequate if the number of positive and null examples exceeds ζ_1 and the number of test trees exceeds ζ_2. Suitable values for ζ_1 and ζ_2 are 20 and 10, respectively. This simple criterion 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.

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 ζ_3, then it resumes the matching process with new test trees. A suitable value for ζ_3 is 10.

Suppose that the tester matches its set of patterns against the nodes in search tree τ_1 (see Table V). (N_8, N_7), (N_{16}, N_{14}), (N_{17}, N_{16}), (N_{21}, N_{20}), (N_{11}, N_{12}), (N_{23}, N_{22}), (N_{26}, N_{24}), and (N_{27}, N_{26}) 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.

5) Scrounger: The scrounger investigates the implications of a decision on other rules using value inheritance and parallelism. 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 or impossible rules. The scrounger attempts to identify the decision values of the child rules by matching the examples of c_i against the *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 examples of c_i which match d_j is the 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 $[\mathrm{SIB}(x, y)]$, $[R(x) > R(y)]$, 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 $[\mathrm{SIB}(x, y)]$, $[R(x) > R(y)]$, $[W(x) > W(y)]$, and $[B(x) > B(y)]$, that is, a positive example of the resulting child rule α_5. Similarly, a null (respectively, negative) example of rule α_2 that satisfies $[W(x) > W(y)]$ is also a null (respectively, negative) example of rule α_5.

An empty subset implies that there are no examples of α_j and its antirule among the test trees used to generate the example set of α_i. The scrounger does not know whether or not the set of test trees used in α_i is 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 these rules.

Similarly, if the subset does not contain negative (respectively, positive) examples of α_i, then α_j is a potentially acceptable (respectively, impossible) rule, and its antirule is a potentially impossible (respectively, acceptable) rule. The generate–test–scrounge process gives a higher priority to rules that have been identified as potentially impossible or acceptable rules.

For example, the rejection of α_2 causes the scrounger to reject the parallel rules β_2 and β_3. The scrounger then tries to identify the decision values of α_2's children. In tree τ_1 (see Table V), node pairs (N_8, N_7), (N_{16}, N_{14}), (N_{17}, N_{16}), (N_{21}, N_{20}), (N_{11}, N_{12}), (N_{23}, N_{22}), (N_{26}, N_{24}), and (N_{27}, N_{26}) 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 match the d-condition. There are positive, null, and negative examples of α_5 in tree τ_1. The scrounger rejects rule α_5, which leads to the rejection of α_9, β_6, and β_9.

Next, the scrounger considers the condition $[W(x) < W(y)]$. Only (N_{11}, N_{22}) satisfies this condition. This result implies that there are only positive examples of the resulting rule α_6 in the test tree τ_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 higher priority for testing siblings of an acceptable rule and their descendants.

6) Generator: TEACHER-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 as follow.

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 motivations behind reasons 2)–4) have been discussed elsewhere in this paper.

VII. Verification

A prototype of the searcher has been implemented on the VAX 3600 using Franz Lisp [52]. 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 because 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 TEACHER-1 was applied to the 0/1-knapsack problem, it learned the relation IF $[L(x) = L(y)]$ $[R(x) \geq R(y)]$ $[W(x) \leq W(y)]$ THEN $[B(x) \geq B(y)]$, which is the well-known dominance relation for the 0/1-knapsack problem. It is also able to learn the relation IF $[L(x) = L(y)]$ $[R(x) \geq R(y)]$ $[K \Delta W(x) \geq RW(\ )]$ THEN $[B(x) \geq B(y)]$, which leads the problem solver to include all 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 $[\mathrm{SIB}(x, y)]$ $[R(x) \geq R\Sigma RR(y)]$ THEN $[B(x) \geq B(y)]$ where $R\Sigma RR(\ )$ is defined as the sum of $R(\ )$ and $RR(\ )$. $R\Sigma RR(\ )$ is an upper bound on the return from any node.

TEACHER-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 flow shop problem, sequencing to minimize the latest completion time, and sequencing to minimize task weights.

VIII. Concluding Remarks

Even though dominance relations are problem dependent, it is possible to find them by machine-learning techniques. In this paper, we have studied machine learning of dominance relations and heuristics 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 and heuristics can be learned by experimentation, deduction, and analogy. Learning by deduction learns only relations that 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 heuristics in unfamiliar problem domains and new types of heuristics. Thus, learning by experimentation is the primary learning mechanism studied in this paper. Both analogy and deduction are also necessary, as they are very effective when used in the correct context.

We have studied learning dominance heuristics by experimentation in great detail and implemented a prototype program. This program uses model-driven generate-and-test since it is impossible to identify training instances needed to drive a data-driven paradigm based on heuristic constraints derived from manipulating the standard descriptions as well as constraints in the space of relations. In addition, the program is able to define and manipulate new variable symbols automatically.

A future paper presents the design of TEACHER-2, a system that learns by experimentation the splitting, transformation, and selection functions in combinatorial searches [53].

References

[1] Special Issue on Non-Monotonic Logic, *Artif. Intell.*, vol. 13, 1980.

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

[3] S. Amarel, "Program synthesis as a theory formation task: Problem representations and solution methods," in *Machine Learning: An Artificial Intelligence Approach, Vol. II*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Los Altos, CA: Morgan Kaufmann, 1986.

[4] P. M. Andrae, "Constraint limited generalization: Acquiring procedures from examples," in *Proc. Nat. Conf. Artif. Intell.*, AAAI, Austin, TX, 1984, pp. 6-10.

[5] Y. Anzai and H. Simon, "The theory of learning by doing," *Psychol. Rev.*, vol. 36, no. 2, pp. 124-140, 1979.

[6] A. A. Araya, "Learning problem classes by means of experimentation and generalization," in *Proc. Nat. Conf. Artif. Intell.*, AAAI, Austin, TX, 1984, pp. 11-15.

[7] A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence, Vols. 1-3*. Los Altos, CA: William Kaufmann, 1981, 1982.

[8] R. Bellman and S. Dreyfus, *Applied Dynamic Programming*. Princeton, NJ: Princeton Univ. Press, 1962.

[9] F. M. Brown and S. A. Tarnlund, "Inductive reasoning on recursive equations," *Artif. Intell.*, vol. 12, pp. 207-229, 1979.

[10] M. H. Burstein, "Concept formation by incremental analogical reasoning and debugging," in *Machine Learning: An Artificial Intelligence Approach., Vol. II*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Los Altos, CA: Morgan Kaufmann, 1986.

[11] J. G. Carbonell, R. S. Michalski, and T. M. Mitchell, "An overview of machine learning," in *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Palo Alto, CA: Tioga, 1983.

[12] J. G. Carbonell, "Learning by analogy: Formulating and generalizing plans from past experiences," in *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Palo Alto, CA: Tioga, 1983.

[13] R. Davis and R. B. Lenat, *Knowledge-Based Systems in Artificial Intelligence*. New York: McGraw-Hill, 1982.

[14] N. Dershowitz and Z. Manna, "The evolution of programs: Automatic program modification," *IEEE Trans. Software Eng.*, vol. SE-3, no. 6, pp. 377-385, 1977.

[15] T. G. Dietterich and R. S. Michalski, "A comparative review of selected methods for learning from examples," in *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Palo Alto, CA: Tioga, 1983.

[16] J. Dole, "A truth maintenance system," *Artif. Intell.*, vol. 12, no. 3, pp. 231-272, 1979.

[17] T. G. Evans, "A program for the solution of geometric analogy intelligence test questions," in *Semantic Information Processing*, M. L. Minsky, Eds. Cambridge, MA: M.I.T. Press, 1968.

[18] R. E. Fikes, P. E. Hart, and N. J. Nilsson, "Learning and executing generalized robot plans," *Arif. Intell.*, vol. 3, pp. 251-288, 1972.

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

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

[21] T. Ibaraki, "The power of dominance relations in branch-and-bound algorithms," *J. ACM*, vol. 24, no. 2, pp. 264-279, Apr. 1977.

[22] R. Kling, "A paradigm for reasoning by analogy," *Artif. Intell.*, vol. 2, pp. 147-178, 1971.

[23] P. Langley, "Data-driven discovery of physical laws," *Cognitive Sci.*, vol. 5, pp. 31-54, 1981.

[24] P. Langley, G. L. Bradshaw, and H. A. Simon, "Rediscovering chemistry with the BACON system," in *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Palo Alto, CA: Tioga, 1983.

[25] D. B. Lenat, F. Hayes-Roth, and P. Klahr, "Cognitive economy in artificial intelligence systems," in *Proc. IJCAI*, 1979, pp. 531-536.

[26] D. B. Lenat, "The nature of heuristics," *Artif. Intell.*, vol. 19, no. 2, pp. 189-249, 1982.

[27] ——, "Theory formation by heuristic search; The nature of heuristics II: Background and examples," *Artif. Intell.*, vol. 21, pp. 31-59, 1983.

[28] ——, "EURISKO: A program that learns new heuristics and domain concepts; The nature of heuristics III: Program design and results," *Artif. Intell.*, vol. 21, pp. 61-98, 1983.

[29] Z. Manna and R. J. Waldinger, "Knowledge and reasoning in program synthesis," *Artif. Intell.*, vol. 6, pp. 175-208, 1975.

[30] D. McDermott, "Planning and acting," *Cognitive Sci.*, vol. 2, pp. 71-109, 1978.

[31] J. McDermott, "Learning to use analogies," in *Proc. IJCAI*, 1979, pp. 568-576.

[32] R. S. Michalski, "A theory and methodology of inductive learning," in *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Palo Alto, CA: Tioga, 1983.

[33] ——, "Understanding the nature of learning: Issues and research directions," in *Machine Learning: An Artificial Intelligence Approach*, vol. II, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Los Altos, CA: Morgan Kaufmann, 1986.

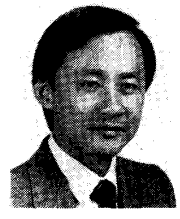
[34] M. Minsky, "A framework for representing knowledge," in *The Psychology of Computer Vision*, P. H. Winston, Ed. New York: McGraw-Hill, 1975.

[35] S. Minton, "Constraint-based generalization: Learning game-playing plans from single examples," in *Proc. Nat. Conf. Artif. Intell.*, AAAI, Austin, TX, 1984, pp. 251-254.

[36] T. M. Mitchell, "Generalization as search," *Artif. Intell.*, vol. 18, pp. 203-226, 1982.
[37] J. A. Moore and A. Newell, "How can MERLIN understand?" in *Knowledge and Cognition*, L. Gregg, Ed. Hillsdale, CA: Erlbaum, 1973.
[38] D. J. Mostow and F. Hayes-Roth, "Operationalizing heuristics: Some AI methods for assisting AI programming," in *Proc. IJCAI*, 1979, pp. 601-609.
[39] N. J. Nilsson, *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga, 1980.
[40] S. Ohlsson, "A constrained mechanism for procedural learning," in *Proc. Int. Joint Conf. Artif. Intell.* Los Altos, CA: William Kaufman, 1983, pp. 426-428.
[41] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *J. ACM*, vol. 12, no. 1, pp. 23-41, 1965.
[42] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM J. Res. Develop.*, vol. 3, pp. 210-229, 1959.
[43] ——, "Some studies in machine learning using the game of checkers II—Recent progress," *J. Res. Develop.*, vol. 11, no. 6, pp. 601-617, 1967.
[44] S. A. Vere, *Inductive Learning of Relational Productions Pattern-Directed Inference Systems*. New York: Academic, 1978.
[45] B. W. Wah and C. F. Yu, "Stochastic modeling of branch-and-bound algorithms with best-first search," *IEEE Trans. Software Eng.*, vol. SE-11, no. 9, pp. 922-934, Sept. 1985.
[46] D. A. Waterman, "Generalization learning techniques for automating the learning of heuristics," *Artif. Intell.*, vol. 1, no. 1/2, pp. 121-170, 1970.
[47] P. H. Winston, "Learning structural descriptions from examples," in *The Psychology of Comptuer Vision*, P. H. Winston, Ed. New York: McGraw-Hill, 1975.
[48] ——, "Learning and reasoning by analogy," *Commun. ACM*, vol. 23, pp. 869-703, 1979.
[49] ——, *Artificial Intelligence*, 2nd ed. Reading, MA: Addison-Wesley, 1984.
[50] 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.
[51] L. Wos, R. Overbeek, E. Lusk, and J. Boyle, *Automated Reasoning: Introduction and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1984.
[52] C.-F. Yu, "Efficient combinatorial search algorithms," Ph.D. dissertation, School Elec. Eng., Purdue Univ., West Lafayette, IN, Dec. 1986.
[53] M. B. Lowrie and B. W. Wah, "Learning heuristic functions for numeric optimization problems," in *Proc. Computer Software and Applications Conf.*, IEEE, Chicago, IL, Oct. 1988.
[54] G. DeJong and R. Mooney, "Explanation based learning: An alternative views," *Machine Learning*, vol. 1, no. 2, pp. 145-176, 1986.
[55] T. M. Mitchell, R. H. Keller, and S. T. Kedar-Cabelli, "Explanation-based generalization: A unifying view," *Machine Learning*, vol. 1, no. 1, pp. 47-80, 1986.

Chee-Fen Yu (S'82-M'87) received the B.E. (honors) degree in electrical engineering from the University of Malaya in 1980 and the M.S. and Ph.D. degree in electrical engineering from Purdue University, West Lafayette, IN, in 1983 and 1986, respectively.

He is currently a Senior Engineer in the Artificial Intelligence Laboratory of the Intel Corporation, Santa Clara, CA. His current research interests include artificial intelligence, computer algorithms, and hypermedia.



Benjamin W. Wah (S'74-M'79-SM'85) received the Ph.D. degree in engineering from the University of California, Berkeley, in 1979.

He was on the faculty of the School of Electrical Engineering at Purdue University, West Lafayette, IN, between 1979 and 1985. He is now an Associate Professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign, Urbana, IL. For 1988, he is on leave as a Program Director in the MIPS Division, National Science Foundation. His current research interests include parallel computer architectures, artificial intelligence, distributed databases, computer networks, and theory of algorithms.

Dr. Wah was a Distinguished Visitor of the IEEE Computer Society between 1983 and 1986. He is the Associate Editor-in-Chief of the forthcoming IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING and an Editor of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING and the *Journal of Parallel and Distributed Computing*. He served as Program Chairman (1987) and Conference Chairman (1988) of the IEEE International Conference on Data Engineering.