LEARNING DOMINANCE RELATIONS IN COMBINATORIAL SEARCH PROBLEMS

Chee-Fen Yu and Benjamin W. Wah

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 relations is presented. A classification of dominance relations and the applicable learning mechanisms have been identified. In addition, a study of learning dominance relations using learning-by-experimentation is described. This system has been able to learn dominance relations for the 0/1 knapsack problem, an inventory problem, the reliability-by-replication problem, the two-machine flowshop problem, a number of single-machine scheduling problems, and a two-machine scheduling problem.

INDEX TERMS: Combinatorial search, dominance relations, learning, learning-by-experimentation.

1. INTRODUCTION

Combinatorial search problems require the search of one or more solutions in a defined problem space. They may be classified into decision problems and optimization problems. A *decision problem* seeks solutions that satisfy a given set of constraints, while an *optimization problem* seeks the solution that satisfies the set of constraints as well as optimizes a given objective function. Examples of combinatorial search problems include theorem proving, game playing, evaluating a logic program, solving a database query, designing a computer system, assigning registers for a compiler, finding the shortest path in a graph, evaluating the maximum flow in a network, solving a mathematical programming problem, and searching for a permutation order to sort a set of numbers. These problems exist in a large spectrum of engineering and science applications, including artificial intelligence and operations research.

A search problem can be represented as a search graph, which is characterized by a root node that has no edge entering it, and one or more terminal nodes that has no edge exiting it. A special case of the search graph is the search tree in which each node except the root has exactly one edge entering it. A search graph can always be transformed into a search tree by duplicating common nodes. A branch in a search tree represents the assignment of a value to an unassigned parameter. For example, in the knapsack problem, the unassigned parameters are the set of unassigned items. In expanding a node, an unassigned item is selected, and two alternatives are created: (a) the item is included in the knapsack, causing the capacity of the knapsack to decrease by the size of the item and its profit to increase by the value of the item, and (b) the item is not included.

This research was supported by the National Science Foundation Grant NSF DMC 85-19649.

Chee F. Yu is with the School of Electrical Engineering, Purdue University West Lafayette, IN 47907.

Benjamin W. Wah is with the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois, 1101 W. Springfield Avenue, Urbana, IL 61801.

Computer Software and Applications Conference, 1986.

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

A node is *active* until it has been expanded, whereupon it becomes *dormant*. In a serial algorithm, the set of active nodes are maintained in a single list. A heuristic value defined by a heuristic function is computed for each node. The active node with the minimum heuristic value is always expanded first. A search is called a *depth-first search* if the negation of the level number is used as the heuristic function. In this case, the nodes in the active list are expanded in a last-in/first-out order. A search is called a *breadth-first search* if the level number is used as the heuristic function. In this case, the nodes in the active list are expanded in a first-in/first-out order. Lastly, a lower bound can be computed for each node in the active list. This represents the lower bound of the best solution that can be obtained from this node. By using the lower-bound function as the heuristic function, a *best-first search* expands the node with the minimum lower bound.

To reduce the search space in searching OR nodes in a search tree, *dominance relations* [5, 1, 12] are used to prune unnecessary expansions. 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, that is, a solution in T_i, which is no worse than the best solution in T_j, does not imply the presence of a dominance relation between N_i and N_j. Identifying good dominance relations are very important in combinatorial searches because many problem with good dominance relations are either polynomially solvable or pseudo-polynomially solvable, while the same set of problems without dominance relations are solvable in exponential time and/or space.

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 using the above dominance relations, the shortest-path problem is only solvable in exponential time.

One form of dominance relations is the lower-bound test, which is used in branch-and-bound algorithms [16]. Here, only the best solution node found so far in the search process is allowed to dominate other nodes. Any node with a lower bound not smaller than this value will not lead to a better solution and may be pruned. The pruning power of the lower-bound tests depends on the quality of the lower-bound function [26].

Dominance relations are problem-dependent and cannot be derived by a general procedure. Identifying and encoding problem-specific dominance relations is a painstaking and difficult process, particularly for complex problems. One possible solution to this problem is to automate the process of discovering domain-specific dominance relations by machine-learning techniques.

Carbonell, Michalski and Mitchell [7] have used the amount of inference performed by the learner as one method of classifying work in machine learning. In order of increasing amount of inference, the various types of learning mechanisms are rote learning, learning from instruction, learning from analogy, learning from examples, and learning from observation.

In rote learning, new knowledge is implanted directly as a computer program or as data. In learning from instruction, knowledge is acquired from a teacher or an organized source such as a textbook. In learning by analogy [8, 13, 29], new concepts are learned by transforming existing knowledge that bears a strong similarity to the desired new concept.

Learning from examples [2, 6, 10, 11, 20-22, 25, 27, 28] has been the most widely investigated form of machine learning. Here, the learner induces a general concept description from a set of examples or counter-examples of the concept. The learner must be provided with either a set of examples or an oracle to classify generated examples.

Learning from observation is characterized by problems that do not have the benefit of an external teacher or an oracle. In passive observation, the learner classifies observations of the environment. In active experimentation, the learner explores the environment, while applying different strategies as need arises. Lenat has studied active experimentation in mathematics [9] and heuristics [17-19]. Langley, Bradshaw and Simon have studied learning physical laws by active experimentation in the BACON system [14, 15].

Mitchell, et al. have studied the acquisition of problem-solving heuristics in the domain of symbolic integration using experimentation [23]. A four-step process is used: a practice problem is generated, available heuristics are used to solve the practice problem, the solutions are analyzed, and new domain-specific heuristics are proposed and refined to improve the performance of the system on subsequent problems. Similar work have been carried out by other researchers on games and puzzles [24, 15, 3, 4].

One common aspect of these work is the use of the solution sequence as a heuristic oracle to classify decisions. Any decision on the solution sequence is a positive example, and any decision that leads away from the solution sequence is a negative example. Heuristics are then induced from these examples using learning from examples. No such heuristic oracle exists for dominance relations. Each pair of nodes in the problem space is a potential example of a dominance relation, but the actual examples cannot be identified without explicit knowledge of the dominance relation.

In this paper, we present the results of our study of a machine learning system for dominance relations. We have classified dominance relations and have identified the applicable learning mechanisms. We have also studied in detail one of the learning mechanisms that we have identified and have developed a system to incorporate this mechanism to learn dominance relations for a number of different search problems.

2. METHODOLOGY

Winston [30] has suggested the following commandments of a good methodology in artificial intelligence.

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

We started with the problem of understanding how to learn dominance relations in combinatorial search problems. This covers an extremely wide spectrum of problem domains. Since little is known about the process of learning dominance relations, we have chosen the more prudent course of restricting the scope of our study to problems that are known to be rich in dominance relations. Many such problems are numerical optimization problems and are polynomially solvable.

This was followed by an examination into the selection of appropriate representations. A representation is a set of conventions about how to describe a class of objects. A description makes use of the conventions of a representation to describe some particular objects. In theory, all computer-based representations are identical. In practice, one scheme may be better another because it offers more convenience to the user. The much greater perspicuity and the inherent thinking advantages of powerful representations enable progress that would be impossible with anything less than adequate. A dominance relation is essentially an IF-THEN rule, so a production rule seems to be an appropriate representation for dominance relation. Appropriate representations for numerical optimization problems were also studied and are discussed in detail in Section Three.

Next, we searched for constraints in the problem domain which can simplify the learning task. Some constraints were found using a classification of dominance relations that exposes those relations with excessive overheads. Section Four covers this topic.

We then set out to devise the actual learning mechanisms and procedures. The methodology to do this consists of 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 are derived. Based upon the results obtained, it may be necessary to refine the problem domain classification and the overall description. The refinement tasks should be ordered in a manner reminiscent of a multi-tasking system. This approach recognizes the interaction among the different refinement tasks. Without periodic evaluations and task switching, resources may be wasted on a refinement task, which can be avoided if results from another refinement task were available. Section Five presents the results from the initial phase, and Six describes a system produced by the refinement of one of the identified learning mechanisms.

3. PROBLEM DESCRIPTION

The problems studied are numerical optimization problems. Three examples of numerical optimization problems are the 0/1-knapsack problem, the inventory problem, and the sequencing problem to minimize weighted completion time.

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

In sequencing to minimize the weighted completion time, we have a set T of n tasks. Each task $t \in T$ has a length $\theta(t)$, a weight $w(t)$, and a deadline $d(t)$. The desired solution is the initiation time $\sigma(t)$ in the single processor, which minimizes $\sum_{t \in T} [\sigma(t)+\theta(t)]w(t)$ and satisfies $[\sigma(t)+\theta(t)] \leq d(t)$ for $t \in T$.

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 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 $x(m_i)$, the amount purchased, and $y(m_i)$, the amount sold, in each period m_i, $1 \leq i \leq n$, such that $\sum_{i=1}^{n} [x(m_i)s(m_i) - y(m_i)b(m_i)]$ is maximized. There can at most be B units of the item at any time, so

$v+\sum_{i=1}^{j}[x(m_i)-y(m_i)] \leq B$ for all $j \in \{1, \ldots, n\}$. Amounts purchased are delivered at the end of each month, so the amount sold in each month must not exceed the stock at the end of the previous month, and $y(m_i) \leq v+\sum_{j=1}^{i-1}[x(m_j)-y(m_j)]$ for all $i \in \{1, \ldots, n\}$.

Although these three problems involve rather different domains, they are quite similar in form as seen from Table 1. Each problem description has at least one set of entities. A set of decisions must be made on each set of entities, and a solution is found when decisions have been made for every entity. Each description contains constraints that define the allowable sets of decisions, and an optimization criterion that defines the desirable optimal solution. The optimization involves the maximization or the minimization of an objective expression. The objective expression and the constraints are arithmetic expressions.

One way to classify the problems is by their desired decisions. A subset problem, such as the 0/1-knapsack problem, is one that seeks a subset with certain properties. A permutation problem, such as the single-machine scheduling problem, seeks a permutation that satisfies a given objective. A quantity problem, such as the inventory problem, is one in which the decisions are quantities of the entities.

There are three types of symbols in the descriptions: decision symbols, entity properties, and system constants. *Decision symbols* are associated with the set of decisions. *Entity properties* are associated with the set of entities, each of which can further be classified as a return property, a resource property, or a limit property. A return property is an entity property in the optimization criteria, whereas resource properties and limit properties appear in the constraints. Resource properties are usually combined with decision symbols in the constraints. Limit properties establish the limits in the constraints. It is possible for an entity property to be both a return and a resource property. The remaining symbols in the descriptions are system constants. *System constants* include set sizes, initial values, and global limits.

Before we can solve a search problem efficiently, we must devise a bookkeeping scheme that allows efficient orderly exploration of all possible solutions. A key in such a definition of the search space is the transformation of the set of decisions into a form suitable for efficient sequential decision making. Let this new set be known as the set of working decisions. For instance, the subset U' in the 0/1-knapsack problem has to be transformed into membership decisions x(u) on each item $u \in U$. The schedule in sequencing to minimize the weighted completion time is usually transformed into the position in the schedule $\rho(t)$ of each task $t \in T$. Such transformations usually require deep knowledge about the desired decisions. In the second example, it requires the recognition that tasks are scheduled without waiting in an optimal single-machine schedule.

Problem solvers make working decisions by assigning values to decision variables. Let a working description be the problem description resulted from supplanting the decision symbols with the decision variables. Working descriptions often have state variables that assist in assigning values to decision variables. State variables keep track of the effects of decisions that have already been made.

A complete specification of a working decision requires a two-tuple (*e-tuple*, *v-tuple*), where *e-tuple* identifies its entity, and *v-tuple* specifies its value. Decision variables in a *general-description* explicitly specify both the e-tuple and the v-tuple. For many problems, this is not necessary. A decision variable explicitly determines only the v-tuple of the corresponding working decision in a *value-description*. A common implementation assumes a fixed order of consideration of the entities $\{E_1, \ldots, E_{n_E}\}$, and the decision corresponding to decision variable v_i is (E_i, v_i). The use of this description results in a fixed-height search tree in which all solution nodes are at a fixed distance from the root. Figure 1(a) shows the value-description for the 0/1-knapsack problem.

Similarly, the decision variables only specify the e-tuple in an *identity-description*. One common implementation assigns a constant value to the v-tuple of the decision whose e-tuple is given by the decision variable. This is often combined with the assumption that the v-tuples of all unassigned decisions have some other constant values. Another widely used implementation assumes that the decision made by decision variable i_j is (i_j, j). The problem tree generated by this description may either be a fixed-height tree or a variable-height tree. Figure 1(b)

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

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

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

(a) Value description

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

(b) Identity description

Figure 1. Problem description of 0/1 knapsack problem.

shows the identity-description for the 0/1-knapsack problem.

4. A CLASSIFICATION OF DOMINANCE RELATIONS

Dominance relations may be classified by the possible positions of its dominating node and the possible positions of its dominated node. A dominance relation is described in our classification 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 the dominated nodes. The possible symbolic values for T_1 are 'terminal' and 'all.' When T_1 has the value 'terminal,' the dominating node must be a terminal node; when T_1 has the value 'all,' the dominating node can be any terminal or non-terminal node. T_2 can assume one of the following symbolic values: 'same,' 'specific,' or 'all.' When T_2 has a value of 'same,' the dominating node must be on the same level as the dominated node. When T_2 is 'specific,' the difference between the levels must be a symbolic constant. The dominated nodes can be any node in the problem tree if T_2 is 'all.' T_3 can either be 'sib' or 'all.' The dominating and dominated nodes must be siblings if T_3 is 'sib,' but can be any two nodes in the tree if T_3 is 'all.'

Table 2 lists examples of the different types of dominance relations. A relation of type {terminal, same, sib or all} is a trivial relation as the relation is between terminal or solution nodes. The values of solutions represented by terminal nodes are known, so it is trivial to determine which is the better solution. The lower-bound test is of type {terminal, all, all}. The dominating node in the lower-bound test is the best solution node that has been found so far, and the dominated node is any node whose lower bound is larger than the value of the dominating solution.

A greedy algorithm chooses the optimal value for the decision variable. This implies that the search tree generated is a skewed tree in which one node dominates all its siblings in each level of the search tree. This suggests that greedy algorithms are dominance relations of type {all, same, sib}. The upper-bound test in branch-and-bound algorithms 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 generally either trivial relations or relations that correspond to well-behaved relations in other descriptions. For example, in an item-description for the knapsack problem, each node in the problem space is a feasible solution and dominates its parent node. This relation corresponds to the trivial relation between terminal nodes for the corresponding value-based description.

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

Table 2. Examples of the different classes of dominance relations.

The implementation overhead for different types of dominance relations 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 (Table 3). The number of comparisons required by dominance relations between nodes on arbitrarily different levels is on the order of the number of active nodes. Unless the dominance relation is very powerful, the number of active nodes grows exponentially as the search proceeds, which diminishes the value of such dominance relations. In contrast, dominance relations belonging to classes {terminal, same, sib or all} are trivial relations. Hence, this study concentrates on dominance relations belonging to the classes {all, same, sib or all}.

5. APPLICABLE LEARNING MECHANISMS

Learning by rote can only be used for problems with known dominance relations and is useless for finding new dominance relations. Learning by instruction cannot be used as the absence of a general procedure for discovering dominance relations implies the absence of any organized source of knowledge about dominance relations. Moreover, there is no oracle that is able to identify positive and negative examples of an unknown dominance relation. Learning from examples is not directly applicable to this problem.

Dominance relations can be learned by analogy. However, analogy alone is insufficient as existing knowledge about dominance relations is limited. Moreover, analogy will not discover dominance relations dissimilar to existing dominance relations, nor will it work well for problems dissimilar to the problems with known dominance relations. Hence, analogy has to be conjugated with other mechanisms. One is learning by experimentation. Another is automated reasoning [31]. Here, dominance relations are deduced by logical inference from explicit axioms representing the problem description and implicit axioms representing implicit knowledge about the problem domain and the problem solution process. Figure 2 shows these three mechanisms and how they are related to each other.

Relations found by experimentation are only true for the set of sample problems considered during the experiments. Sample problems are usually heuristically generated, and those that contradict the derived relation may have been missed. A degree

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

Table 3. Upper bounds on the implementation overheads of different classes of dominance relations (LEV(A) is the level of node A).

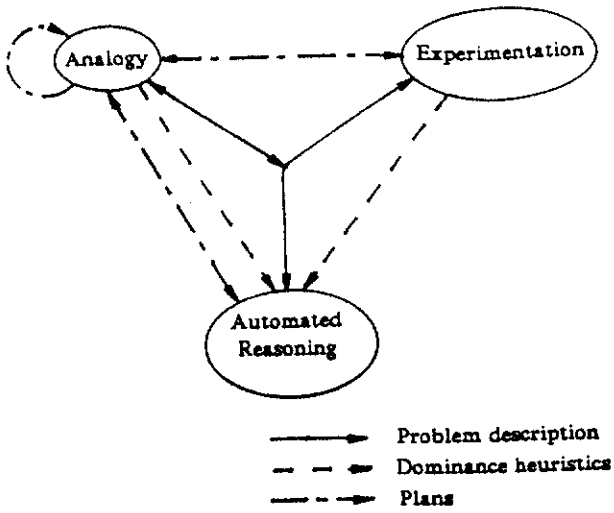

Figure 2. Mechanisms for discovering dominance relations in combinatorial searches.

of uncertainty also exists in relations derived by analogy. The match between a problem and its analogous problem is often inexact, and heuristics are used to transform the relations between problems. One possible approach towards the resolution of this uncertainty is to submit the relations as candidate relations to the automated reasoning component.

Analogy can be used to derive plans for experimentation, automated reasoning, and analogy itself. Plans derived by analogy are more robust than relations derived by analogy since they only suggest the strategies to be used in the learning mechanisms.

To learn dominance relations by experimentation, the system must be able to generate sample problems, solve the sample problems, and identify possible dominance relations in the search space. Sample problems can be solved with a weak problem solver. The recognition and testing of possible dominance relations in the search space is the key problem in the application of experimentation to discovering dominance relations. The generation of sample problems that are rich in examples of dominance relations is another important problem.

Analogy requires a pool of problems with known dominance relations and known plans, a similarity measure, and routines to transform relations based on the analogy. Many researchers have studied the general mechanisms needed to draw analogies and to transform relations between problems. The refinement of the analogy mechanism should, thus, concentrate on the domain-dependent aspects of the problem, including the similarity measure, the efficient representation of the problem descriptions, the dominance relations, and the plans.

Similarly, many general mechanisms for automated reasoning have been investigated and implemented in general automated reasoning systems. The domain-dependent aspects in automated reasoning are the explicit axioms representing the problem description, implicit axioms representing implicit knowledge about the problem domain and general problem solving, and preferences among the various axioms and the inference mechanisms. These domain-dependent aspects are key issues to be addressed in using automated reasoning.

A detailed study of learning dominance relations with experimentation has been carried out. Experimentation was chosen as it requires the least amount of a priori knowledge of dominance relations. The results of this study are presented in the next section.

6. MUTT: A SYSTEM FOR LEARNING DOMINANCE RELATIONS BY EXPERIMENTATION

Figure 3 shows the organization of MUTT. The key component of MUTT is the searcher that sifts through possible candidate relations for dominance heuristics. Candidate rules are tested by matching the rules against node pairs in test trees. The dominance heuristic is a relation for which testing finds no contradictory evidence. The forester is responsible for providing the searcher with appropriate test trees. The searcher will screen candidate relations until it either finds a dominance heuristic or exhausts its resource quantum. When this happens, the focus of activity returns to the scheduler. If the searcher has found a dominance heuristic, the scheduler finds out whether the heuristic can result in a linear search space. An affirmative answer implies the complete solution of the learning problem, and the system stops. Otherwise, the searcher resumes its screening of candidate relations.

Exhaustion of the searcher's resource quantum precipitates a re-evaluation of the effectiveness of the current problem description by the description manager and possibly a switch to a "better" description. Only when this has been completed will the scheduler assign a fresh resource quantum to the searcher and permit it to resume its screening of candidate relations.

Due to space limitation, only a sketch of the various components in MUTT is described here.

6.1. Description Manager

The transformation of the problem statement into an efficient working description requires deep domain-specific knowledge. The description manager in MUTT is incapable of carrying out automatic derivation of working descriptions. The user provides it with a set of useful working descriptions.

The effectiveness of each working description can be measured by the ratio of the number of dominance heuristics discovered in the description to the time spent in the description. The description manager makes sure that the description used by the searcher in each resource quantum is the most efficient one. A context switch is needed if it chooses a description different from the one used in the previous resource quantum. It involves saving the system state under the current description into a file and restoring the system state under the new description.

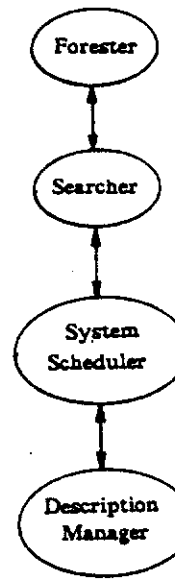

Figure 3. Structure of MUTT.

6.2. Forester

The forester supplies test trees to the searcher from its pool of test trees. New trees are generated if either the pool has been exhausted, or the trees in the pool do not satisfy the searcher's demands. These new trees will be added into the pool for use by future candidate relations.

Generating a new tree consists of creating a new problem instance and then solving this new problem instance with a weak problem solver. The forester creates a new problem instance by creating a new frame instance and filling in the slots. The problem description contains the set of legal value for each slot. Random values are used to fill the slots initially. Some of these random values may have to be modified later when the candidate relation has equality conditions to increase the number of nodes in the test tree which satisfy the conditions of the candidate relation.

6.3. Searcher

MUTT represents candidate relations by production rules of the following form:

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

The implication of the rule identifies the dominating node, x, and the dominated node, y, by the relationship between the values of their best solutions, B(x) and B(y). The type condition (t-condition) identifies the rule type. The dominance conditions (d-conditions) specify 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. The set of possible symbols consists of the state variables, entity properties, system constants, functions of other symbols, and numbers.

Let an example of a rule be any node pair that satisfy the conditions of the rule. An example is a supporting example if it agrees with the implication of the rule and a contradicting one if it disagrees with the implication of the rule. An acceptable rule is a rule for which testing finds only supporting examples. The acceptable rule, thus, corresponds to a dominance heuristic. A rejected rule is a rule which has contradicting examples. An impossible rule is a rule with no examples.

The process of finding an acceptable rule involves a systematic search in a rule space, which may be organized as a simple directed graph. Each edge represents the addition of a d-condition, and its terminal vertex represents the rule formed by adding the d-condition represented by the edge to the rule represented by its initial vertex.

6.4. Generate-and-Test

The basic generate-and-test scheme consists of two modules: a generator and a tester. The generator enumerates possible relations. The tester evaluates each proposed relation, either accepting it or rejecting it.

There are two types of generators: the exhaustive generator and the informed generator. The simpler exhaustive generator generates all possible rules, while an informed generator uses possibility-limiting information to restrict the relations that it proposes. An exhaustive generator may be used when the number of possible rules is small. As the number of possible rules increases, the use of an informed generator becomes increasingly desirable.

Figure 4 shows the design of the generate-and-test process. In this design, the generator cooperates with the cartographer and the scrounger to ensure that only reasonable candidate rules are proposed. The task of the cartographer is to create a series of maps, each of which represents a sub-graph of the rule graph that is believed to contain acceptable rules.

The generate-test-scrounge cycle is responsible for exploring the individual rules in the map. It is similar to the generate-test cycle in that the generator still selects rules for testing, and the tester still determines the decision value of the proposed rule. However, there is now also a scrounger that is used to determine the effects of the tester's decision on other rules. These effects reduce the number of rules to be proposed and tested. The generate-test-scrounge cycle repeats until the map has been completely explored.

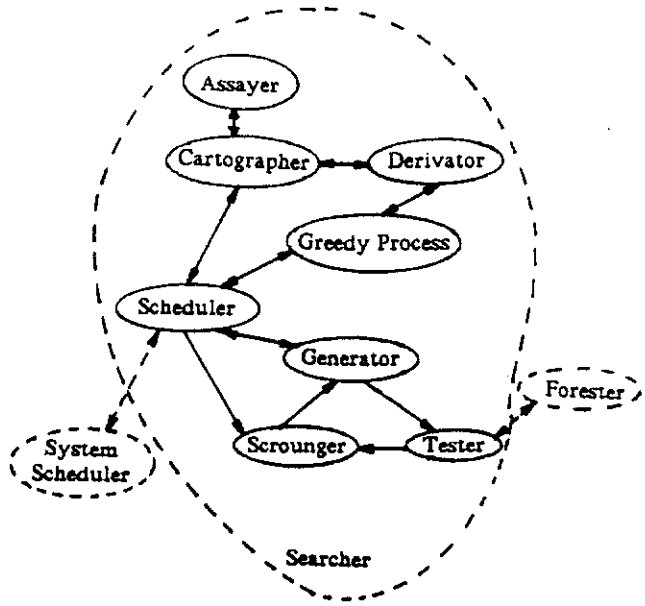

Figure 4. Structure of the Searcher.

When this happens and the resource quantum allocated to the searcher has not yet been exhausted, the scheduler will call upon the cartographer to expand the current map by including previously unexplored regions of rule space. Once the cartographer has completed the new map, the generate-test-scrounge cycle resumes.

As mentioned in Section Four, greedy algorithms correspond to dominance relations of type {all, same, sibling}. Many greedy algorithms contain only entity properties. Let rules with only d-conditions between entity properties be known as greedy rules. The generate-test-scrounge cycle cannot detect greedy rules in a value-description due to its limited horizon. Only equi-level nodes are considered, and these nodes are all associated with the same entity in a value-description. This prevents the generation of rules involving entity properties of different entities. The greedy process is used to overcome this deficiency.

8. CONCLUDING REMARKS

In this paper, a methodology for machine learning of dominance relations has been proposed and studied. One result of this study is a classification of the dominance relations based upon the permissible positions of the dominating and dominated nodes. Another result is a characterization of the applicable problem representations. An overall description of the applicable learning mechanisms has also been derived. These include analogy, automated reasoning, and experimentation. The framework of a system for learning dominance relations using experimentation is described. The system can learn dominance relations for the 0/1-knapsack problem, inventory problem, reliability-by-replication problem, two-machine flowshop problem, several single-machine scheduling problems, and a two-

machine scheduling problem. Due to space limitation, these results are not presented here. To apply the methodology to other combinatorial search problems requires the identification of the necessary meta-information. At present, this system is being implemented.

REFERENCES

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

[2] P. M Andrae, "Constraint Limited Generalization: Acquiring Procedures from Examples," *Proc National Conference on Artificial Intelligence*, pp. 6-10, AAAI, 1984.

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

[4] A. A. Araya, "Learning Problem Classes by Means of Experimentation and Generalization," *Proc National Conference on Artificial Intelligence*, pp. 11-15, AAAI, 1984.

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

[6] B. G. Buchanan and T. M. Mitchell, *Model-Directed Learning of Production Rules Pattern-Directed Inference Systems*, Academic Press, 1978.

[7] J. G. Carbonell, R. S. Michalski, and T. M. Mitchell, "An Overview of Machine Learning," in *Machine Learning*, ed. R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Tioga, 1983.

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

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

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

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

[12] T. Ibaraki, "The Power of Dominance Relations in Branch-and-Bound Algorithms," *J. of the ACM*, vol. 24, no. 2, pp. 264-279, ACM, April 1977.

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

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

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

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

[17] D. B. Lenat, "The Nature of Heuristics," *Artificial Intelligence*, vol. 19, no. 2, pp. 189-249, North-Holland, 1982.

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

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

[20] R. S. Michalski, "A Theory and Methodology of Inductive Learning," in *Machine Learning*, ed. R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Tioga, 1983.

[21] S. Minton, "Constraint-Based Generalization: Learning Game-Playing Plans from Single Examples," *Proc. National Conf. on Artificial Intelligence*, pp. 251-254, AAAI, 1984.

[22] T. M. Mitchell, "Generalization as Search," *Artificial Intelligence*, vol. 18, pp. 203-226, 1982.

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

[24] S. Ohlsson, "A Constrained Mechanism for Procedural Learning," *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 426-428, William Kaufman, Los Altos, CA, 1983.

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

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

[27] D. A. Waterman, "Generalization Learning Techniques for Automating the Learning of Heuristics," *Artificial Intelligence*, vol. 1, no. 1/2, pp. 121-170, North-Holland, 1970.

[28] P. H. Winston, "Learning Structural Descriptions from Examples," in *The Psychology of Computer Vision*, ed. P. H. Winston, McGraw Hill, New York, NY, 1975.

[29] P. H. Winston, "Learning and Reasoning by Analogy," *Comm. of the ACM*, vol. 23, pp. 869-703, 1979.

[30] P. H. Winston, *Artificial Intelligence*, Second Edition, Addison-Wesley, 1984.

[31] L. Wos, R. Overbeek, E. Lusk, and J. Boyle, *Automated Reasoning: Introduction and Applications*, Prentice-Hall, 1984.