INTELLIGENCE IN NUMERICAL COMPUTING: IMPROVING BATCH SCHEDULING ALGORITHMS THROUGH EXPLANATION-BASED LEARNING

Matthew J. Realff[1]

School of Chemical Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332

[1]The work reported in this chapter was carried out while the author was a Ph.D. student in the Laboratory for Intelligent Systems in Process Engineering, Department of Chemical Engineering, Massachusetts Institute of Technology, Cambridge, MA 02139.

Learning comes from reflection on accumulated experience and the identification of patterns found among the elements of previous experience. All numerical algorithms used in scientific and engineering computing are based on the same paradigm: *Execute a predetermined sequence of calculational tasks and produce a numerical answer*. The implementation of the specific numerical algorithm is oblivious to the experience gained during the solution of a specific problem, and the next time a different, or even the same, problem is solved through the execution of exactly the same sequence of calculational steps. The numerical algorithm makes no attempt to reflect on the structure and patterns of the results it produced, or reason about the structure of the calculations it has performed. This chapter shows that this need not be the case. By allowing an algorithm to reflect on and reason with aspects of the problems it solves and its *own structure of computational tasks*, *the algorithm can learn* how to carry out its tasks more efficiently. This form of *intelligent numerical computing* represents a new paradigm, which will dominate the future of scientific and engineering computing. But, in order to unlock the computer's potential for the implementation of truly intelligent numerical algorithms, the *procedural* depiction of a numerical algorithm must be replaced by a *declarative* representation of the algorithmic logic. Such a requirement upsets an established tradition and imposes new educational challenges which most educators and educational curricula have not, as yet, even recognized. This chapter shows how one can take a branch-and-bound algorithm, used to identify optimal schedules of batch operations, and endow it with the ability to learn to improve its own effectiveness in locating the optimal scheduling policies for flowshop problems. Given that most batch scheduling problems are NP-hard, it becomes clear how important it is to improve the effectiveness of algorithms for their solution. Using the framework of Ibaraki (1978) a branch-and-bound algorithm is declaratively modeled as a *discrete decision process*. Then, *explanation-based machine learning* strategies can be employed to uncover patterns of generic value in the experience gained by the branch-and-bound algorithm from solving specific instances of scheduling problems. The logic of the uncovered patterns (i.e., new knowledge) can be incorporated into the control strategy of the branch and bound algorithm when the next problem is to be solved.

I. Introduction

The main goal of this chapter is to introduce the concept of *intelligent numerical computing* within the context of solving optimization problems

of relevance to chemical engineering. Before focusing on this area, it is worth noting that intelligent scientific computing has been the subject of research within the context of other problems such as bifurcation analysis and nonlinear dynamics, and chemical reaction kinetic schemes (Abelson et al., 1989; Yip, 1992), both of which are clearly subjects of interest to chemical engineers.

Our first task is to define intelligent numerical computing and to indicate what features it possesses to make it a distinct subclass of general numerical computing. For us, the essence of the distinction is the difference between *reasoning* and *calculation* (Simon, 1983). Implementations of numerical algorithms, such as the simplex method for linear programming (Dantzig, 1963), or gradient descent methods for more general nonlinear optimization problems (Avriel, 1976), take a problem formulated as a set of algebraic relationships and calculate an answer. Following performance of the calculation task, the implementation is completely oblivious to the *experience*, and the next time a different, or even the same, problem is solved exactly, the same set of steps will be performed. The implementation makes no attempt to reflect on its experience, or reason about the calculation it performed; it is completely oblivious to its own structure, which has not been declaratively represented, but that is procedurally embodied in the set of calculations performed. *Intelligent numerical computing* attempts to incorporate reasoning into the calculation procedure by allowing the computer to reason and reflect on various aspects of the problem, and its own structure. To do this we need to adopt new representations of information pertinent to problem solving, in particular symbolic representations, and to introduce new algorithms, which compute with symbolic as opposed to numerical information, such as natural deduction and resolution (Robinson, 1965) for logic-based reasoning.

To illustrate these concepts we will focus on a particular numerical algorithm, branch and bound (Nemhauser and Wolsey, 1988), which has been a workhorse for solving optimization problems with discrete structure, such as chemical batch scheduling (Kondili et al., 1993; Shah et al., 1993). This is a generic problem-solving strategy, and its successful application relies on the identification of effective *control information*, often in (1) the form of a lower-bound function, which characterizes how good an incomplete solution can be, and (2) as dominance and equivalence conditions, which use information about one partial solution to terminate another. In this chapter, we will present a methodology for *automating* the acquisition of this control knowledge for branch-and-bound algorithms, using flowshop scheduling as an illustration. Here, automation means that the *computer itself* acquires new control information, unaided by the human user, save for the specification of the problem. The acquisition is

carried out by analyzing the experience gained by the computer during problem solving, illustrating the incorporation of one facet of reasoning, *learning from experience*.

The rest of the chapter will focus on solving four goals, engendered by the problem of automatically improving problem-solving experience.

1. Before we can learn new control information, we must formally specify what it is we are trying to learn and how what we learn is to be applied within the original problem-solving framework (Section II).
2. Having established the formal specification of the problem-solving framework, the next goal is to establish how the solution of example problems can be turned into relevant problem-solving experience (Section III).
3. To convert the problem-solving experience into a useful form, we need to be able to represent it to the computer, along with the other information necessary to reason about it efficiently (Section IV).
4. Finally, having set up the learning problem, we need to employ a learning method that will guarantee preservation of the correctness of the branch-and-bound algorithm and make useful additions to the control information we have about the problem (Section V).

These four goals are addressed sequentially in the next four sections. The flowshop problem will be used as an illustration throughout, because of its practical relevance, difficulty of solution, and yet relative simplicity of its mathematical formulation.

A. Flowshop Problem

Many chemical batch production facilities are dedicated to producing a set of products that require for their manufacturing a common set of unit operations. The unit operations are performed in the same sequence for each product. This type of production problem is often solved by configuring the available equipment so that each unit operation is carried out by a fixed set of equipment items that are disjoint from those used in any other operation. If one unit is assigned to each step, this is called a *flowshop* (Baker, 1974).

The flowshop problem has been widely studied in the fields of both operations research (Lagweg et al., 1978; Baker, 1975) and chemical engineering (Rajagopalan and Karimi, 1989; Wiede and Reklaitis, 1987). Since the purpose of this chapter is to illustrate a novel technique to synthesize new control knowledge for branch-and-bound algorithms, we

will adopt the simplest, and most widely studied, form of the flowshop problem. The assumptions with respect to the problem structure are given below.

1. *Storage policy*. Much attention has been paid to exploring the implications of different storage policies, such as having no intermediate storage, finite intermediate storage, running the plant with a zero-wait policy, or combinations of the policies discussed above. We will assume unlimited intermediate storage is available to simplify the constraints between batches.
2. *Clean out, set up, and transfer policies*. In general, these operations could be dependent on the order in which the products are routed through the equipment. We will assume that these operations are *sequence independent* and can be factored into the processing times for each step.

With these simplifications, we can now formulate the mathematical model, which describes the flowshop. Let

$p_{ik} \equiv$ the processing time of job i on machine k,
$c(\sigma, k) \equiv$ the end-time of the last job of sequence σ on machine k.

Then, to calculate the end-time of an operation of job a_i on machine k, when a_i has been scheduled after a sequence σ, we use the following equations:

Machine 1:

$$c(\sigma a_i, 1) = c(\sigma, 1) + p_{a_i 1}. \tag{1}$$

Machine k:

$$c(\sigma a_i, k) = \max\{c(\sigma, k), c(\sigma a_i, k-1)\} + p_{a_i k} \quad k = 2 \ldots m. \tag{2}$$

To generate a specific instance of a flowshop problem we will assume that the plant produces a fixed set of products. In addition to allowing the type of product to vary, we will also allow the size of the batch to be one of a fixed set of sizes. Further details of the formulation are given in Section III, A.

B. Characteristics of Solution Methodology

To solve the flowshop scheduling problem, or indeed most problems with significant discrete structure, we are forced to adopt some form of

enumeration to find, and verify, optimal solutions (Garey and Johnson, 1979). To perform this enumeration, we must solve two distinct problems.

1. *Representation*. The solution space is composed of discrete combinatorial alternatives of batch production schedules. For example, in the permutation flowshop problem, where the batches are assumed to be executed in the same order on each unit, there are $N!$ number of solutions, where N is the number of batches. We must find a way to compactly represent this solution space, in such a way that significant portions of the space can be characterized with respect to our objective as either "poor" or "good" without explicitly enumerating them.
2. *Control strategy*. Having found an appropriate representation for the space of solutions, we must then take advantage of this representation to explore only as small a fraction of the space as possible. This involves the following:
   (a) Finding a method to systematically enumerate subsets of the solution space without explicitly enumerating their members.
   (b) Finding ways to prove that certain subsets of the space cannot contain optimal solutions, or if they do, that we have other subsets of the space that will produce equivalent solutions.

To solve the problems of representation and control, we will employ the framework of the branch-and-bound algorithm, which has been used to solve many types of combinatorial optimization problems, in chemical engineering, other domains of engineering, and a broad range of management problems. Specifically, we will use the framework proposed by Ibaraki (1978), which is characterized by the following features:

1. The combinatorial problem is represented by a *discrete decision process* (DDP) (Ibaraki, 1978) where the underlying information in the problem is captured by an explicit *state-space model* (Nilsson, 1980).
2. The control is divided into three parts.
   (a) Elimination of alternative solutions through the use of a *lower bound* on the value of the objective function.
   (b) Elimination of alternative solutions through *dominance* conditions.
   (c) Elimination of alternative solutions through *equivalence* conditions.

The next section will highlight these features of the branch-and-bound framework, within the context of the flowshop scheduling problem. Then we will give an abstract description of the algorithm, followed by the

paraphrasing of an important theorem from Ibaraki (1978), which characterizes the relative efficiency of different branch-and-bound algorithms in terms of their control strategies.

II. Formal Description of Branch-and-Bound Framework

If an algorithm is to reflect on its own computational structure and thus *learn* how to improve its performance, then this algorithm needs to have a *declarative* representation of its components. Thus, a branch-and-bound algorithm used to generate optimal schedules of batch operations should possess declarative representations of (1) the schedules of batch operations (2) the predicates that determine the feasibility of a scheduling policy, (3) the objective function, and (4) the conditions that determine the control strategy of the branch-and-bound algorithm. The material in this section will provide such declarative representations for the various components of branch-and-bound algorithms, and will introduce a metric that can be used to measure improvements in the efficiency of such algorithms.

A. Solution Space Representation—Discrete Decision Process

The first step in solving a combinatorial optimization problem is to model the solution space itself. Such a model should be declarative in character, if it is to be independent of the characteristics of the specific algorithm that will be used to find the solution within the solution space. The model we have adopted for the scheduling of flowshop operations is the *discrete decision process* (DDP) introduced originally by Karp and Held (1967). As defined by Ibaraki (1978) a DDP, Y, is a triple (Σ, S, f) with its elements defined as follows:

Σ is a finite nonempty alphabet whose symbols are used to build the description of solutions. Let Σ^* denote the set of finite strings generated from the concatenation of symbols present in Σ.

S is a subset of Σ^* whose members satisfy a set of feasibility predicates over Σ^*. The members of Σ^* will represent the "feasible" solutions.

f is a cost function over S, mapping S into the set of real numbers.

For the flowshop problem, one choice of Σ is an alphabet with as many symbols as the number of distinct batches to be scheduled, i.e., one symbol for each batch, then, each discrete schedule of batches is represented by a

finite string, σ, of symbols drawn from the alphabet, Σ. Only strings which contain each alphabetic element once and only once represent acceptable schedules of batch operations. We will use the notation σ to denote the set of symbols from Σ, which are contained in the solution string, Σ^*. Also, with $O(Y)$ we will denote the set of optimal solutions of the DDP, Y.

The DDP is a formalism more general than the customary formalisms of the combinatorial optimization problems and offers an excellent framework for unifying the formalization of very broad classes of such problems (Karp and Held, 1967; Ibaraki, 1978). It also exhibits many common features with the *state-space representation* of problems commonly employed by researchers in artificial intelligence (AI) (Kumar and Kanal, 1983; Nilsson, 1980). This is a feature that we will find very convenient in subsequent sections as we try to integrate machine learning algorithms, which use the state space representation, with branch-and-bound algorithms solving combinatorial optimization problems.

Thus, it can be effectively argued that the DDP formalism is superior to other formalisms since it can uniformly accommodate

1. A variety of problems seeking the set of feasible solutions, the set of optimal solutions, or one member of either set.
2. Various types of alphabets (used to describe solutions), predicates (used to describe feasibility of solutions), and cost functions.

1. Illustration: Flowshop Example

Let

$\Sigma \equiv$ {all batches}.

$S \equiv$ Set of all σ_x, if $\{\sigma_x\}$ contains all symbols of Σ once and only once.

$f(x) \equiv C(\sigma_x, m)$, completion time of the last operation of the final batch.

In this alphabet, each batch is assigned its own symbol. The problem formulation allows for the same combination of product and size to be selected multiple times. Hence the schedules that have the same batch type in two or more different positions will be enumerated multiple times, even though they represent schedules which are indistinguishable from one another.

A more compact alphabet for this situation is one for which we create a unique symbol for each batch type, and allow the multiple occurrence of this symbol to stand for scheduling the batch type more than once. The feasibility predicate would be suitably modified to check to see when enough of the type had been added to a given branch. This gives the

following DDP:

$\Sigma \equiv \{N$ distinct symbols: $N =$ all (product size) combinations$\}$.

$S \equiv$ Set of all σ_x, such that $\{\sigma_x\}$ contains all the symbols of Σ a number of times equal to the number of batches for each product-size combination that was selected for production.

$f \equiv C(\sigma_i, m)$ completion time of the last operation on the final machine.

The elimination of spurious equivalent solutions is important computationally, because either we will have to enumerate the spurious equivalent solutions, doubling the effort for each equivalent pair of solutions, or we will have to introduce rules that can detect the equivalence explicitly. For the purpose of illustrating the ideas of this chapter we will continue to use the naive alphabet, although the method is not restricted to such a choice.

A scheduling policy, x, is optimal if and only if the following conditions are satisfied:

Condition 1. Scheduling policy is feasible, i.e., $\sigma_x \in S$.

Condition 2. Scheduling policy has cost not higher than that of an other policy, y, i.e., $f(x) \leq f(y)$ for any y such that $\sigma_y \in S$.

B. The Branch-and-Bound Strategy

1. Branching in General

The large size of the solution space for combinatorial optimization problems forces us to represent it *implicitly*. The branch-and-bound algorithm encodes the entire solution space in a *root node*, which is successively expanded into branching nodes. Each of these nodes represents a subset of the original solution space specialized to contain some particular element of the problem structure.

In the flowshop example, the subsets of the solution space consists of subsets of feasible schedules. We can organize these subsets in a variety of ways; for example, fixing any one position of the N available positions in the schedule to be a particular batch creates N subsets of size $(N-1)!$. Subsequently, as we fix more and more of the positions, the sets will include fewer and fewer possibilities, until all the positions are fixed, and we have a single element in the set corresponding to a single, feasible schedule.

2. Formal Statement of Branching

The branching, or specialization of the solution subsets, should obey certain constraints to avoid potential problems of inefficiency, nontermination, or incorrectly omitting solutions.

1. *Mutual exclusivity.* In branching from one node to its descendants, we should ensure that none of the subsets overlap with one another; otherwise we could potentially explore the same solution subsets in multiple branches of the tree. Formally, if X is the original solution space and x_i is the ith subset, then
$$x_i \cap x_j = \varnothing \; \forall i, j > i$$
2. *Inclusiveness.* In branching to the descendants, we should ensure that no solution of the parent set is omitted from the child subsets. If we fail to have inclusiveness, the procedure may not correctly find the optimum solution. Formally, this requirement implied that
$$\bigcup_i x_i = X.$$

We must now link these two properties, and the notion of branching to our problem representation, i.e., the DDP formalism. To do this, we introduce a variant on our earlier notation; let

$$Y(x) \equiv [\Sigma, S(x), f],$$

where $S(x)$ is the feasibility predicate over all strings that begin with the partial sequence of symbols, x.

If ε denotes the empty strings, then

$$Y(\varepsilon) \equiv (\Sigma, S, f)$$

and $Y(\varepsilon)$ is equivalent to the original problem, since $S(\varepsilon)$ includes all the original set of solutions. Thus $Y(\varepsilon)$ is the problem associated with the root node and branching from the root node creates problems $Y(a_1)$ $Y(a_2) \cdots Y(a_n)$, one for each of the n elements of the alphabet. Each problem, $Y(a_i)$ captures a subset of the original problem, specialized by the inclusion of the alphabet symbol. If $\Sigma = \{a_1, a_2, \ldots, a_n\}$, then a branch and-bound algorithm decomposes $Y(\varepsilon)$ into $|\Sigma|$ problems, $\{Y(a_1)$ $Y(a_2), \ldots, Y(a_n)\}$, where the solution strings are required to start with the respective alphabetic element. We will also extend the definition of the objective function values as follows:

$$f(x) = \begin{cases} \inf\{f(xy) : \sigma = xy \in S\} \\ \infty \text{ otherwise, i.e., } Y(x) \text{ infeasible} \end{cases}$$

According to this extension of the cost function definition, the following two conditions are always satisfied:

(a) $f(xy) \geq f(x)$ for $x, y \in \Sigma^*$.
(b) $f(x) = \min\{f(xa): a \in \Sigma\}$ for $x \notin S$.

3. *Illustration: Flowshop Problem*

With our alphabet equal to the set of batches to be scheduled, we will interpret ε, the empty string, to be the empty schedule. $Y(a_j)$ is then the problem $Y(\varepsilon)$ specialized to have schedules that "begin with" alphabet element a_j in the first position of the schedule. Thus, the discrete decision process $Y(a_j)$ is equivalent to solving the original scheduling problem with the first batch fixed to be a_j. The set $S(a_j)$ includes all feasible solutions that have a_j as their first batch, and, in our naive formulation, $S(a_j)$ differs from $S(\varepsilon)$ by prohibiting a_j to appear again in the string.

We can see that, in general, the branching process satisfies the property of inclusiveness, since we generate a subset for each possible specialization of the original set. We cannot, however, claim exclusiveness unless we are guaranteed that each string maps to a unique solution subset. For example, if an alphabet element corresponds to assigning a batch to a specific position, then the order in which batches are assigned their positions is irrelevant, hence, with that alphabet, $Y(a_i a_j) \equiv Y(a_j a_i)$.

Having defined the process of branching, we must now formally define the mechanisms for controlling the expansion of the subsets. The basic intuition behind each of these mechanisms is that we can "measure" the quality not just of a single feasible solution, but of the entire subset represented by the partial solution string.

4. *Lower-Bound Function*

Having formally defined the branching structure, we must now make explicit the mechanisms by which we can eliminate subsets of the solution space from further consideration. Ibaraki (1978) has stated three major mechanisms for controlling the evolution of the branch-and-bound search algorithms, by eliminating potential solution through

(a) Comparison with an estimate of the objective function's lower bound.
(b) Dominance conditions.
(c) Equivalence conditions.

In this section we will discuss the characteristics of the first mechanism, leaving the other two for the subsequent two sections.

In our scheduling example, a partial solution represents all the possible completions of a partial schedule. To estimate the quality of the partial solution, we could assume the best scenario, and assign it a lower-bound value based on the lowest possible value of its makespan. There are many ways to estimate the makespan; for example, we could simply ignore the remainder of the batches, and report the makespan of the current partial schedule, clearly a lower bound on the final makespan. Stated formally, the lower-bound function $g(x)$ satisfies the following requirements (Ibaraki, 1978):

(a) $g(x) \leq f(x)$
(b) $g(x) = f(x) \quad x \in S$
(c) $g(x) \leq g(xy) \quad$ for $x, y \in \Sigma^* \quad x \in \Sigma^*$

If $g(x)$ satisfies these conditions, we can use the following *lower-bound elimination* criterion to terminate the solution of a discrete decision process, $Y(y)$.

Condition: If $x \in S$ and $g(y) \geq f(x)$, then all solutions of $Y(y)$ can be discarded.

Clearly, since $g(y) \leq f(y)$, it follows that $f(y) \geq f(x)$, and hence the solution of $Y(y)$ cannot lead to a better objective function value.

The disadvantage of the lower-bound elimination criterion is that it cannot eliminate solutions with lower bounds better than the optimal solution objective function value.

Since

$$f(x^*) \leq f(x) \; \forall x \, x^* \in \text{optimal solution set}, O[Y(\varepsilon)]$$

and

$$\text{If } g(y) < f(x^*)$$

$$\text{then } g(y) < f(x) \; \forall x$$

and the condition for lower-bound elimination cannot be satisfied.

The efficiency of branch-and-bound algorithms is profoundly influenced by the *strength* of a lower-bounding scheme or function. In general, the closer a lower-bound function value of a node is to the true objective function value, the fewer nodes will be expanded, and the more efficient the algorithm will be. For example, in solving mixed-integer linear programs (MILP) via branch and bound, if the lower-bound scheme employed is a linear programming (LP) relaxation of the integer variables, then the efficiency of the solution technique is a strong function of how closely the

LP objective function value approximates the best integer solution value that could be generated by solutions emanating from that node.

If using a stronger lower-bound function leads to the expansion of fewer nodes, why not always use the strongest lower bound? There are two potential problems with this. First, in many cases, it is impossible to prove that one lower-bounding function, f_1, yields a higher value for minimization than another, f_2, for every node in the tree. Second, stronger lower bounds tend to require more computational effort to evaluate; thus, even though fewer nodes are evaluated, they take longer to evaluate. This tradeoff can lead to weaker lower bounds, yielding better overall computational efficiency.

In addition to the elimination of partial solutions on the basis of their lower-bound values, we can provide two mechanisms that operate directly on pairs of partial solutions. These two mechanisms are based on *dominance* and *equivalence conditions*. The utility of these conditions comes from the fact that we need not have found a feasible solution to use them, and that the lower-bound values of the eliminated solutions do not have to be higher than the objective function value of the optimal solution. This is particularly important in scheduling problems where one may have a large number of equivalent schedules due to the use of equipment with identical processing characteristics, and many batches with equivalent demands on the available resources.

5. *Dominance Test*

The intuitive notion behind a *dominance condition*, D, is that by comparing certain properties of partial solutions x and y, we will be able to determine that for every solution to the problem $Y(y)$ we will be able to find a solution to $Y(x)$ which has a better objective function value (Ibaraki, 1977). In the flowshop scheduling problem several dominance conditions, sometimes called elimination criteria, have been developed (Baker, 1975; Szwarc, 1971). We will state only the simplest:

Condition: If $\{x\} = \{y\} \wedge \forall i\ C(x,i) < C(y,i)$ then $x.D.y$.

For example, if the completion times of the partial schedule x on each machine i are less than those of the partial schedule y on each machine i, and x and y have scheduled the same batches, then, if we complete the partial schedules x and y in the same way, the completion time of any schedule emanating from x will be less than the completion time of the same schedule emanating from y; that is,

$$\forall u\ C(xu,i) < C(yu,i),$$

and thus, specifically

$$C(xu, m) < C(yu, m),$$

and the makespan of schedule, xu, will be less than that of yu.

Formally, we define the dominance condition by specifying a set of necessary and sufficient conditions, which it has to satisfy (Ibaraki, 1978):

Definition. A dominance relation, D, is a partial ordering of the partial solutions of the discrete decision processes in Σ^*, which satisfies the following three properties for any partial solutions, x and y:

1. Reflexivity $x.D.x$
2. Transitivity $x.D.y \wedge y.D.z \rightarrow x.D.z$
3. Antisymmetry $x.D.y \rightarrow \neg(y.D.x)$

Furthermore, a dominance relationship is constructed in such a way that it satisfies the following properties:

4. $x.D.y$ and $x \neq y \Rightarrow f(x) < f(y)$
5. $x.D.y$ and $x \neq y \Rightarrow g(x) < g(y)$
6. $x.D.y$ and $x \neq y \Rightarrow \forall u \in \Sigma^* \exists v \in \Sigma^*$ such that $xu.D.yv$ and $xv \neq yu$
7. For each $x \in \Sigma^*$ $\exists y \in \Sigma_D^*$ such that $y.D.x$

Note that the set Σ_D^* is defined such that it contains all solutions that have not been dominated by any other solution. Property 4 guarantees that we will not miss an element of the optimal set, since the objective function value of x is lower than that of y.

6. *Equivalence Test*

The final relationship between solution subsets, which we can use to curtail the enumeration of one subset, is an *equivalence condition*, EQ. Intuitively, equivalence between subsets means that for every solution in one subset, y, we can find a solution in the subset x, which will have the same objective function value, and that plays a similar role in the execution of the algorithm.

In the case of the flowshop example, we have the following equivalence condition

Condition: If $\{x\} = \{y\} \wedge \forall i\ C(x, i) = C(y, i)$ then $x.EQ.y$.

This condition implies that any complete schedule, xu, emanating from the partial schedule, x, will have the same end-times on all machines as the completions, yu, of y, not just for the completed schedule, but also for all intermediate partial schedules.

Formally, we define the equivalence condition by specifying a set of necessary and sufficient conditions, which it has to satisfy (Ibaraki, 1978).

Definition. An equivalence condition, EQ, between two partial solutions, $x \in \Sigma^*$ and $y \in \Sigma^*$ is a binary relationship, $x.EQ.y$, which has the following properties:

1. Reflexivity $\quad x.EQ.x$
2. Transitivity $\quad x.EQ.y \wedge y.EQ.z \rightarrow x.EQ.z$
3. Symmetry $\quad x.EQ.y \rightarrow y.EQ.x$

and satisfies the following conditions.

Condition a:

$$x.EQ.y \Rightarrow \forall u \in \Sigma^* \begin{cases} xu \in S \Leftrightarrow yu \in S, \\ f(xu) = f(yu). \end{cases}$$

Condition b:

$$x.EQ.y \Rightarrow \forall u, w \in \Sigma^* \begin{cases} g(xu) = g(yu), \\ xu.EQ.w \Leftrightarrow yu.EQ.w, \\ xu.D.w \Leftrightarrow yu.D.w, \\ w.D.xu \Leftrightarrow w.D.yu. \end{cases}$$

These two definitions of dominance and equivalence have been stated in the context of seeking the optimal solution set $O(Y(\epsilon))$. If we are trying to seek only a single optimal solution, we can merge the definitions of dominance and equivalence into a single relationship, since we no longer need to retain solutions that might potentially have equal objective function values, but that are not strictly equivalent. In our flowshop example, this is equivalent to allowing the end-times of x be less than *or equal* to those of y, instead of strictly less than.

C. Specification of Branch-and-Bound Algorithm

The preceding definitions allow us to explicitly characterize a branch-and-bound algorithm by

1. Its representation as a discrete decision process, Y denoted by the triple of (Σ, S, f).
2. Its control information, which is represented by the lower-bound function, g, the dominance conditions, D, and the equivalence conditions, EQ.

To complete the specification of the algorithm, we require one additional decision parameter: *how to select the next problem $Y(x)$, which we will solve*, or equivalently, *which node in the branching structure to expand.* We will define a *search function*, s, which allows us to select a node from the currently unexpanded nodes for expansion. In this chapter, as in Ibaraki (1978), we consider only *best bound search*, where we select the node with the minimum $g(x)$ value for expansion. Thus our branch-and-bound algorithm, A, is explicitly specified by

$$A = (Y, g, D, EQ, s),$$

and procedurally is given by the following steps:

Step-1. **Initialize**
Let $A \leftarrow \{\epsilon\}$, $N \leftarrow \{\ \}$ $z \leftarrow \infty$ where $\{\epsilon\}$ denotes the set of null strings (solutions of length zero).

Step 2. **Search**
If $A = \varnothing$, exit; else $x \leftarrow s(A)$ goto Step-3.

Step-3 **Test for feasibility**
If $x \in S$, let $z \leftarrow min(z, f(x))$. Goto Step-4

Step-4 **Test through the lower bound**
If $g(x) > z$ goto Step-8; else goto Step-5.

Step-5 **Test for relative dominance**
If $y.D.x$ for some $y(\neq x) \in N$; goto Step-8; else goto Step-6

Step-6 **Test for equivalence**
If $y.EQ.x$ for some $y \in N$ which has already been tested; goto Step-8; else goto Step-7

Step-7 **Decompose active node**
$A \leftarrow A \cup \{xa | a \in \Sigma\} - \{x\}$
$N \leftarrow N \cup \{xa | a \in \Sigma\}$; return to Step-2.

Step-8 **Terminate**
$A \leftarrow A - \{x\}$ and return to Step-2.

D. Relative Efficiency of Branch-and-Bound Algorithms

One measure of efficiency is the number of nodes of the branching tree that are expanded. Ibaraki (1978) has proved that the number of nodes expanded can be linked to the strength of the control knowledge, and proves that if one branch-and-bound algorithm has a better lower-bounding function, dominance and equivalence rules, then it will expand fewer nodes.

Let us briefly discuss the theoretical results providing the basis for the improved efficiency of branch-and-bound algorithms. Let $F = \{x \in \Sigma^*: g(x) \leq f(x^*)\}$ be the set of solutions that cannot be terminated by the lower-bound test. Then, the set L_A, defined by $L_A = F \cap \Sigma_D^*$, contains all the partial solutions, which can be terminated only by an equivalence relation. Recall that, by definition, no node in Σ_D^* can be terminated by a dominance rule.

Theorem 1

Let $A = (Y, g, D, EQ, s)$ be a branch-and-bound algorithm. The algorithm terminates after decomposing exactly $|L_A/EQ|$ nodes, provided that $|L_A/EQ| < \infty$, where L_A/EQ denotes the set equivalent classes of solutions induced by the equivalence conditions EQ.

For the proof of this theorem see Ibaraki (1978) as a result of Theorem 1, a natural measure of the efficiency of a branch-and-bound procedure is the number of the resulting equivalence classes under EQ. Furthermore, the following theorem (Ibaraki, 1978) allows a direct comparison of the efficiencies of two distinct branch-and-bound algorithms:

Theorem 2

Assume that the following two branch-and-bound algorithms

$$A = (Y, g, D, EQ, s) \quad \text{and} \quad A' = (Y, g', D', EQ', s')$$

terminate and satisfy the following conditions:

$g(x) \leq g'(x)$ for $x \in \Sigma^*$
$x.D.y \Rightarrow x.D'.y$ for $x, y \in \Sigma^*$
$x.EQ.y \Rightarrow x.EQ'.y$ for $x, y \in \Sigma^*$
Then

$$T_A' = |L_A/EQ'| < |L_A/EQ| = T_A,$$

i.e., algorithm A' is more efficient than algorithm A.

The importance of Theorem 2 is that it provides a *theoretical basis* for improving the control knowledge we have available to solve a given problem class. If we can derive new dominance and/or equivalence rules —or in the single optimal solution case, the enhanced dominance rules—we will be able to reduce the number of nodes enumerated. The focus of this chapter is the description of a methodology for achieving this goal, so that the *computer itself*, can carry out the process of acquiring new control information. The acquisition is based on the analysis of problem-solving activity, in the light of the formal specifications laid out for

dominance and equivalence rules, and the generalization of this "experience" to apply it to new problems within the same problem class.

E. Branching as State Updating

Branching from one partial solution to another involves more than just concatenating an alphabet element to a string; it also changes the underlying *state* of the problem. In our flowshop example the state of the partial schedule is conveniently and parsimoniously represented by the start- and end-times of the last operation on each unit. If the problem involved sequence-dependent equipment cleaning, we would have to include information about the identity of the batches as well. The choice of state variables for a problem is critical in determining the ease with which additional state properties and new state properties are calculated. For example, we could record just the list of batches scheduled in a given state and each time we wanted to calculate a new start- or end-time recalculate it from the list of batches. This would be computationally expensive, but our state information would be less complex. Thus, what we choose to record in a state addresses the issue of time/space tradeoff explicitly. In general, the more state variables we have the easier it will be to update each one during branching, but the more we will have to compute and store.

In addition to having to assign state variables to the strings of the DDP, we also have to assign properties to the alphabet symbols. In our flowshop example, the alphabet symbols can be interpreted as batches to be executed with a series of processing times. Thus, if we use the notation, $\Theta(x)$, to denote the state of partial solution, x, then

$$Y(x) \rightarrow Y(xa_i) \qquad \Theta(xa_i) = H[\Theta(x), I(a_i)],$$

where I returns the interpretation of the alphabet symbol and H is a vector of functions h_j; $j = 1, 2, \ldots, k$, which compute the properties, p_j; $j = 1, 2, \ldots, k$, characterizing $\Theta(xa_i)$:

$$p_j(xa_i) = h_j[\Theta(x), I(a_i)].$$

The state updating functions combine information about the constraints on the state variables with the objective function minimization. The feasibility predicate forces the state variables to obey certain constraints, such as the nonoverlap of batches, forcing the start-times of successive operations to be greater than the end-times of the previous operation. The constraints do not force the start-times to be equal to the previous

end-times; this comes as the effect of the objective function minimization. Consequently, in addition to the update functions, we also need to represent the underlying set of constraints:

Update functions:

$$s_{i,\sigma_{j+1}} = \max(e_{i,\sigma_j}, e_{i-1,\sigma_{j+1}}),$$
$$e_{i,\sigma_{j+1}} = s_{i,\sigma_{j+1}} + p_{ij}.$$

Constraints:

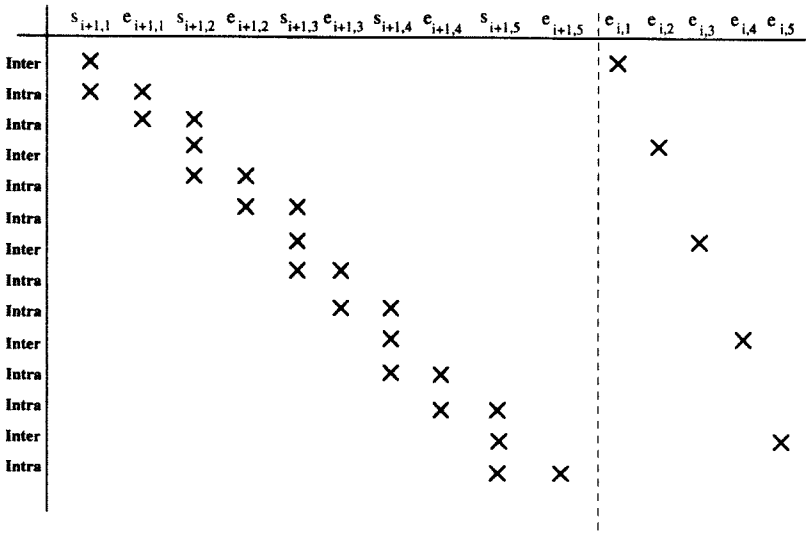
$$s_{i,\sigma_{j+1}} \geq e_{i,\sigma_j}, \tag{3}$$

$$s_{i,\sigma_{j+1}} \geq e_{i-1,\sigma_{j+1}}, \tag{4}$$

$$e_i, \sigma_{j+1} = s_{i,\sigma_{j+1}} + p_{ij}. \tag{5}$$

Figure 1 gives the structure of the constraint set. We can now classify these constraints into two classes. The first class of constraints includes the *intersituational constraints*, which have variables of the next state as their potential output variables, and the variables of the current state as their input variables. We can see that these correspond to constraints (3). We then have the *intrasituational constraints*, which relate variables of the next state together and correspond to constraints (4) and (5). We will similarly define those variables that appear in intersituational constraints to be *intersituational variables*, and those that appear in intrasituational constraints to be *intrasituational*. Note that these classes are not mutually exclusive. Thus, we will refine our classes to reflect whether the variable in a specific constraint is a potential output of that constraint or is an input to the constraint. In essence this input/output distinction separates the variables of the current state from those of the next state. In the flowshop example, the end-times of the current state are the input intersituational variables and the start-times of the next state are the output intersituational variables. The start-times also belong to the class of input intrasituational variables, and the end-times of the next state are the output intrasituational variables.

Our ability to make these distinctions rests on the fact that we know the direction that the branching generation imposes on the updating of the variables. If we were not solving the problem in such a way that all the variables are explicitly determined by the branching, then these distinctions would not be so clear. For example, if some variable values were the result of solving an auxiliary linear program that involved these constraints, we could not classify the variables this way.



Inter Intersituational Constraints

Intra Intrasituational Constraints

s Output Intersituational Variables, Input Intrasituational Variables

$e_{i+1,j}$ Output Intrasituational Variables

$e_{i,j}$ Input Intersituational Variables

FIG. 1.

The purpose of defining these classes of constraints and variables is that this will enable us to conveniently express general notions of equivalence and dominance, without explicit reference to the flowshop domain, but at a more abstract level.

F. FLOWSHOP LOWER-BOUNDING SCHEME

To define the lower-bound function for flowshop scheduling, we will use the lower-bounding schemes proposed in Lagweg et al. (1978). The lower-bound schemes are organized into a hierarchy that reflects the *strength* of the lower-bounding scheme.

The *strength* of a lower-bounding scheme is a binary relation, over two lower-bounding schemes, which indicates that if $\Omega' \rightarrow_D \Omega$, then the lower-bounding scheme Ω' will always yield an equal or lower value for

any σ, and hence will always lead to the expansion of fewer nodes, by the theorem of Section II, D.

We will use the simplest of lower-bounding schemes proposed in Lagweg et al. (1978), where all but one of the flowshop units are relaxed to be nonbottleneck machines, A nonbottleneck machine is a machine in one which the capacity constraint has been relaxed; i.e., it can process all the jobs simultaneously. Thus, to compute the lower bound, assuming that the bottleneck machine is u, and that we have completed the partial schedule σ, we need to schedule a single machine knowing that each unscheduled job, $i \in \phi$ will be released after a time $q_{i.u}$, representing the time it spends on the nonbottleneck machines up to u processed on u for a time p_{iu}, and then completed after a further period $q_{iu.}$, representing the time it spends on the nonbottleneck machines after u. We can calculate $q_{i.u}$ by adding the sum of the processing times of i on machines up to u to the partially completed schedule end-time where $q_{i.u} \equiv$ release time before reaching machine u and $q_{iu.} \equiv$ time spent on machines after u,

$$q_{i.u} = \max_{l} \left\{ C(\sigma, l) + \sum_{k=l}^{u-1} p_{ik} | l = 1 \ldots u \right\};$$

$q_{iu.}$ is simply the sum of the remaining processing time of i on the machines following u.

$$q_{iu} = \sum_{k=u+1}^{m} p_{ik}.$$

At this point the lower-bounding scheme consists of solving a single machine scheduling problem where for each job i, the release time is $q_{i.u}$, the due date is $-q_{iu.}$, and the processing time is p_{iu}. This nonbottleneck scheme can be further simplified in two steps. First, we can assume that $q_{i\alpha\beta} = \min_{i \in \phi}\{q_{\alpha\beta}\}$ for $\alpha\beta = .uu.$, avoiding the need to consider release times of $q_{i.u}$, and due dates of $-q_{iu.}$, which turns an NP-complete problem, into one solvable in polynomial time. If only one of these were to be relaxed, the schedule can still be found in polynomial time by Jackson's rule (Jackson, 1955). Second, we can avoid the computation of $q_{i.u}$ completely, by assuming that the maximum is obtained at $l = u$ for all values of i.

$$q_{i.u} = C(\sigma, u) \; \forall i.$$

Thus, our final lower-bound scheme

$$g(\sigma) = \max_{u} \left\{ C(\sigma, u) + \sum_{j \in \phi} p_{ju} + q_{u.} \right\},$$

where $q_{u.} = \min_{j}\{q_{ju.}\}$.

The reason we call this a scheme is because it is still parameterized by, u, the machine that we choose to retain as our bottleneck. The simplest choice of u is $u = m$, for which we get

$$g(\sigma) = C(\sigma, m) + \sum_{j \in \phi} p_{jm}.$$

The computational complexity of the lower bound is presented in Lagweg et al. (1978).

1. At the root node, we can calculate q_{iu}. for all (i, u) in $O(mn)$ steps, for $u = m$ we avoid this calculation.
2. At each node during the branching, we calculate the lower bound, by taking the minimum of the remaining unscheduled batches, q_{iu}.. This takes $o(|\phi|)$ for each machine or $O(m|\phi|)$ in all. The final calculation is then of order 1, provided we update and store $\Sigma_{j \in \phi} p_{ju}$ at each node. For $u = m$, the calculation is $O(1)$.

III. The Use of Problem-Solving Experience in Synthesizing New Control Knowledge

The purpose of this section is to illustrate the role that the problem-solving experience plays in improving problem-solving performance, via learning, and examine the question; "*exactly what is it that we are trying to learn*?"

A. An Instance of a Flowshop Scheduling Problem

To make the ideas of this section more concrete we will use a specific example of a flowshop problem. The problem is a small one, only five batches will be considered, since this enables us to examine the enumeration tree by hand.

Consider the flowshop in Fig. 2. The unit operations being performed are mixing, reaction, distillation, cooling, and filtration. We have made the following assumptions about the way the flowshop operates:

1. Processing times will be similar for each product type on a given processing step, reflecting the fact that technologically identical operations have the same, or similar, equations governing their behavior, with only parametric variation between products.

2. Processing times have some proportionality to batch size. We have chosen to allow three batch sizes (500, 1000, 1500 mass units). The reactor

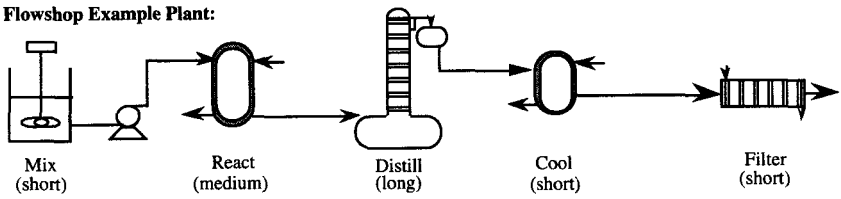


FIG. 2. Example configuration of unit operations in flowshop production.

processing time is the least sensitive to batch size, because the reaction time is assumed proportional to concentration and the increased time is due only to a constant pump rate assumption. The distillation processing time is the most sensitive, due to the assumption of constant boilup rate. The filtration period is roughly proportional to the size, because we have assumed a constant filter cake size and divided the larger batches to achieve this size.

3. The processing parameters have been chosen so that the distillation step is, in general, the longest step. In fact, processing periods for each operation have been roughly ordered as follows: D_1 (distillation) > R_1 (reaction) > F_1(filtration) > C_1(cooling) > M_1(mixing). Obviously, the size considerations may alter this ranking for batches of the same type but of different sizes.

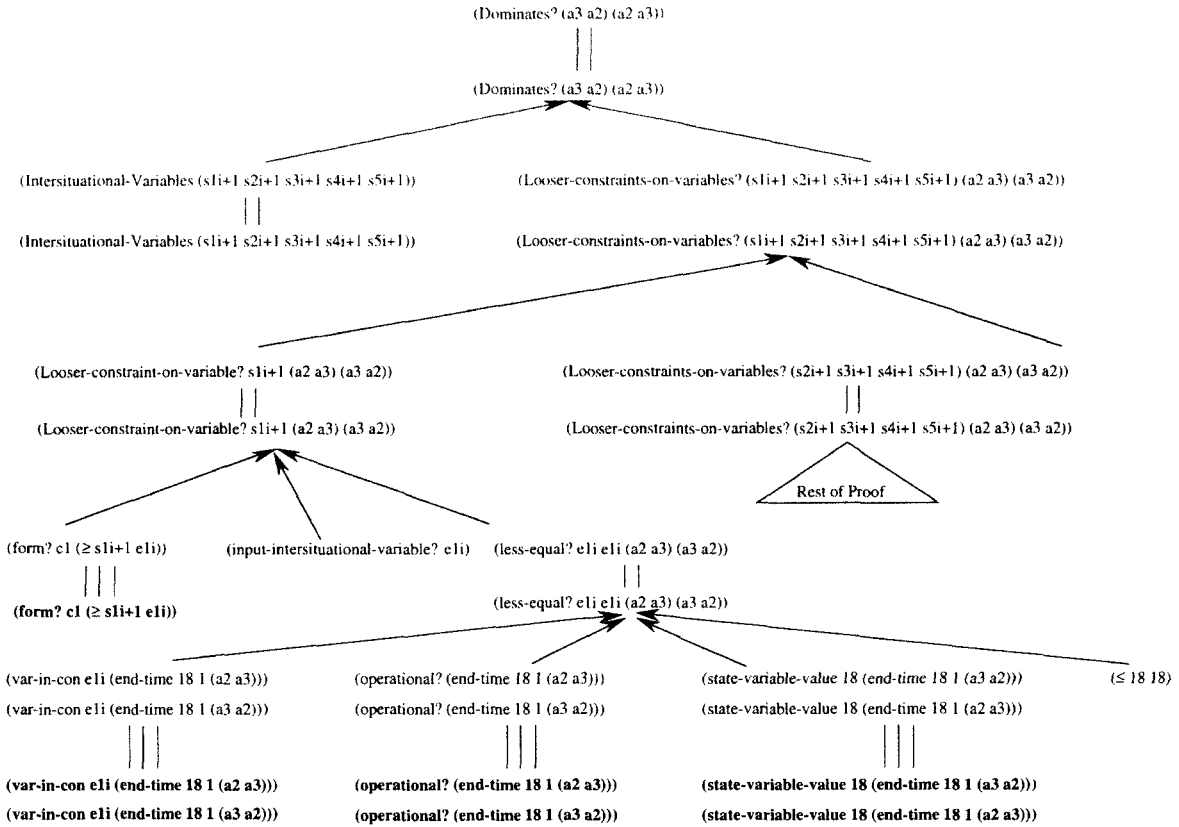
The example problem was generated by picking the type and size at random, while ensuring that no size and type combination was repeated. This last requirement was imposed to avoid making the equivalence condition appear to perform better due to spurious equivalences.

In particular, consider the batches shown in Table I. The first two columns of the table indicate the product type and batch size. The next column gives the processing times for each operation. The fourth column gives a unique identifier to each batch composed of a prefix "B-P," the

TABLE I

PRODUCTS IN THE SCHEDULING FLOWSHOP

Product	Size	Processing times M_1	R_1	D_1	C_1	F_1	Code	Alphabet symbol
2	500	2	17	24	4	10	"B-P-2-0"	a_1
1	1000	4	20	26	8	12	"B-P-1-1"	a_2
3	1500	5	26	36	15	19	"B-P-3-2"	a_3
3	500	2	24	12	5	6	"B-P-3-3"	a_4
1	1500	6	21	40	11	19	"B-P-1-1"	a_5



Facts are in bold.

FIG. 10. Proof of simple dominance condition.

product type and a counter. The fifth column lists the alphabet symbol associated with each batch. The total number of feasible schedules is 5! or 120. The total number of nodes in the tree is 326 (including the root node).

Utilizing only the simple lower bounding scheme with the bottleneck machine fixed to the last machine, we generate the branching tree given in Fig. 3, where much of the detail has been suppressed. The boldface numbers indicate the size of the subtrees (i.e., the number of nodes in the subtree) beneath the node.

We can now pose the following question:

> Do any of the nodes, which we have enumerated, satisfy the necessary and sufficient conditions for dominance or equivalence?

If we could answer the question affirmatively, we might be able to extract out the features of the nodes that cause them to participate in the relationship, and use these features to identify similar nodes in future problem solving activities. *This is the essence of the goal of learning.*

B. Definition and Analysis of Problem-Solving Experience

For branch-and-bound algorithms the notion of "problem-solving experience" is defined in term of the branching structure generated during the solution, and the various relationships between the nodes of the branching structure that are engendered by existing control knowledge.

1. Identification of Nodes for Dominance and Equivalence Relationships

To identify pairs of nodes that might be capable of being proved to participate in dominance or equivalence relationships, we will employ a strategy of *successive filtering* of the branching structure. We will concentrate on identifying equivalent nodes (for purposes of illustration), although a similar methodology applies to the case of nodes that participate in a dominance relationship.

The necessary and sufficient conditions for equivalence are repeated here for convenience.

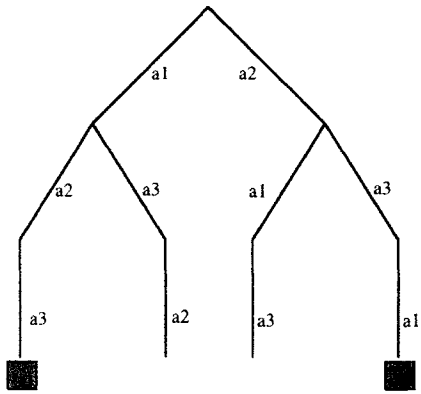
Condition-a:

$$x.EQ.y \Rightarrow \forall u \in \Sigma^* \begin{cases} xu \in S \leftrightarrow yu \in S, \\ f(xu) = f(yu) \end{cases}$$

Condition-b:

$$x.EQ.y \Rightarrow \forall u, w \in \Sigma^* \begin{cases} g(xu) = g(yu), \\ xu.EQ.w \leftrightarrow yu.EQ.w, \\ xu.D.w \leftrightarrow yu.D.w, \\ w.D.xu \leftrightarrow w.D.yu \end{cases}.$$

First, let us concentrate on the meaning of *Condition-a*. Essentially, we are required to ensure that all the solutions, which lie below nodes x and y, have the same objective function value, and that x and y engender the same set of solutions. A problem involving a finite set of objects, each of which must be included in the final solution, is termed a *finite-set problem*. For finite-set problems, we can be confident that $x\omega \in S \leftrightarrow y\omega \in S$ if x and y contain the same set of alphabet symbols. In fact, if x and y do not contain the same set of alphabet symbols, they cannot be equivalent, the proof of which is trivial. Thus our first step in identifying candidates is to find two solutions $x', y' \in S$ that have equal objective function values. We then try to identify their ancestors x, y, such that the set of alphabetic symbols contained in x and y are equal. If in the process of finding this "common stem" we find that the stem is the same for both x', y', then we have the situation depicted in Fig. 4, and $x = y$. This is of no value; a solution is trivially equivalent to itself. We term this *ancestral-equality*. So far, all that has been established for x and y is that there exist two



Two solutions can have different tails which lead to solutions with equal objective function values but which cannot produce equivalence relations because they have the same ancestor.

FIG. 4. Schematic depicting ancestral equality.

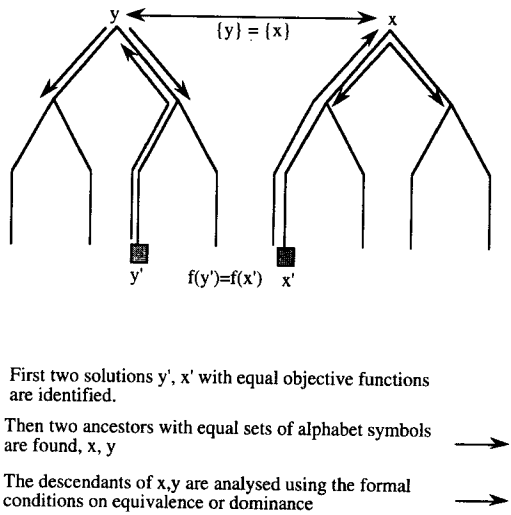


FIG. 5. Pattern of reasoning required to identify examples of equivalence or dominance by the syntactic criterion.

feasible solutions, x' and y', for which $f(x') = f(y')$ and x', y' are the descendants of x, y. We must now reverse the direction of reasoning, and verify that for all the feasible children of x and y that $f(x\omega) = f(y\omega)$, as in Fig. 5.

If we are successful, then we have verified that for the conditions prevailing in the example, the partial solutions, x and y would indeed be equivalent, as far as *Condition-a* is concerned. We now move to *Condition-b*, which ensures that the equivalent node will play an equivalent role in the enumeration as the one that was eliminated. Thus, as we examine the children of x, y, regardless of whether they are members of the feasible set, we would verify that their lower-bound values were equal, and that if we had any existing dominance, or equivalence conditions that the equivalent descendant of x, i.e., xu, participates in the same relationships as does yu.

To make the above discussion more concrete, consider the example branching structure of Fig. 3. In this structure, we have identified an x' and y', which have the same objective function values, and their ancestors, (x, y) which are characterized by the same set of symbols, (i.e., batches). Furthermore, we can see that the children of x, y do indeed satisfy the requirements of *Condition-a* and *Condition-b* and hence, (x, y) would be considered as candidates to develop a new equivalence relationship. If we examine the partial schedules (x, y) as depicted in Fig. 6, our knowledge

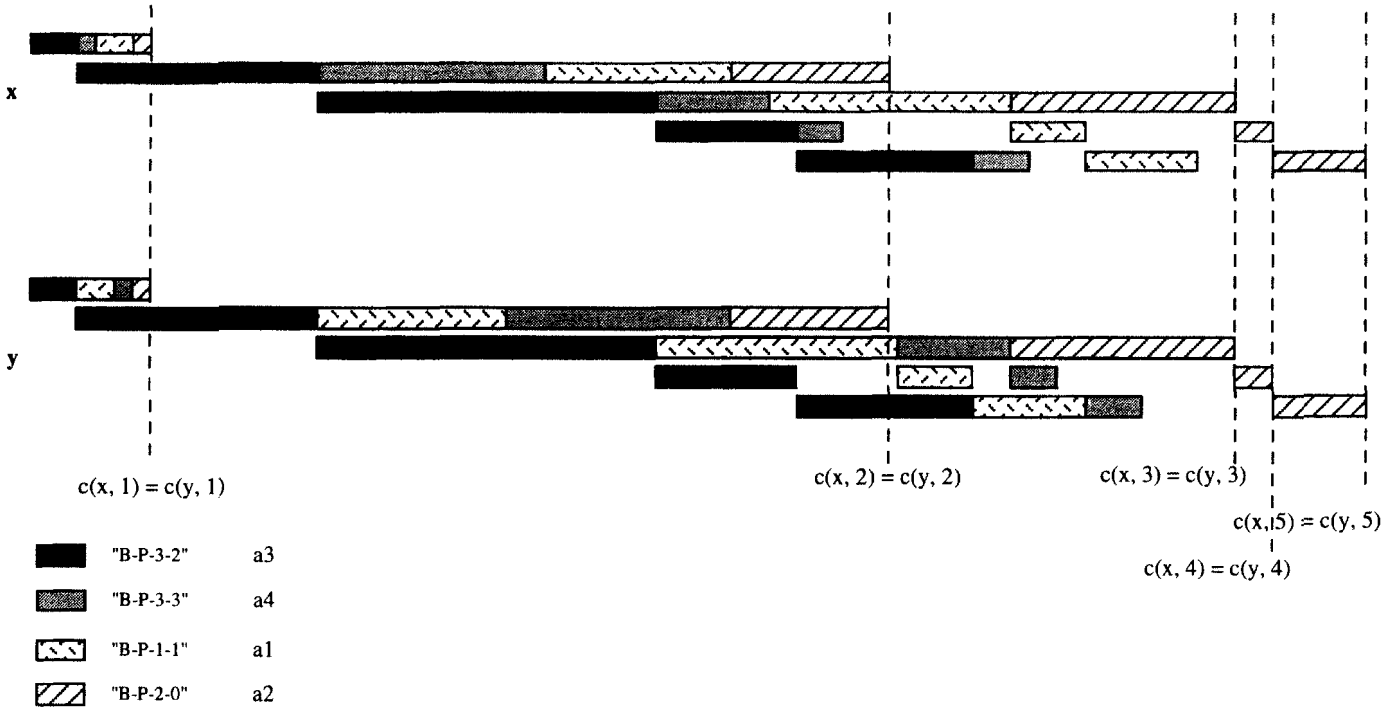


FIG. 6. Potentially equivalent schedules x and y.

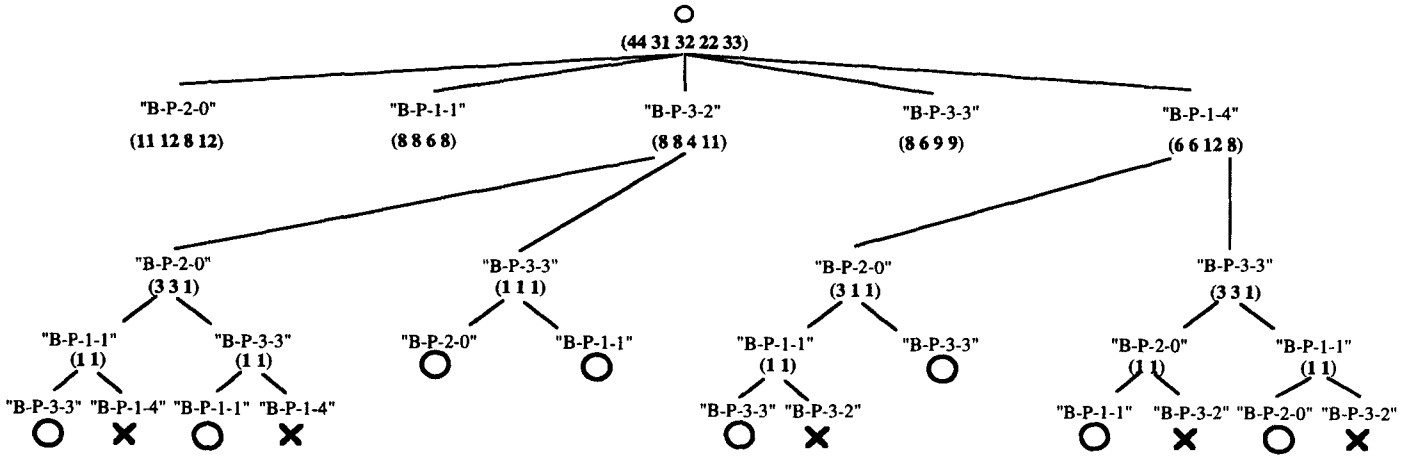


FIG. 7. Enumeration modified by basic equivalence condition.

of flowshop scheduling should enable us to see that indeed (x, y) are equivalent because the completion times on each machine are equal, i.e., $C(x,i) = C(y,i)$ for every machine, i, and hence for each arrangement of the remaining batches, the completion times will be equal, i.e., $C(xu, m) = C(yu, m)$ for all feasible completions, u.

The last step in the preceding argument, the use of our knowledge about flowshop scheduling, turns what had been a mainly *syntactic criterion* over the tree structure of the example, into a criterion based on state variables of (x, y). The state variable values, the completion times of the various flowshop machines, are accessible *before* the subtrees beneath x and y have been generated. Indeed, they determine the relationships between the respective elements of the subtrees (xu, yu). If we can formalize the process of showing that the pair (x, y) identified with our syntactic criterion, satisfies the conditions for equivalence or dominance, we will in the process have generated a "new" equivalence rule.

Using this equivalence rule in the example, we generate the branching structure of Fig. 7, where we can see the number of nodes generated has been reduced from 218 to 211. Our syntactic criterion, combined with ensuring that all the lower bounds are equal, will identify five pairs of potentially equivalent solutions in the example. All five of these pairs can be proved to be equivalent on the basis of a general theory of equivalence. Of these pairs, one would have enabled the elimination of a partial schedule of length three, which would have been expanded to give three more solutions, and the other four pairs each eliminate one solution for a total of seven.

The extension of this scheme to dominance conditions requires us to deal with the problem of *incomplete tree structure*. When certain objective and lower-bound values are unavailable, due to nodes being terminated before full expansion, we cannot be certain that the (x, y) pair satisfies the criterion for dominance. To solve this problem, we have to extend and relax the syntactic criteria. Such extensions are beyond the scope of this chapter, and for more details, the reader is referred to Realff, (1992).

C. Logical Analysis of Problem-Solving Experience

The analysis of problem-solving experience that has taken place so far has been based on finding subproblems within an existing branching structure that, when solved, will produce subtrees that satisfy the definitions of dominance and equivalence. As we have noted, this is insufficient for generating new dominance and equivalence conditions because we

want to avoid the subproblem solution, and not use it as the means of proof.

Thus, the next step in the problem-solving analysis is to use information about the *domain of the problem*, in this case flowshop scheduling, and information about dominance and equivalence conditions that is pertinent to the overall *problem formulation*, in this case as a state space, to convert the experience into a form that can be used in the future problem-solving activity.

We can now identify constraints on the "logical analysis" which must be satisfied for it to produce the desired dominance and equivalence conditions.

1. *Validity.* The reasoning involved in this phase must be logically justifiable; that is, the final conclusion should follow deductively from the facts of the example and from the theory of the domain. If we do not ensure this, we may be able to derive conditions on the two solutions that do not respect either the structure of the domain or the example. The use of these conditions could then invalidate the optimum-seeking behavior of the branch-and-bound algorithm.

2. *Sufficiency.* Given the example, the theory we employ to deduce the conditions must guarantee that the conditions on equivalence or dominance will be satisfied, since the example itself is simply an instance of the particular problem structure, and may not capture all the possible variations. The sufficient theory will thus end up being *specialized* by the example, but since we have asserted the need for validity of this process, it will still be guaranteed to satisfy the abstract necessary and sufficient conditions on dominance and equivalence.

D. Sufficient Theories for State-Space Formulation

Before showing how the logical analysis will be carried out, it is useful to describe the sufficient theory we will be using for the specific flowshop example. This theory is not restricted to flowshop scheduling, but applies to many state-space problems.

We have characterized our equations and variables as inter- and intra-situational. In transitioning from one state to another, we can view the intersituational equations as constraining the values of the next state's variables. Viewed as constraints, we can define the "looseness" or "tightness" of the constraints that values of a state's variables place on the next state. Thus, for example, since the start-times of the units in the next state are constrained to be larger than the end-times of the same unit in the

current state, the constraint will be "tighter" if the end-times are greater in one state than in another. The question then arises, what about the intrasituational constraints on the variables, such as the end-times of the previous unit in the same state?

By definition, the intrasituational variables must eventually be expressed in terms of the intersituational variables. It can be shown (Realff, 1992) that as long as we guarantee that the intersituational variables are at least as loosely constrained in x and y, the intrasituational variables will be, also. The final issue concerns intersituational variables that are constrained via an equality constraint.

In these cases there is no well defined notion of a "looser constraint," the choice is then either to force those variables to be equal in x and y, or to find some path from their value to a constraint on another inter- or intrasituational variable and thus be able to show that their values in x, y should obey some ordering based on these other constraints. This topic is the subject of current research, but is not limiting in the flowshop example, since no such constraints exist. Lastly, it is not enough to assert conditions on the state variables in x and y, since we have made no reference to the discrete space of alternatives that the two solutions admit. Our definition of equivalence and dominance constrains us to have the same set of possible completions. For equivalence relationships the previous statement requires that the partial solutions, x and y, contain the same set of alphabet symbols, and for dominance relations the symbols of x have to be equal to, or a subset of those of y. Thus our sufficient theory can be informally stated as follows:

> If every variable that is the output of an intersituational constraint is at least as loosely constrained in a state x and state y, and x has a subset of, or equal set of, the alphabet symbols of y, then x dominates y.

1. General Qualities of Sufficient Theories

In general, we would like our sufficient theory to have the following qualities:

1. *Simplicity.* The complexity of the sufficiency conditions will tend to translate into complex dominance and equivalence conditions. This complexity can take the form of either a large number of predicates to guarantee the conclusion or complex predicates.

2. *Efficacy.* We must balance making the sufficient theories simple versus making them effective. If, for example, the relative ordering of the intrasituational variables or some function of the state variables, such as the difference between the end-times of successive units, is the driving

force behind the problem structure, then the sufficient theory will be unable to explain why the observed relationship between x and y should hold in general.

3. *Generality.* The more abstract the general theory, the wider the class of problems to which it will apply, and hence the higher the likelihood of being able to transfer knowledge acquired in one class of problems to another, related class. However, we have to construct problem domain theories that connect the information from the specific problem-solving experience to the general theories. The more abstract the theory, the more effort required to do this. On the other hand, if the theory is very specific, then it will probably admit few generalizations, and apply in few problem domains.

4. *Modularity.* Since we would like to use the sufficient theory in a variety of contexts and problems, we need a theory that was easy to extend and modify depending on the context. In our state-space formulation the sufficient theory is couched in terms of constraints on variables. This theory gives us the opportunity to modularize its representation, partitioning the information necessary to prove the looseness of one type of constraint from that required to prove the looseness of a different constraint type. The ability to achieve modularity is a function not only of the theory but also of the representation, which should have sufficient granularity to support the natural partitioning of the components of the theory.

The next section will focus on the representation necessary to express this sufficient theory to the computer, so that it can automatically carry out the reasoning associated with analyzing the examples selected by the syntactic criteria presented in this section. Section V will describe the learning methodology, which, using the representation of Section IV, will generate the new dominance and equivalence conditions.

IV. Representation

This section details the different aspects of the representation we have adopted to describe the problem solutions and the new control knowledge generated by the learning mechanism. Throughout the section we will continue to use the flowshop scheduling problem as an illustration. The section starts by discussing the motives for selecting the horn clause form of first-order predicate calculus, and then proceeds to show how the representation supports both the synthesis of problem solutions and their analysis. The section concludes with a description of how the sufficient

theory for state-space models can be expressed in a general way using the proposed representation. The "horn clause form" is described in the paragraphs that follow.

It is beyond the scope of this chapter to explain the syntax and semantics of first-order predicate logic, and the reader is referred to Lloyd's text (1987) for a general introduction. However, it is useful to provide some details on the horn clause form.

A logical clause is a disjunction of terms, and a horn clause is one where, at most, one term is positive. In propositional logic, where only literals are allowed horn clauses have the following form:

$$\neg p \vee \neg q \vee \neg r \vee s.$$

This statement is logically equivalent to

$$p \wedge q \wedge r \Rightarrow s.$$

In predicate calculus, where we are allowed function and predicate symbols that take one or more arguments, the form is

$$\neg p(x) \vee \neg q(y) \vee \neg r(x, y) \vee s(x, y),$$

where implicitly, the x, y variables are allowed to range over the entire domain; i.e., they are universally quantified.

We have chosen this representation for a variety of reasons:

1. Theoretically it has been shown (Thayse, 1988) that the DDP formalism is closely related to a simpler form of horn clause logic, i.e., the propositional calculus. This would suggest that we could use the horn clause form to express some of the types of knowledge we are required to manipulate in combinatorial optimization problems. The explicit inclusion of state information into the representation, necessitates the shift from the simpler propositional form, to the first-order form, since we wish to parsimoniously represent properties that can be true, or take different values, in different states. By limiting the form to horn clauses, we are striving to retain the maximum simplicity of representation, whilst admitting the necessary expressive power.

2. First-order predicate calculus admits proof techniques that can be shown to be *sound* and *complete* (Lloyd, 1987). The soundness of the proof technique is important because it ensures that our methodology will not deduce results that are invalid. We are less concerned with completeness, because in most cases, although the proof technique will be complete, the theory of dominance or equivalence we have available will be incomplete for most problems. Restricting the first-order logic to be of horn clause form, enables the employment of SLD resolution, a simpler

proof technique than full resolution (Robinson, 1965). [*Note*: SLD stands for linear resolution with selection function for definite clauses (Lloyd, 1987)].

3. First-order horn clause logic is the representation that has been adopted by workers in the field of *explanation-based learning* (Minton et al., 1990). Hence, using this representation allows us take advantage of the results and algorithms developed in that field to carry out the machine learning task.

A. Representation for Problem Solving

The representation introduced in the previous subsection must now be utilized to express the information derived from the problem-solving experience, and required to derive the new control knowledge. Our first step will be to define the types of predicates we require to manipulate the properties of the branching structure and the theory that is needed to turn those properties into useful dominance and equivalence conditions.

In Section II, we presented the computational model involved in branching from a node, σ, to a node σa_i. In this model, it was necessary to *interpret* the alphabet symbol a_i, and ascribe it to a set of properties. In the same way, we have to interpret σ as a state of the flowshop, and for convenience, we assigned a set of state variables to σ that facilitated the calculation of the lower-bound value and any existing dominance or equivalence conditions. Thus, we must be able to manipulate the variable values associated with state and alphabet symbols. To do this, we can use the distinguishing feature of first-order predicates, i.e., the ability to parameterize over their arguments. We can use two place predicates, or binary predicates, where the first place introduces a variable to hold the value of the property and the second holds the element of the language, or the string of which we require the value. Thus, if we want to extract the lower bound of a state σ, we can use the predicate (*Lower-bound* ?g [σ]) to bind ?g to the value of the lower bound of σ. This idea extends easily to properties, which are indexed by more than just the state itself, for example, unit-completion-times, v, which are functions of both the state and a unit

$$(\text{unit} - \text{completion} - \text{time } ?\text{v k } [\,\sigma\,])$$

where k can range over the number of units in the flowshop.

We must now connect these various predicates to be able to derive state variables from other state variables. The process of transitioning from one

state to another, can be modeled by the deductive form of the horn clauses. This is a common strategy in artificial intelligence, and is formalized by *situational calculus* (Green, 1969).

However, we do not need the full complexity of situational calculus, which allows for multiple operators to be applied to a given state, because we have only one way of going from one state to another, i.e., branching. Branching has the effect of updating the state properties and the partial string, σ. We will "borrow" the notation of Prolog (Clocksin and Mellish, 1984) to indicate a list of items, where $[?a.?\sigma]$ parses the list into the head of the list, $?a$, and the rest of the list, $?\sigma$. Thus, our horn clause *implications* (*rules*) will have the form

$$\left.\begin{array}{l} p(?v_1,\ldots,?\sigma) \\ q(?v_2,\ldots,?\sigma) \\ r(?v_i,\ldots,?a) \\ (?i_1 = h_1(?v_1,\ldots,?v_n)) \\ (?\omega = h_\omega(?v_1,\ldots,?v_n,?i_1,\ldots,?i_m)) \end{array}\right\} \Rightarrow p(?\omega,\ldots,[?a.?\sigma])$$

Notice that the left-hand side of this rule contains two types of clauses. The first type is the variable values of the current state and those necessary to compute the new state, while the second, represented by " = " computes the value of the variable in the new state. This last clause enables the procedural information about how to compute the state variables to be attached to the reasoning. We must, however, be careful about how much of the computation we hide procedurally, and how much we make explicit in the rules. The level to which computation can be hidden will be a function of the theories we employ to try to obtain new dominance and equivalence conditions. If we do not hide the computation, we will be able to explicitly reason about it, and thus may find simplifications or redundancies in the computation that will lead to more computationally efficient procedures.

We can further subdivide the implications of the above form into two categories; *intersituational* and *intrasituational*:

1. The first type of implications involve clauses with $?\sigma$ on the left-hand side and $[?a.?\sigma]$ on the right-hand side, indicating a branching step.
2. The second type of implications involve clauses with the same state on the left- and right-hand sides of the implication, indicating the derivation of a state variable value from other variables in the state.

These implications types correspond closely to the definitions of state variable types we gave in Section II, E. Those variables that appear in the

clause of the right-hand side of an implication that is intersituational are intersituational variables; those variables that appear on the right-hand side of an implication that is intrasituational are intrasituational variables.

1. Horn Clauses in the Synthesis of Branching Structures

Let us now illustrate how the horn clause for of predicate logic can be used for the unfolding of the branching structure and the synthesis of new solutions.

In the flowshop example, when we branch, we have to calculate the start-times of the new batch to be scheduled. Here are the rules that can accomplish this:

Axiom-1: (*End-time* 0 ?j [ ])
Axiom-2: (*End-time* 0 0 ?σ)

a. Intersituational Implications (Rules).

Rule-1: (End-time ?e_1 1 ?σ) $\Rightarrow$ (Start-time ?e_1 1 [?a.?σ])

$$\textit{Rule-2:}\quad \left.\begin{array}{l}(?j = ?i - 1)\\ (\textit{End-time}\ ?e_\sigma\ ?i\ ?\sigma)\\ (\textit{End-time}\ ?e_j\ ?j\ [?a.?\sigma])\\ (\geq ?e_j\ ?e_\sigma)\end{array}\right\} \Rightarrow (\textit{Start-time}\ ?e_j\ ?i\ [?a.?\sigma])$$

$$\textit{Rule-3:}\quad \left.\begin{array}{l}(?j = ?i - 1)\\ (\textit{End-time}\ ?e_\sigma\ ?i\ ?\sigma)\\ (\textit{End-time}\ ?e_j\ ?j\ [?a.?\sigma])\\ (\geq ?e_\sigma\ ?e_j)\end{array}\right\} \Rightarrow (\textit{Start-time}\ ?e_\sigma\ ?i\ [?a.?\sigma])$$

There are several important points about these rules.

1. *Explicit conditioning.* We could have written the following rule:

$$\left.\begin{array}{l}(?j = ?i - 1)\\ (\textit{End-time}\ ?e_\sigma\ ?i\ ?\sigma)\\ (\textit{End-time}\ ?e_j\ ?j\ [?a.?\sigma])\\ (?s = \max(?e_j\ ?e_\sigma)\end{array}\right\} \Rightarrow (\textit{Start-time}\ ?s\ ?i\ [?a.?\sigma])$$

However, by doing so we hide, in the computation of the maximum, a potentially important symbolic constraint between the values of the end-times of the previous unit and the current unit.

2. *Bidirectionality.* During the branching step, we could use these rules to deduce the new value of the start-time, by reasoning from the antecedents to the conclusion, i.e., the facts in the state associated with $?\sigma$ are known such as the unit end-times. We can use these facts to establish the new facts in $[?a.?\sigma]$ by opportunistically applying our rules. However, actually carrying out the updating in this way would be extremely inefficient, and a purely procedural approach is sufficient. During the logical analysis of partial solutions we would start with facts about $[?a\ ?\sigma]$ and be able to deduce constraints on the earlier partial solution variable values.

2. *Horn Clauses in the Analysis of Branching Structure*

The analysis of the branching structure turns the preceding deduction process around. We have all the facts available to us at the end of the solution synthesis, i.e., at the end of solving a particular problem. Our task is to select and connect subsets of those facts to prove new results that are useful for deriving new control information. In essence, we have to *turn facts about solutions and partial solutions at lower levels in the tree, into constraints on the properties of states and alphabet interpretations higher in the tree.*

For example, if our goal were to show that the end-time of the schedule, x, was equal to a particular value, but we wanted to verify this not for x directly, but for some ancestor of x, then we could use the following implication, in addition to our conditions for start-time deduction (see Section IV, A, 1, a) to carry out the reasoning.

a. Intrasituational Implication.

$$\left.\begin{array}{ll} & (\textit{Start-time}\ ?s\ ?j\ [?a.?\sigma]) \\ \textit{Rule-4} & (\textit{Proc-time}\ ?p\ ?j\ ?a) \\ & (=?e(+?s\ ?p)) \end{array}\right\} \Rightarrow (\textit{End-time}?e\ ?j\ [?a.?\sigma])$$

Figure 8 shows the trace of the application of *Axiom*-1, *Axiom*-2, *Rule*-1, *Rule*-2, *Rule*-3, and *Rule*-4 in our specific scheduling problem. The trace consists of a repeated pattern of rule applications *Rule*-4 followed by either *Rule*-2 or *Rule*-3. At each step the intrasituational rule converts an end-time to a start-time, and then the start-time is matched to the

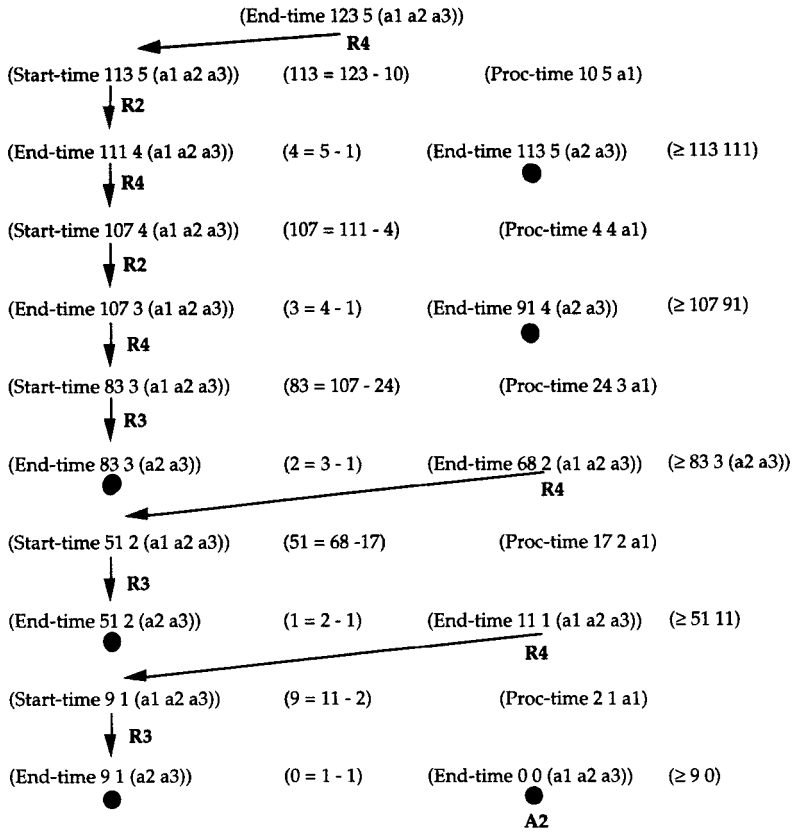


FIG. 8. Trace of deduction for proving the value of the end-time of batch a_1 on unit 5.

appropriate intersituational rule, to generate another end-time. This process is the reversal of the calculation of the state variables performed during the branching, where end-times were derived from start-times. The other rule antecedents fix the relationships between the relative sizes of end-times, and enable calculations to be performed. These conditions must be true of the example for the proof to be successful. We stopped the proof at the predicates

$$(\textit{End-time}\ 113\ 5[a_2a_3]), (\textit{End-time}\ 83\ 2[a_2a_3]),$$
$$(\textit{End-time}\ 51\ 2[a_2a_3]), (\textit{End-time}\ 9\ 1[a_2a_3])$$

Why did we not continue the deduction process beyond these particular predicates? Also, we continued to expand the end-time predicate,

$$(\text{end-time}\ 111\ 4[a_1a_2a_3])$$

which did not constrain the start-times. This lead to the derivation of a number of constraints between end-times that would appear to be unnecessary.

The first of these issues is known as *operationality* (Minton et al., 1990) and will be discussed in Section V. The second problem requires us to equip the deduction process with a means to know when deductions are irrelevant, or to change the way the conditions are expressed to avoid the problem altogether. In general, neither of these solutions can be accomplished in a domain independent way. However, the latter solution confines the domain dependency to the conditions themselves, rather than the deduction mechanism, and thus is preferred.

B. Representation for Problem Analysis

The focus of the representation so far, has been on giving the form of rules, which enable us to reason about the values of state variables. This, however, is only one part of the overall reasoning task. We must also represent the theoreies we are going to use to derive the new control knowledge.

The goal of the reasoning is to prove that two partial solutions are equivalent to one another, or that one dominates the other. To do this, we will start with conditions that contain

$$(equivalent\ ?x?y), (dominates\ ?x?y)$$

and terminate with predicates that are easy to evaluate within the branch-and-bound procedure.

The type of theories we will be using to prove dominance and/or equivalence of solutions will not be specific to the particular problem domain, but will rely on more general features of the problem formulation. Thus, for our flowshop example, we will not rely on the fact that we are dealing with processing times, end-times, or start-times, to formulate the general theory. The general theory will be in terms of sufficient statements about the underlying mathematical relationships, as described in Section III.

For example, consider trying to prove that one solution, x, will dominate another solution, y, if they both have scheduled the same batches, but the end-times of each machine are earlier in the partial schedule (partial solution), x, than in y. The general theory will not be couched in these terms, but more abstractly, in terms of the properties of binary operators,

such as max, min, +, −. Specifically, the implications

$$A \leq B \wedge C \leq D \Rightarrow A + C \leq B + D$$

$$A \leq B \wedge C \leq D \Rightarrow \max(A, C) \leq \max(B, D)$$

would enable us to deduce the requisite sufficient conditions for dominance–equivalence relationships. The tree of algebraic manipulations and implications, shown in Fig. 9, is such a case in point.

We can thus think of our predicates as falling into two classes: (1) the *formulation-specific*, and (2) the *problem-specific*. The implications fall into three classes: (1) those that interconnect the general concepts of the formulation, (2) those that connect the general concepts of the formulation to the specific details of the problem, and finally (3) those that enable reasoning about the specific details of the problem. We have already described the predicates necessary for reasoning within the flowshop problem; thus the rest of this section will focus on the general predicates and their interconnection with the specific problem details.

1. Predicates for Problem Analysis

To support the analysis of the solutions we will introduce several new predicate types. We have already described the basic state model in Section II, E. Explicitly manipulating parts of this model is an important component of the reasoning required to derive new control conditions. We will introduce the predicate, "form?," which will enable us to analyze the form of the state update rules, or constraints between variables. Thus, for our flowshop example we have two basic types of constraints:

$$(form?\ ?c\ (?d \geq ?a))$$
$$(form?\ ?c\ (?d = +?a?b))$$

These predicate structures can then be bound to the various instances of the constraints in the problem. For example, we would include among our facts declarations of the forms of the constraints, such as

$$(form?\ c_1\ (s_{i+11} \geq e_{i1}))$$
$$(form?\ c_2\ (s_{i+12} \geq e_{i+11}))$$

In addition to being able to identify the types of constraints, we must also be able to identify the types of variables. We introduce two predicates, "Intersituational?" and "Intrasituational?" to identify the variable types.

The definition of the abstract form of the constraints between states must now be linked to the actual values of the variables in the constraints

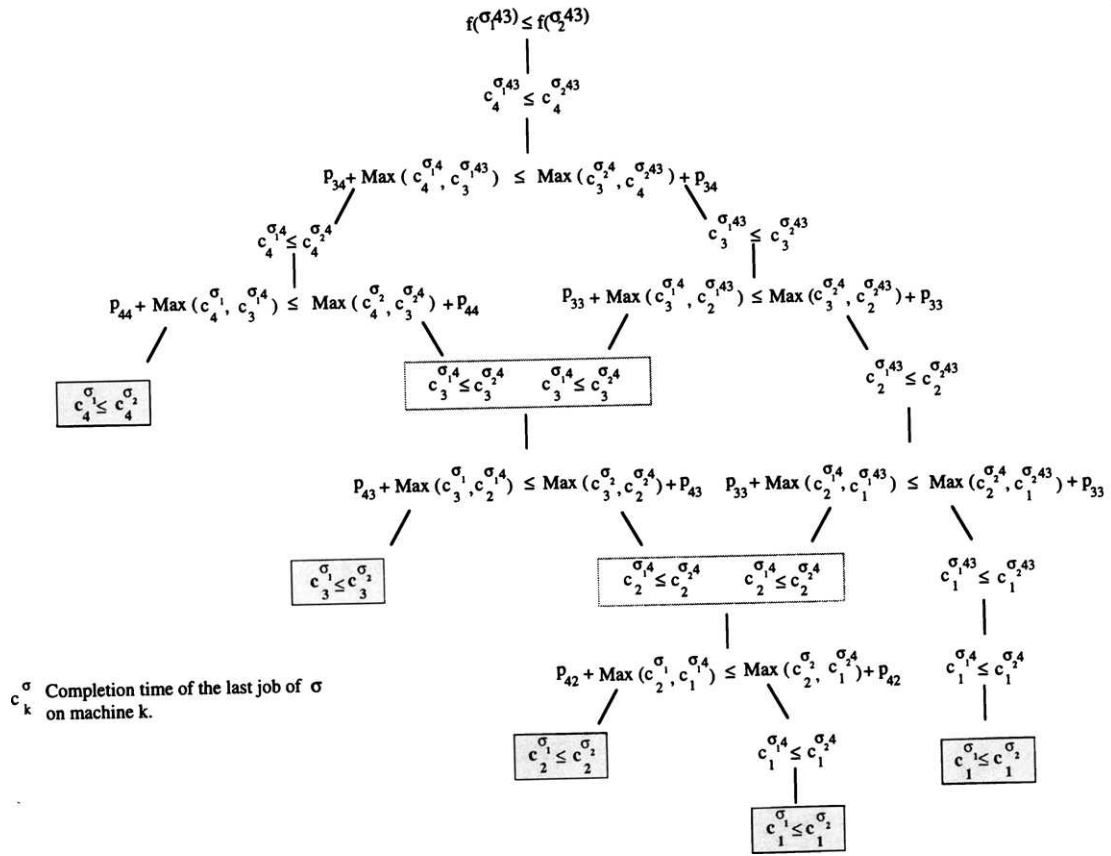


FIG. 9. Tree showing algebraic manipulations necessary to derive sufficient conditions for dominance or equivalence.

themselves, for different states. For this we use the predicates

(*variable – in – con*? ?*c* ?*var* ?*partial – solution*)
(*State – variable – value* ?*val* ?*var*).

These predicates could be created for the various states that are explored during the branch-and-bound algorithm, or an appropriate subset that has been identified during the example identification.

We then use a predicate for each variable type to be able to pattern match from the variable to its value:

(*State – variable – value* ?*val* (*end-time* ?*val* ?*unit* ?*state*))
(*State – variable – value* ?*val* (*start-time* ?*val* ?*unit* ?*state*)).

These predicates are clearly dependent on the specific problem. We will assume that they are available as basic facts, but we could continue to analyze them using the specific theory of the problem, to turn the start- and end-times of one state into constraints on processing times, and start- and end-times of other states.

We need to introduce two more types of predicates to support the sufficient theory used in the analysis. As we have stated in Section III, D, the sufficient theory rests on being able to prove that the constraints on one state are looser than those on another. The predicate that is used to express this is "looser – constraint – on – variable?," which takes the form:

(*looser – constraint – on – variable*? ?*var* ?*x* ?*y*)

where ?*x* and ?*y* will be bound to particular partial solutions and the variable will be one that occurs on the left-hand side of the constraint. To prove that variables are more loosely constrained, we need to compare variable values in different states. Thus, we introduce a new predicate, "*less-equal*" which implements the predicate, "*looser-constraint-on-variable*." This predicate has the form:

(*less-equal*? ?*a* ?*b* ?*x* ?*y*)

where ?*a* ?*b* will be variables in the constraints and ?*x* ?*y* will be the partial solutions.

2. *Expression of Sufficient Theory*

With the basic predicate types in place we can now define the various implications that will allow us to express the sufficient theory. There are two key steps that have to be made. The first is to take an intersituational variable and figure out what the constraint on the variable is, which of the

current state variables are involved in the constraint, and how these variables should be ordered for the constraint to be looser in x than in y. In our example there is a single constraint type, the greater than constraint, leading to the following rule:

$$\left.\begin{array}{l}(\textit{form}?\ ?c\ (?d \geq ?a)) \\ (\textit{Intersituational}?\ ?a) \\ (\textit{less-equal}?\ ?a\ ?a\ ?x\ ?y)\end{array}\right\} \Rightarrow (\textit{looser-constraint-on-variable}\ ?d\ ?x\ ?y)$$

For the second step we define predicates that analyze the relationships between variables, recognizing that we are interested only in comparing the actual numerical values of variables associated with solutions, $?x$ and $?y$, which are explicitly declared to be "operational." Thus, we create the following rule:

$$\left.\begin{array}{l}(\textit{Variable-in-con}?\ ?a\ ?v_1\ ?x) \\ (\textit{Variable-in-con}?\ ?b\ ?v_1\ ?y) \\ (\textit{Operational}\ ?v_1) \\ (\textit{Operational}\ ?v_2) \\ (\textit{State-variable-value}\ ?val_1\ ?v_1) \\ (\textit{State-variable-value}\ ?val_2\ ?v_2) \\ (>=?val_1\ ?val_2).\end{array}\right\} \Rightarrow (\textit{less-equal}?\ ?a\ ?b\ ?x\ ?y)$$

We can now use the preceding implications (rules) to help build the general sufficient theory. As we stated in Section III, D, we have to ensure that all the intersituational variables of the next state are more loosely constrained in x than in y. Thus, our top-level implication (rule) is simply

$$\left.\begin{array}{l}(\textit{Output-intersituational-variables}?\ ?v) \\ (\textit{looser-constraints-on-variables}?\ ?v\ ?x\ ?y)\end{array}\right\} \Rightarrow (\textit{Dominates}?\ ?x\ ?y)$$

Then we create a recursive definition of the second predicate to test each variable individually:

$$\left.\begin{array}{l}(\textit{looser-constraint-on-variable}?\ ?var\ ?x\ ?y) \\ (\textit{looser-constraints-on-variables}?\ ?rest\ ?x\ ?y)\end{array}\right\} \Rightarrow (\textit{looser-constraints-on-variables}?\ (?var\ ?rest)\ ?x\ ?y)$$

The specific value of the first predicate would have been entered into the system from the results of the analysis of the structure of the con-

TABLE II

PREDICATES REQUIRED FOR PROOF OF DOMINANCE CONDITION

(*Dominates*? ?*x* ?*y*)
(*Output-intersituational-variables*? ?*v*)
(*looser-constraints-on-variables*? ?*vars* ?*x* ?*y*)
(*looser-constraint-on-variable* ?*d* ?*x* ?*y*)
(*Intersituational* ?*a*)
(*less-equal*? ?*a* ?*b* ?*x* ?*y*)
(*Variable-in-con*? ?*b* ?v_1 ?*y*)
(*Operational* ?v_1)
(*State-variable-value* ?val_1 ?v_1)
(*form*? ?*c* (?*d*?*rel*?*a*))

straints between two states in the branching structure. This is a predicate that connects the general theory to the specific details of the problem. It can be done manually or by the computer deducing the structure based on the definitions of variable types we gave in Section II, E.

Then if there are no variables, the following predicate is satisfied:

(*looser-constraints-on-variables*? ( ) ?*x* ?*y*)

This completes the representation of the sufficient theory required for the flowshop example. It consists of about 10 different predicates listed in Table II and configured in four different implications (rules). These predicates have an intuitive appeal, and are not complex to evaluate, thus the sufficient theory could be thought of as being "simple." The theory is capable of deriving the equivalence–dominance condition in flowshop problem. It is, however, expressed in terms that could be applied to any problem with that type of constraint. Thus it has *generality*, and since we can add new implications to deal with new constraint types, it has *modularity*.

Expressing this theory shifts the emphasis of creating specific knowledge for each problem formulation, to developing "pieces" of theory for more general problems. The method allows the computer to put these pieces together, based on the specific details of the problem, and the opportunities that the problem solving reveals.

V. Learning

In the previous sections, we presented three of the four components necessary to carry out the improvement of problem-solving performance for flowshop scheduling, using branch-and-bound algorithms. The first of

these dealt with the declarative representation of the branch-and-bound algorithms, including the representation of both the solutions and control mechanisms. It also introduced a metric to quantify the efficiency of a branch-and-bound algorithm. The second component dealt with the methods that are needed in order to analyze the experience gained from the solution of specific problems and convert it into new knowledge of generic value. The third component covered the representation needed to analyze and generalize the selected portions of the acquired problem-solving experience.

In this section, we will briefly outline the method of *explanation-based learning* that we use to carry out the synthesis of new control conditions. This method has become well established within the AI community, and for more details, see Minton et al. (1990), Dejong (1990), and Mitchell et al. (1986).

The ultimate goal of the learning process is to transform a problem-solving experience into useful control knowledge that can be applied in future problem-solving episodes. We have already stated that the methodology should be sound, which translates to no dominance or equivalence rules being introduced that would violate the optimum-seeking behavior of the branch-and-bound algorithm. In addition, there are several other requirements that the learning methodology should try to include

1. *Generalizations derived from a few problem-solving instances.* Solving branch-and-bound problems is computationally expensive. Thus we would like to be able to achieve improvements in problem solving as rapidly as possible.
2. *Branching structure generalization.* It is unlikely that we will be solving problems with exactly the same length of alphabet all the time. Thus, the underlying branching structure on which the control conditions act is subject to change. It is important to be able to allow for these changes without having to re-learn the control rules.

A. Explanation-Based Learning

The following definition of explanation-based learning is taken from Minton et al. (1990).

Given:

1. Target concept definition. A concept definition describing the concept to be learned.
2. Training example. An example of the target concept.

3. Domain theory. A set of rules and facts to be used in explaining how the training example is an example of the target concept.
4. Operationality criterion. A predicate over concept definitions, specifying the form in which the learned concept must be expressed.

Determine:

A generalization of the training example that is a sufficient concept description for the target concept and that satisfies the operationally criterion.

The purpose of this section is to try and link these abstract components to the learning task which we have defined.

1. Target Concept Definition

The targets of our learning are new dominance and equivalence conditions. During problem solving each existing dominance or equivalence will pick out a subset of all the pairs of DDPs that appear in the branching structure. In the flowshop example, the pairs of DDPs are representative of partial schedules that share certain features, and whose properties obey certain relationships, such as having equal end-times and having the same set of batches scheduled. Thus, the target concept is defined by the set of all instances that belong to it; in other words, *our target concept is a dominance or equivalence condition, where the instances are all the pairs of DDPs for which the condition is true.*

Discrete decision processes (DDPs) are defined by the three parameters Σ, S, f. It is important to understand the *coverage* of the dominance and equivalence rules in terms of how much we can change the structure of the DDP before we invalidate the condition. It should be clear that any change to S or f will change the target concept definition, since we may no longer be able to guarantee that the necessary conditions on dominance and equivalence are satisfied. For example, if we were to redefine the objective of the flowshop problem to include some weighted value of the mean flowtime, or introduce a new property such as a lateness or earliness penalty, we could not be sure that $f(x) < f(y)$. Similarly, if we were to include extra conditions on the feasibility of a schedule, such as forbidding certain batches from following each other, we would again potentially disrupt the previous ordering on $f(x), f(y)$.

Having demonstrated that S, f must remain constant, we must now define how Σ is allowed to change. If we examine the necessary conditions on dominance and equivalence, Σ is not explicitly mentioned, but we are required to ensure that for all subsequent DDPs, certain properties hold. This suggests that, provided the *interpretation* of the alphabet remains the same, we can allow its length and thus the symbols it contains to vary. In

our scheduling example, this corresponds to allowing additional batches or batch types to be added to the alphabet, but we would not be allowed to change the interpretation of the alphabet symbol from being associated with a single batch to being a batch type.[2]

In summary, the branch-and-bound algorithm as defined in this chapter assumes that the semantics of "objective function," "feasibility," and "branching operation" are fixed with respect to the problem class. As long as their interpretations are not changed, the derived equivalence and dominance rules would remain valid.

Our last concern is for any change in the control information itself. Such change can be caused by a change in the lower-bound function, or a change in the dominance and/or equivalence conditions. In the former case, we must again abandon our existing dominances and equivalences, unless the altered lower-bound function, g', satisfies the condition

$$g'(x) < g'(y) \Rightarrow g(x) < g(y),$$

or, unless we abandon lower-bound consistency.

Changes in dominance conditions can, in very special circumstances, lead to possible expansion of more nodes, but *only* in situations where we have relaxed our conditions on dominance (for details, see Realff (1992)).

2. *Training Example*

A training example is an instance of the target concept, which in our case is a pair of DDPs, and their enumeration via the branch-and-bound algorithm. The information about the training example that the learning algorithm manipulates is dependent on the sufficient theory. For each DDP the problem-specific predicates will be constructed using variables, their values, and the constraints associated with the DDP. These problem-specific predicates given the values from DDPs constitute the *facts* about the example.

3. *Domain Theory*

The domain theory consists of the horn clauses that represent

1. The general implications of the sufficient theory (i.e., the state-space sufficient theory).
2. The implications that link the general theory to the specific problem structure
3. The facts associated with the training example.

[2]Note, however, in this case, this change would make a change in S also.

4. Operationality Criterion

In an earlier section, we had alluded to the need to stop the reasoning process at some point. The operationality criterion is the formal statement of that need. In most problems we have some understanding of what properties are easy to determine. For example, a property such as the processing time of a batch is normally given to us and hence is determined by a simple database lookup. The optimal solution to a nonlinear program, on the other hand, is not a simple property, and hence we might look for a simpler explanation of why two solutions have equal objective function values. In the case of our branch-and-bound problem, the operationality criterion imposes two requirements:

1. The predicates themselves must be easy to evaluate. Thus, we will restrict ourselves to simple comparison predicates, ($<$, $\leq$, $+$, $\geq$, $>$), and predicates that extract properties from alphabet symbols and DDPs.
2. The predicates have to express fact about the DDPs for which we are trying to prove dominance or equivalence. This restriction is clearly necessary to avoid declaring the explanation complete before we have expressed the conditions at the appropriate level in the branching structure.

In choosing the definition of operationality, we are deciding the tradeoff between the number of predicates required to evaluate the condition versus the complexity of each predicate. This tradeoff further reduces to the cost of matching the facts to the predicates, versus the cost of evaluating a given match. If there are many different ways to match the facts to the predicates, but only one is successful, we will tend to have high match cost, unless we can find some ordering of the predicates, such that the unsuccessful ones are eliminated early. This topic has been the subject of much discussion; see Minton et al. (1990) for details.

To express operationality within our representation framework, we will use two methods:

1. Implicitly, we can express the fact that a predicate is operational by not including it on the right-hand side of an implication. In this case, it can never be further expanded or explained, and hence if it is not operational, we will fail to generate a satisfactory explanation. This definition makes it difficult to distinguish between the operationality of facts at one level of the branching structure versus another.
2. Explicitly, we can express the operationality by including a predicate (*Operational*), which does not appear as a consequent. We can then assert that certain facts are operational by including (*Operational fact* $- i$) in the database.

This explicit inclusion of operationality allows us to declare explicitly some facts about the pair (x, y), such as their state variable values, as operational. This will not permit the explanation to stop at other partial solutions, whose states have not been declared operational; thus we will use this approach.

B. Explanation

So far we have described all the inputs to the explanation-based learning procedure. We must now describe a few of the basic feature of the algorithm itself.

The first step of the explanation-based learning paradigm is the construction of an explanation of why the training example is an example of the target concept. This explanation will be constructed by backward chaining from the instantiated target concept definition, through the horn clause implications, to reach a set of facts that are true of the example, and that satisfy the operationality criterion. In the backward chain process, we *unify* (Lloyd, 1987) the successively generated subgoals with the consequences of the horn clause implication.

This is high-level description of the explanation process can be illustrated in the flowshop example. To instantiate the target concept, we use the partial solution strings that represent x and y; call them σ_x, σ_y. Thus, our target concept, for example, becomes

$$(\textit{Dominates}?\ \sigma_x\ \sigma_y).$$

We now search our horn clause implications for one whose consequent unifies with (Dominates? σ_x σ_y). The unification algorithm returns both whether a unification exists and the bindings generated by the algorithm. The bindings of a unification are the substitutions from the specific example for the variables in the consequent. Thus, to unify (Dominates? σ_x σ_y) with (Dominates? $?x$ $?y$), we can do so by binding $?x$ to σ_x and $?y$ to σ_y. Having found an implication whose consequent matches the goal, we backward-chain to the antecedents of the implication, and set these up as subgoals. In these antecedents, we carry the bindings of the consequent through to the corresponding variables of the antecedent. Thus, if the implication were

$$\left.\begin{array}{l}(\textit{Intersituational-variables}?\ ?v) \\ (\textit{looser-constraints-on-variables}?\ ?v\ ?x\ ?y)\end{array}\right\} \Rightarrow (\textit{Dominates}\ ?x\ ?y)$$

then the subgoals would simply be

(*Intersituational-variables*? ?v)
(*looser-constraints-on-variables*? ?v σ_x σ_y)

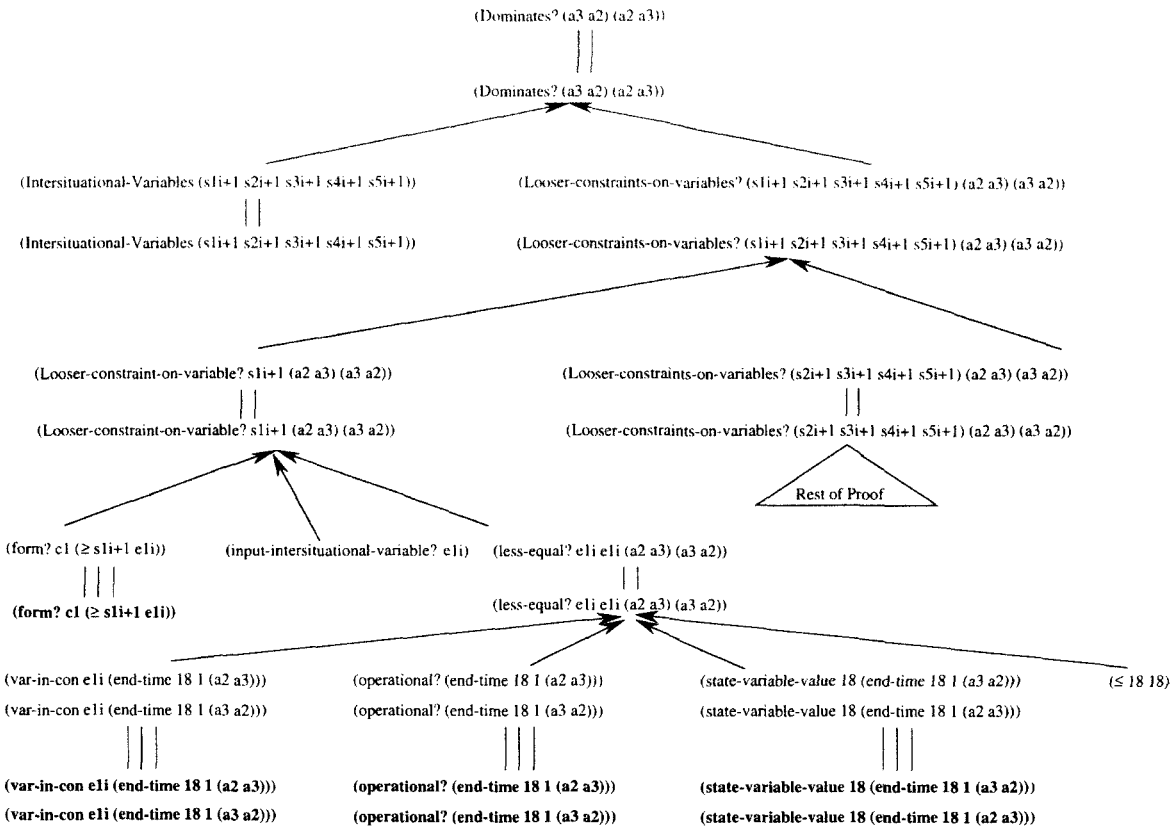
The antecedents are ordered in such a way that bindings, which are necessary to solve certain subgoals, are found before the subgoals are expanded. Thus, in the preceding case, binding for ?v will be sought before trying to solve the subgoal (*Looser-constraints-on-variables*? ?v ?σ_x ?σ_y). Some of the antecedents may not appear as the consequences of other implications, and to satisfy them, we need to search for matching facts in the database. In the preceding example, we would have entered the intersituational variables of the problem formulation as a fact, and hence the subgoal (*Intersituational-variables*? ?v) will be solved by binding ?v to a list of those variables. Figure 10 details the resulting explanation, including the unifications or bindings.

The explanation thus consists of a set of implications that connect the original goal, and subgoals, to the facts of the example. This set of implications is interconnected via the bindings that hold between the subgoal and the consequent of the implication used to solve it. The preceding qualitative description of the form of an explanation can be formalized in graph-theoretical terms (Mooney, 1990). For our purposes, the important feature of the explanation is its *structure*. The structure is the pattern of the inference and the bindings that enable us to connect the objects of the original goal to the facts at the leaves of the explanation.

1. Specific Explanation

The specific explanation structure for the flowshop problem is given in Fig. 10. In the example we have assumed that the sufficient condition is satisfied by having all the end-times of x less than or equal to those of y. Thus the proof begins by selecting the appropriate variable set, and proceeds to prove that each variable is more loosely constrained in x than in y. The intersituational variables in the flowshop problem are the start-times of the next state.

If we assume that the variables are ordered from the first unit to the last, we will begin by examining the start-times on the first unit. The start-times are not operational, but the end-times on the units for partial solution x, y are. The start-times of the first unit in the next state, which are equal to the end-times of the first unit in x and y are analyzed by using one of the less-equal implications. The end-times, which are operational, are then compared.



Facts are in bold.

FIG. 10. Proof of simple dominance condition.

A similar analysis is carried out for the start-times of the second unit. Note that during the solution of the subgoal, it is possible that the constraint involving the end-time of the previous unit will be tried. This will fail since the end-time is not an input-intersituational variable. This process is repeated for each subsequent start-time, and in each case the subgoal is resolved the same way.

C. Generalization of Explanations

According to Mooney (1990), the process of generalization has an input an instance of an explanation; and as output, the explanation, divorced from the specific facts of the example, but retaining the structure implied by it.

We will divide the process of generalization into two parts. In the first, we will consider only the generalization of the explanation while leaving its structure unchanged. In the second, we will consider the generalization of the structure itself.

1. Variable Generalization

During the solution of a specific example we take the implications (rules) and instantiate them with the specific facts of the example. Thus part of the explanation might look like

$$\left.\begin{array}{l}(\textit{Variable-in-con}\ e_{i1}\ (\textit{end-time}\ e_{i1}\ 18\ (a_2\ a_3)))\\ (\textit{Variable-in-con}\ e_{i1}\ (\textit{end-time}\ e_{i1}\ 18\ (a_3\ a_2)))\\ (\textit{Operational?}\ (\textit{end-time}\ e_{i1}\ 18\ (a_2\ a_3)))\\ (\textit{Operational?}\ (\textit{end-time}\ e_{i1}\ 18\ (a_3\ a_2)))\\ (\textit{State-variable-value}\ 18\ (\textit{end-time}\ e_{i1}\ 18\ (a_2\ a_3)))\\ (\textit{State-variable-value}\ 18\ (\textit{end-time}\ e_{i1}\ 18\ (a_3\ a_2)))\\ (<=18\ 18)\end{array}\right\}$$

$$\Rightarrow (\textit{less-equal?}\ e_{i1}\ e_{i1}\ (a_2\ a_3)(a_3\ a_2))$$

and this would connect to the facts of the example. The explanation for why the variable $S_{i+1,1}$ is no more constrained in the solution $(a_2\ a_3)$ than in $(a_3\ a_2)$ is given in Fig. 11, which includes the specific facts and the relevant bindings of variables. From this specific instance of explanation, we generate the explanation structure, which removes the dependency on

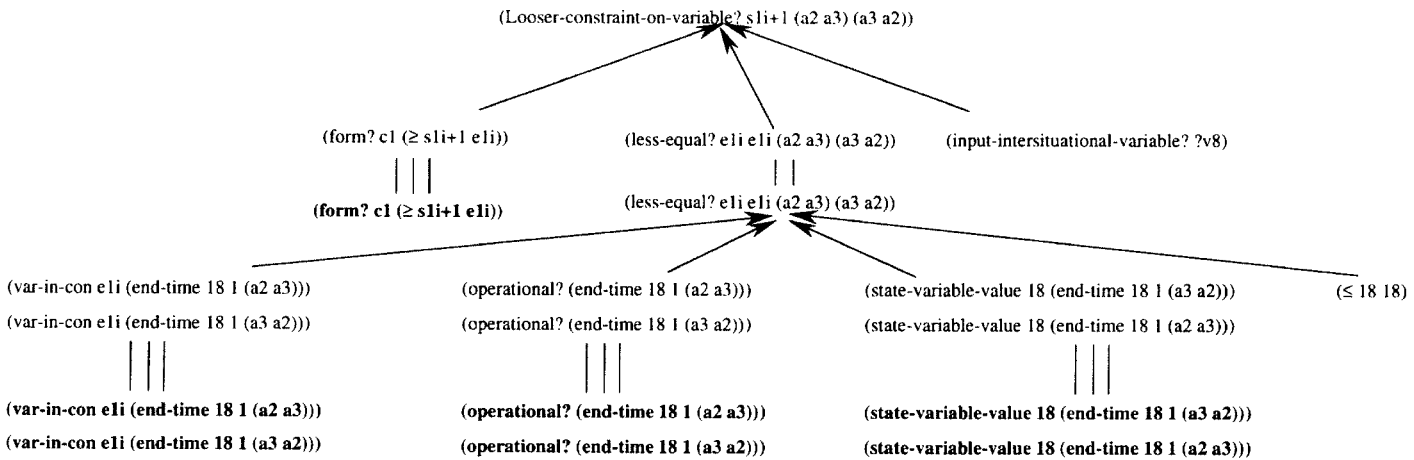


FIG. 11. Explanation with original bindings.

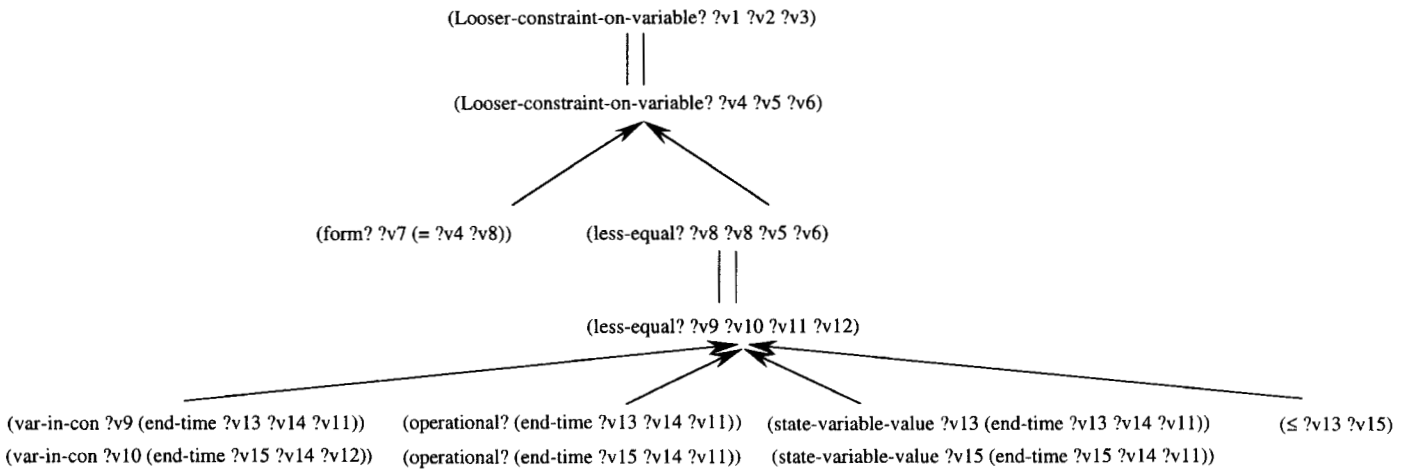
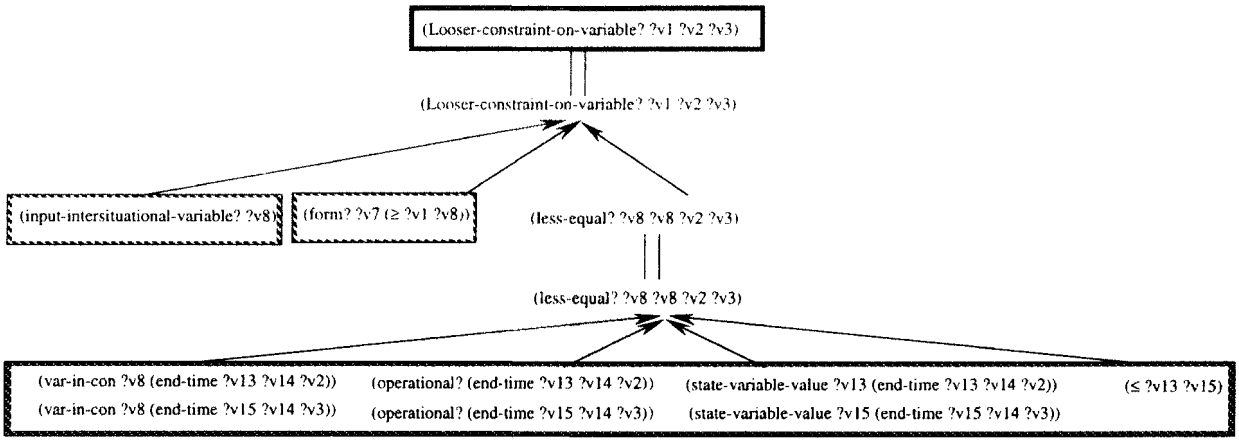
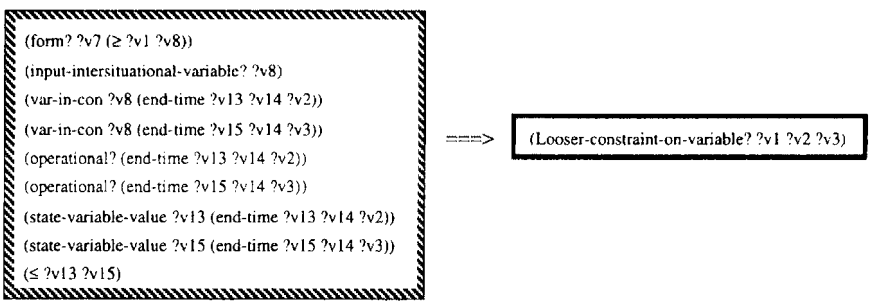


FIG. 12. Explanation structure with uniquized bindings.





Final Condition - The leaves of the explanation and the top level goal.

FIG. 13. Explanation structure with unified bindings and the final rule.

TABLE III

GENERALIZATION ALGORITHM

Let γ be the null substitution { } for each equality between p_i and p_j in the explanation structure do
  Let θ be the *most-general-unifier* of p_i and p_j
  Let γ be $\gamma\theta$
for each pattern p_{kN} in the explanation structure do
  replace p_k with $p_k\gamma$

the specific facts, and makes the arguments of the predicates unique (Fig. 12). Finally, we compute the most *general unifier*, so that at each antecedent consequent match the predicates are identical. The resulting generalized explanation is given in Fig. 13. Generalization of variables thus requires the computation of the most general unifier, which is the same as finding the unification, along with an algorithm for achieving the generalization, given in Table III, taken from Mooney (1990).

The purpose of creating and generalizing the explanation was to generate a way of identifying whether an instance—in this case, a pair of partial solutions—was a member of the target concept, i.e., dominance. The generalized explanations forms a "trace" from the difficult-to-evaluate predicate, at the root, *Dominates*? to a set of easier-to-evaluate predicates at the leaves. To form our condition, we gather up the leaves of the generalized explanation, and use them as antecedents to a new implication:

$$Leaf_1 \wedge Leaf_2 \wedge \ldots \Rightarrow goal\text{-}predicate.$$

In the preceding example:

$$\left.\begin{array}{l}
(form?\ ?con_1\ (?a \geq ?b)) \\
(Input\text{-}intersituational\text{-}variable?\ ?b) \\
(Variable\text{-}in\text{-}con?\ ?b\ (end\text{-}time\ ?val_1\ ?j\ ?x)) \\
(Variable\text{-}in\text{-}con?\ ?b\ (end\text{-}time\ ?val_2\ ?j\ ?y)) \\
(Operational\ (end\text{-}time\ ?val_1\ ?j\ ?x)) \\
(Operational\ (end\text{-}time\ ?val_2\ ?j\ ?y)) \\
(State\text{-}variable\text{-}value\ ?val_1\ (end\text{-}time\ ?val_1\ ?j\ ?x)) \\
(State\text{-}variable\text{-}value\ ?val_2\ (end\text{-}time\ ?val_2\ ?j\ ?y)) \\
(<=\ ?val_1\ ?val_2)
\end{array}\right\}$$

$$\Rightarrow (Looser\text{-}constraint\text{-}on\text{-}variable\ ?a\ ?x\ ?y).$$

In this case, we have omitted one intermediate step of reasoning, but in general, there could be many steps between the original goal and the operational predicates. Any future application of the condition will not be required to perform those intermediate steps, or any of the associated search for implications with matching consequences.

2. *Generalizations of the Proof Structure*

The generalization procedure described above carefully preserves the structure of the proof; it does not attempt to take into account any repetitive structure, which might itself be capable of being generalized. In solving branch-and-bound problems, repetitive structure can occur very easily, since we are performing roughly similar functions as we branch from a parent node to a child, whatever level we are at in the tree.

Furthermore, in the context of flowshop scheduling we have repetitive calculations being performed not only at each level of the branching tree but also within a given node, since each unit essentially has its start-time and end-time calculated by identical procedures. The difficulty we face is understanding when generalization of the proof structure is justified, and when it is simply coincidental that certain elements of the proof have been repeated.

We will adopt the procedure due to Shavlik (1990), in which the justification for generalizing the proof structure is the existence of a recursive pattern of the application of the implications. A recursive pattern implies that at some point in proving a goal a subgoal of the same type, i.e., the same predicate, is generated which contains different arguments. This type of proof structure generalization will enable us to generalize certain facets of the problem structure. Since we have defined the parsing of the list of output intersituational variables recursively, we will be able to generalize the number of items in the list, or in this case the number of start-times. Generalizing the number of start-times implicitly enables us to have an arbitrary number of units in the flowshop. Figure 14 presents the overall structure of the proof.

The way we have stated the domain theory for the state-space representation has enabled us to avoid making explicit reference to the alphabet symbol properties. However, if in other formulations we need to refer to these properties, we would again use a recursive parsing of the list of symbols to enable generalization over the size of the alphabet.

The method that Shavlik developed to carry out the structure generalization is complex, and requires the introduction of an extension to the horn clause deduction scheme to allow the repetitive application of a

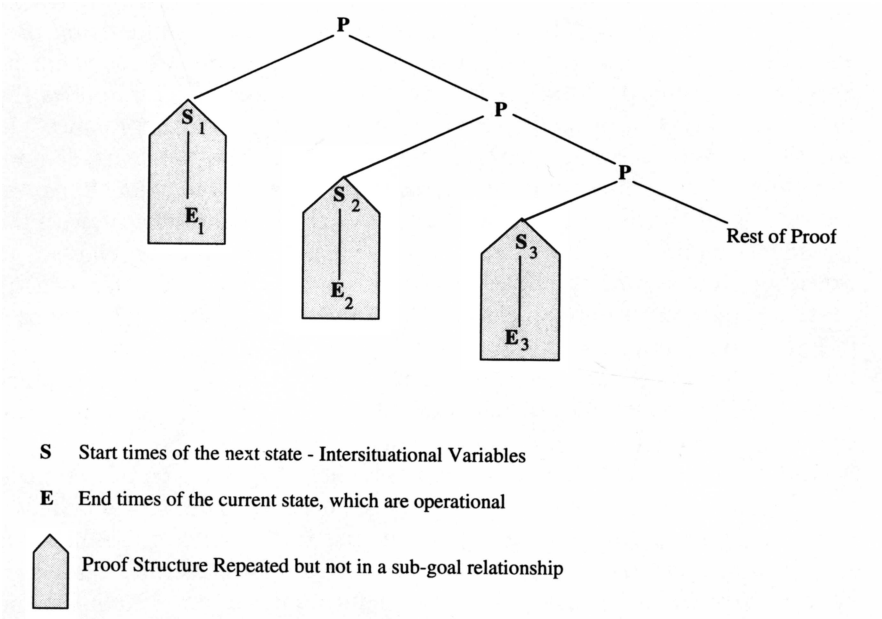


FIG. 14. Abstract structure of the proof with repeated subgoals.

group of implications without going through the general backward chaining procedure. The details can be found in Shavlik (1990).

VI. Conclusions

This chapter has presented a methodology for integrating machine learning into the branch-and-bound algorithm. The methodology has been shown to be capable of acquiring a simple dominance condition used in flowshop scheduling algorithms. In Realff (1992) the application of the methodology to a more general scheduling problem (Kondili et al. 1993) is presented. The general problem is cast as an MILP and a sufficient theory of dominance developed for MILP is used to find a dominance rule. Thus we have developed and represented sufficient theories for dominance in both a state-space problem-solving formulation and the traditional optimization formulation of an MILP, both of which cover a wide range of problems of interest to chemical engineers.

We believe this work can be extended in several directions. First, we have shown how to acquire the dominance condition expressed in first-

order logic; the next step is to use automatic programming concepts (Barstow, 1986) to convert this expression back into the underlying programming language of the branch-and-bound algorithm, to speed up its application within problem solving. Second, we have concentrated on the theoretical aspects of improving the problem-solving performance. In addition to the theoretical improvements in efficiency, we must demonstrate empirical improvements in the execution time of the algorithm, or in the size of the problems tackled. The computer would *experiment* with the dominance and equivalence rules that it acquired, noting the changes in execution time solving the problem with or without the rules. It would try to detect features of the numerical data that make a given problem harder or easier to solve, using an appropriate empirical learning method such as that presented by Saravia (second chapter in this volume). This experience could then be converted into rules for selecting the appropriate configuration of the branch-and-bound algorithm.

This approach has opened up the possibility of a new type of optimization system, one that can learn from its own experience, test its conclusions, and configure itself to best solve the particular problem, based on its structure and on its data. By reducing some of the algorithm designer's synthetic burden of constructing new algorithms, or new extensions to existing general-purpose methods for each problem, allows the designer to concentrate on developing more widely applicable knowledge embodied in the sufficient theories of the system. It takes advantage of both the solution technology and problem-solving architectures of operations research used to *calculate* optimal answers, and the *reasoning* methods of artificial intelligence: a synthesis of a traditional numerical procedure and symbolic deductive techniques.

References

Abelson, H., Eisenberg, M., Halfant, M., Katzenelson, J., Sacks, E., Sussman, G.J., and Yip, K., Intelligence in scientific computing. *Commun. ACM* **32** (1989).

Baker, K.R., "Introduction to Sequencing and Scheduling." Wiley, New York 1974.

Baker, K.R., A comparative study of flow-shop algorithms. *Oper. Res.* **23**(1) (1975).

Barstow, D., A perspective on automatic programming, *In* "Readings in Artificial Intelligence and Software Engineering" (C. Rich and R.C. Waters, eds.), pp. 537–539. Morgan Kaufmann, San Mateo, CA, 1986.

Clocksin, W.F., and Mellish, C.S., "Programming in Prolog." Springer-Verlag, Berlin (1984).

Dantzig, G.B., "Linear Programming and Extensions." Princeton University Press, Princeton, NJ, 1963.

Dejong, G., and Mooney, R., Explanation-based learning: An alternate view. *Mach. Learn.* **1**, 145–176 (1986).

Garey, M.R., and Johnson, D.S., "Computers and Intractability: A Guide to the Theory of NP-Completeness." Freeman, New York, 1979.

Green, C.C., Application of Theorem proving to problem solving. *Proc. Int. Joint Conf. Art. Intell. 1st*, Washington, DC, 1969, pp. 219–239 (1969).

Ibaraki, T., The power of dominance relations in branch bound algorithms. *JACM* **24**(2), 264–279 (1977).

Ibaraki, T., Branch and bound procedure and state-space representation of combinatorial optimization problems. *Inf. Control* **36**, 1–27 (1978).

Jackson, J.R., Scheduling a Production Line to Minimize Maximum Lateness," Res. Rep. No. 43, Management Science Research Project. University of California, Los Angeles, 1955.

Karp, R.M., and Held, M. Finite-state processes and dynamic programming. *SIAM J. Appl. Math.* **15**, 698–718 (1967).

Kondili, E., Pantalides, C.C., and Sargent, R.H.W., A general algorithm for short-term scheduling of batch operations. I. MILP formulation. *Comput. Chem. Eng.* **17** (2) 211–228 (1993).

Kumar, V., and Kanal, L.N., A general branch and bound formulation for understanding and synthesizing and/or tree search procedures. *Art. Intell.* **21** 179–198 (1983).

Lagweg, B.J., Lenstra, J.K., and Rinnooy Kan, A.H.G., A general bounding scheme for the permutation flow-shop problem. *Oper. Res.* **26**(1) (1978).

Lloyd, J.W., "Foundations of Logic Programming," 2nd ed., Springer-Verlag, Berlin, 1987.

Minton, S., Carbonell, J., Knoblock, C., Kuokka, D., Etzioni, O., and Gil, Y., Explanation-based learning: A problem solving perspective in machine learning. *In* "Machine Learning: Paradigms and Methods" (J.G. Carbonell, ed). Massachusetts Institute of Technology/Elsevier, Cambridge and London, 1990.

Mitchell, T., Keller, R., and Cedar-Cabelli, S., Explanation-based generalization: A unifying view. *Mach. Learn.* **1**, 47–80 (1986).

Mooney, R.J., "A General Explanation-Based Learning Mechanism and its Application to Narrative Understanding." Morgan Kaufmann, San Mateo, CA (1990).

Nemhauser, G.L., and Wolsey, L.A., "Integer and Combinatorial Optimization." Wiley, New York (1988).

Nilsson, N.J., "Principles of Artificial Intelligence." Palo Alto, CA, 1980.

Rajagopalan, D., and Karimi, I.A., Completion times in serial mixed-storage multiproduct processes with transfer and set-up times. *Comput. Chem. Eng.* **13**(1/2), 175–186 (1989).

Realff, M.J., "Machine Learning for the Improvement of Combinatorial Optimization Algorithms: A Case Study in Batch Scheduling." Ph.D Thesis, MIT., Cambridge, MA, 1992.

Robinson, J.A., A machine-oriented logic based on the resolution principle, *JACM* **12**(1), 23–41 (1965).

Shah, N., Pantelides, C.C., and Sargent, R.W.H., A general algorithm for short-term scheduling of batch operations. II Computational issues. *Comput. Chem. Eng.* **17**(2), 229–244 (1993).

Shavlik, J.W., "Extending Explanation-Based Learning by Generalizing the Structure of Explanations." Morgan Kaufmann, San Mateo, CA, 1990.

Simon, H.A., Search and reasoning in problem solving. *Art. Intell.* **21**, 7–29 (1983).

Szwarc, W., Elimination methods in the $m \times m$ sequencing problem. *Nav. Res. Logist. Q.* **18**, 295–305 (1971).

Thayse, A., "From Standard Logic to Logic Programming. Wiley, New York, 1988.

Wiede, W., and Reklaitis, G.V., Determination of completion times for serial multiproduct processes. I-III. *Comput. Chem. Eng.*, **11**(4), 337–368 (1987).

Yip, K., "KAM: A System for Intelligently Guiding Numerical Experimentation by Computer." MIT Press, Cambridge, MA, 1992.