HOW GOOD ARE PARALLEL AND ORDERED DEPTH-FIRST SEARCHES?

Guo-Jie Li and Benjamin W. Wah
Department of Electrical and Computer Engineering
and the Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 W. Springfield Avenue
Urbana, IL 61801

ABSTRACT

Parallel depth-first searches are widely used to solve combinatorial optimization and decision problems in artificial intelligence and operations research. These problems are represented by OR-trees and AND/OR-trees. The performance of parallel depth-first searches may be difficult to predict due to the nondeterminism and anomalies of parallelism. In this paper we have derived the performance bounds of parallel depth-first searches with respect to optimization problems represented as OR-trees and have verified these bounds by simulations. These bounds provide the theoretical foundation to determine the number of processors to assure a near-linear speedup. The conditions to cope with parallel-to-parallel anomalies are also investigated. For decision problems represented by AND/OR-trees, such as evaluating logic programs, we have studied an ordered depth-first search that rearranges nodes in each level of the AND/OR tree to minimize the expected search cost.

1. INTRODUCTION

Combinatorial-search problems can be classified into two types. The first type is decision problems that decide whether at least one solution exists and satisfies a given set of constraints [21]. Theorem-proving, expert systems, and evaluating a logic program belong to this class. The second type is combinatorial extremum-search or optimization problems that are characterized by an objective function to be minimized or maximized and a set of constraints to be satisfied. Practical problems such as finding the shortest path, planning, finding the shortest tour of a traveling salesman, job-shop scheduling, packing a knapsack, vertex cover, and integer programming belong to this class.

The non-terminal nodes in a search tree (or graph) can be classified as AND-nodes and OR-nodes. An *AND-node* represents a problem (or subproblem) that is solved only if all its descendant nodes have been solved, while an *OR-node* represents a problem (or subproblem) that is solved only if any of its immediate descendants is solved. Based on these two kinds of nodes, a combinatorial search can be classified into an AND tree, OR-tree, and AND/OR-tree search [25]. Note that a general dataflow graph contains AND-nodes and OR-nodes that relate the descendant nodes, as well as other nodes that relate the ascendant nodes.

In this paper we will concentrate on evaluating problems that arise in nondeterministic computations, namely, those problems that are represented as OR-trees or AND/OR-trees. As an AND-tree represents deterministic computations and all nodes in it must be evaluated, it will not be discussed here [14]. Due to space limitation, we will only present results on the depth-first search strategy. Results on other strategies with respect to OR-trees can be found elsewhere [15, 16].

An OR-tree is a state-space tree in which all non-terminal nodes are OR-nodes, while an AND/OR-tree is a problem-reduction representation that consists of AND-nodes and OR-nodes. Many AND/OR-tree search procedures, such as AO*, SSS*, and dynamic programming, can be formulated as a general branch-and-bound (B&B) procedure [8, 19], which is a well-known OR-tree search method. Likewise, evaluating a logic program can be represented as an OR-tree or AND/OR-tree search [7].

Both combinatorial OR-tree and AND/OR-tree search procedures can be characterized by four constituents: a branching rule, a selection rule, an elimination rule, and a termination condition. The first two rules are used to decompose problems into simpler subproblems and to appropriately order the search. The last two rules are used to to eliminate unnecessary subproblems. Appropriately ordering the search and restricting the region searched are key ideas behind any search algorithms.

The rules to guide the search and to prune unnecessary searches may differ for optimization and decision problems. In optimization problems, a lower bound of the objective value for each nonterminal node can be used to guide the search and to prune nodes that cannot lead to a better solution. Dominance tests, such as α—β pruning, can also be adopted as elimination rules. In decision problems, it was found that the ratio of the success probability of a subproblem to the estimated overhead of evaluating the subproblem is useful to guide the search [21, 1, 12]. The elimination rules are more restricted in decision problems, such as evaluating a logic program. Pruning a subproblem with a smaller success probability or a larger search cost may remove a possible (and possibly a unique) solution. In this case only when a terminal node is found to be true or false, AND-pruning or OR-pruning rules can be applied [12].

There are three basic selection strategies, namely, depth-first, breadth-first, and best-first searches. A generalized heuristic function can be used to unify these three kinds of search strategies and resolve ambiguities in the heuristic function [4, 10]. To resolve the ambiguity on the selection of subproblems, distinct heuristic values must be defined for the nodes to allow ties to be broken. A path number can be used to define an unambiguous heuristic function. The *path number* of a node in a tree is a sequence of (h+1) integers representing the path from the root to this node, where h is the maximum number of levels of the tree [10, 15]. For example, the path numbers of nodes A, B, C, and D in Figure 1c are 0000, 0100, 0200, and 0300, respectively. Note that the nodes having equal path numbers never coexist simultaneously in the search process. For a depth-first search, the generalized heuristic function is defined as

$$h(P_i) = (\text{path number, level number}) \qquad (1)$$

Although a best-first search expands fewer nodes than a depth-first search, it requires a secondary memory to maintain the large number of active nodes, hence the total time, including the time spent on data transfers between the main and secondary memories, to solve a problem may be longer than that required by the depth-first search. Simulations have shown that the best OR-tree search strategy depends on the accuracy of the problem-dependent lower-bound function [24]. Very inaccurate lower bounds are not useful to guide the search, while very

This research was supported by National Science Foundation Grants DMC 85-19649.

International Conference on Parallel Processing, 1986.

accurate lower bounds will prune most unnecessary expansions. In both cases the number of subproblems expanded by depth-first and best-first searches will not differ greatly, and a depth-first search is better as it requires less memory space.

Extensive studies have been conducted on OR-parallelism [2, 12, 15, 18], but very few studies have been done on analyzing the speedups and efficiency of OR-parallelism. Due to the nondeterminism, combinatorial OR-tree and AND/OR-tree searches are quite different from conventional deterministic numerical computations. Simulation results have revealed that using more processors in parallel depth-first searches might degrade the performance, even when the communication overhead is ignored [10]. The prediction of performance and methods to cope with anomalous behavior are important problems to be studied in designing multiprocessors for parallel depth-first searches and will be addressed in this paper.

To take advantage of the search efficiency of best-first searches while avoiding their memory overhead, an informed depth-first search can be used [20]. In this strategy best-first search is performed locally and depth-first search globally. A special case is one in which all sibling nodes are ordered according to heuristic values of the siblings (a more accurate definition will be given in Section 3). We will show that this *ordered depth-first strategy* is very effective to evaluate logic programs represented as AND/OR trees.

2. PARALLEL DEPTH-FIRST OR-TREE SEARCHES

To predict the number of processors needed to assure a near-linear speedup in a parallel depth-first search, we will derive the bounds on computational efficiency. The results in this section indicate the relationship among the number of iterations required in a parallel depth-first search, the number of processors used, and the complexity of the problem to be solved.

2.1. Model of Efficiency Analysis

In analyzing the performance bounds, a synchronous model is assumed, that is, all processors must finish the current iteration before proceeding to the next iteration. This performance results form a lower bound to that of asynchronous models.

The parallel computational model used here consists of a set of processors connected to a shared memory. In each iteration, multiple subproblems are selected and decomposed. The newly generated subproblems are tested for feasibility, eliminated by (exact or approximate) lower-bound tests and dominance tests,* and inserted into the active list(s) if not eliminated. In this model eliminations are performed after branching instead of after selection as in Ibaraki's algorithm [5] to reduce the memory space required.

We have proved that, for best-first searches, the performance is not largely affected by whether the active subproblems are kept in a single shared list or multiple lists [23,15]. However, for depth-first searches, the performance will be problem-dependent when multiple lists are used. In this paper the performance bounds are derived under the assumption that one list is used and that the nodes with the smallest heuristic values are selected in each iteration.

Since subproblems are decomposed synchronously and the bulk of the overhead is on branching operations, the number of iterations, which is the number of times that subproblems are decomposed in each processor, is an adequate measure in both the serial and parallel models. The *speedup* between using k_1 and k_2, $k_2 > k_1$, processors is thus measured by the ratio of the number of iterations when k_1 processors are used to that when k_2 processors are used. Once the optimal solution is found, the time to drain the remaining subproblems from the list(s) is not accounted for, since this overhead is negligible as compared to that of branching operations.

The results proved in this section show the performance bounds of parallel depth-first OR-tree searches for solving optimization problems. The proofs of these theorems require the following definitions on essential nodes. A node expanded in a *serial* depth-first search is called an *essential node*, otherwise it is called a *non-essential node*. The speedup of a parallel depth-first search depends on the number of essential nodes selected in each iteration. An iteration is said to be *perfect* if the number of essential nodes selected is equal to the number of processors, otherwise it is said to be *imperfect*. The *incumbent* at any given time in the search process is the best feasible solution obtained at that time. The incumbent is continuously updated until an optimal solution is found. We denote $T_b(k,\epsilon)$ and $T_d(k,\epsilon)$ as the number of iterations required to find a single optimal (or suboptimal) solution using k, $k \geq 1$, processors in a best-first and depth-first search, respectively, where ϵ is an *allowance function* specifying the allowable deviation of a suboptimal value from the exact optimal value. When an approximate solution is sought, i.e. $\epsilon > 0$, during the search of an OR-tree, an active node P_i is terminated if

$$g(P_i) \geq \frac{z}{1+\epsilon} \qquad \epsilon \geq 0,\ z \geq 0 \tag{2}$$

where z is an incumbent obtained at that time.

2.2. Parallel Depth-First Searches

The following theorem shows that the performance of parallel depth-first searches depends on the problem complexity and the number of distinct incumbents found during the search process.

Theorem 1: For a parallel depth-first OR-tree search with k processors, $\epsilon=0$, and a generalized heuristic function of $h(P_i)$ =(path number, level number), then

$$\left\lceil \frac{T_b'(1,0)-1}{k}+1 \right\rceil \leq T_d(k,0) \leq \left\lfloor \frac{T_d(1,0)}{k}+\frac{k-1}{k}[(c+1)*h-c] \right\rfloor \tag{3}$$

where h is the height of the OR-tree, c is the number of the distinct incumbents obtained during the serial depth-first search, and $T_b'(1,0)$ is the number of essential nodes in a serial best-first search with lower bounds *less than* the optimal-solution value.

Proof: The sequence of iterations obtained during a serial depth-first search can be divided into (c+1) subsequences according to the c distinct monotonically decreasing incumbents obtained. Let the c feasible solutions and their corresponding parents be denoted by $F_1, \ldots, F_c$, and $P_1, \ldots, P_c$. Further, assume that $F_1, \ldots, F_c$ are obtained in the i_1'th, ..., i_c'th iterations, respectively. Hence iterations from 1 to i_1 belong to the first subsequence, and iterations from i_j+1 to i_{j+1} belong to the (j+1)'th subsequence.

We now consider the j'th $1 \leq j \leq c$, subsequence. Let $\theta_{min}(x)$ be the level with the minimum level number in which some active essential nodes, whose heuristic values are between $h(P_{j-1})$ and $h(P_j)$, reside in the x'th iteration. For levels less than $\theta_{min}(x)$, all active nodes, whose heuristic values are between $h(P_{j-1})$ and $h(P_j)$, are non-essential. We show that Iteration x is imperfect only if all essential nodes, whose heuristic values are between $h(P_{j-1})$ and $h(P_j)$, in $\theta_{min}(x)$ are selected for expansion. Suppose that Iteration x is imperfect, the selected non-essential node must have heuristic value larger than $h(P_j)$, because otherwise this node would have to be eliminated by the feasible solution F_{j-1} (F_0 is the initial feasible solution obtained by a heuristic method). Thus after Iteration x is carried out, $\theta_{min}(x)$ must be increased by at least one. Consequently, after at most h imperfect iterations, F_j, must be found.

During the last subsequence of iterations, since the optimal solution has been generated, all iterations are imperfect only if less than k nodes are selected in each iteration. In other words, an imperfect iteration implies that all currently active nodes are selected and expanded, and only descendants of these nodes can be active in the next iteration. Hence no active node remains after at most h imperfect iterations in the last subsequence. The previous analysis shows that at most (c+1)*h imperfect

* Dominance tests will not be discussed in this paper due to space limitation.

iterations can appear in a parallel depth-first search. Since at least one node in each iteration in the parallel case belongs to Φ^1, the set of nodes expanded in the serial depth-first search [10, 15], the upper bound of $T_d(k,0)$ can be derived as

$$T_d(k,0) \leq \left\lfloor \frac{T_d(1,0)-(c+1)*h}{k} + (c+1)*h \right\rfloor$$

In the above discussion, the expansion of the root is counted in each of the (c+1) subsequences. Since the root is only expanded once, the above upper bound should be compensated by the additional number of times that the root is expanded (Eq. (3)).

The lower bound on $T_d(k,0)$ can be proved easily because all essential nodes in a serial best-first search with lower bounds less than the optimal solution must be expanded in the parallel depth-first search. □

For problems such as integer programming and 0-1 knapsack problems, all feasible solutions are located in the bottom-most level of the OR-tree. In this case the following corollary shows that all essential nodes of a serial depth-first search must be expanded in a parallel depth-first search, and a tighter lower bound is obtained.

Corollary 1: In searching an OR-tree using a parallel depth-first search and a heuristic function of (path number, level number), if $\epsilon = 0$ and all feasible solutions are in Level h, then

$$\left\lfloor \frac{T_d(1,\epsilon) - 1}{k} + 1 \right\rfloor \leq T_d(k,\epsilon) \qquad (4)$$

where h is the maximum number of levels of the OR-tree.

Proof: The proof is omitted due to the space limitation [13, 10].

The bounds in Theorem 1 are tight in the sense that we can construct examples to achieve the lower- and upper-bound of computational times. These degenerate cases occur rarely. Although c, the number of distinct incumbents, is unknown until the solution is found, c is usually small and can be estimated when integral solutions are sought. It has been observed that c is less than 10 for vertex-cover problems with less than 100 vertices. For most integer programming problems, $c \approx 1$. In these cases the range on $T_d(k,0)$ is tight, and a near-linear speedup can be achieved in a large range of k.

Let w be $T_d(1,0)/h$. w can be viewed as the "average width" of an OR-tree, which only consists of essential nodes. Eq. (3) can be rewritten as

$$\frac{T_d(1,0)}{T_d(k,0)} \geq \frac{k*w}{w+(c+1)(k-1)} \qquad (5)$$

From Eq. (5), it is easy to see that if $w >> k$ and c is small, then the speedup is close to k; whereas if $w << k$, then the lower-bound speedup is close to $w/(c+1)$.

In Table 1, the theoretical bounds derived above are compared with the simulation results of parallel depth-first searches to solve two 35-object knapsack problems. In generating the knapsack problems, w(i), the weights, were chosen randomly between 0 and 100 with a unform distribution, and the profits were set to be $p(i) = (w(i) + 10)$. This assignment is intended to increase the complexity of the randomly generated problems. The results demonstrate that the bounds on parallel depth-first searches are tight, hence its performance can be predicted quite accurately. Table 1 also shows that the speedup depends strongly on w. In Case 1 $w \approx 2023$, and a near-linear speedup of 0.88k is achieved with 256 processors. In Case 2 $w \approx 188$, and a speedup of 0.29k is obtained with 256 processors. Note that when the number of processors is large, the number of essential nodes in each imperfect iteration of the parallel depth-first search is usually larger than one. In contrast to the upper bound in Eq. (3), which was derived with the assumption of one essential node in each imperfect iteration, $T_d(k,0)$ may be much smaller than the upper bound. Simulations have also revealed

No. of Proc.	Lower bound	No. of iterat.	Upper bound	Speedup
Case 1				
1	70790	70790	70790	1.000
2	35395	35630	35787	1.987
4	17698	18044	18285	3.923
8	8849	8884	9534	7.968
16	4425	4460	5159	15.872
32	2213	2247	2971	31.504
64	1107	1143	1877	61.934
128	554	592	1330	119.578
256	277	316	1057	224.019
Case 2				
1	6566	6582	6582	1.000
2	3283	3488	3513	1.887
4	1642	1940	1978	3.393
8	821	1161	1211	5.669
16	411	777	827	8.471
32	206	584	635	11.271
64	103	485	539	13.571
128	52	219	491	30.055
256	26	90	467	73.133

Table 1. Comparisons between theoretical bounds and simulation results on parallel depth-first searches for knapsack problems with 35 objects. ($T_b'(1,0)=T_b(1,0)$. During depth-first searches, c=22 in Case 1 and c=12 in Case 2.)

that for a number of OR-tree search problems, $T_d(k,0)$ may be very close to $T_b(k,0)$.

Analogous to the proof of Theorem 1, the upper bound on $T_d(k,\epsilon)$, $\epsilon > 0$, can be derived. To find the lower bound on $T_d(k,\epsilon)$, let f_o be the optimal-solution value and $MINT_b(\epsilon)$ be the minimum number of nodes to be expanded in the approximate best-first search. $MINT_b(\epsilon)$ represents the number of nodes whose lower bounds are less than $f_o/(1+\epsilon)$, since these nodes must be expanded in the best case. $MINT_b(\epsilon)$ may be estimated from the distribution on the number of subproblems with respect to lower bounds. From the above analysis, we get

$$\left\lfloor \frac{MINT_b(\epsilon)-1}{k} + 1 \right\rfloor \leq T_d(k,\epsilon) \leq \left\lfloor \frac{T_d(1,\epsilon)}{k} + \frac{k-1}{k}[(c+1)*h-c] \right\rfloor \qquad (6)$$

2.3. Coping With General Parallel-to-Parallel Anomalies

Some results on coping with serial-to-parallel anomalies have been published elsewhere [10, 11, 15]. We now present results on coping with parallel-to-parallel anomalies of depth-first OR-tree searches based on the performance bounds derived in the last section. When comparing the efficiency between using k_1 and k_2 processors, $1 \leq k_1 < k_2$, a k_2/k_1-fold speedup (ratio of the number of iterations in the two cases in our model) is expected. However, simulations have shown that the speedup can be (a) less than one (called a *detrimental anomaly*) [6, 17, 9]; or (b) greater than k_2/k_1 (called an *acceleration anomaly*) [6, 9]; or (c) between one and k_2/k_1 (called a *deceleration anomaly*) [6, 22, 17, 9]. So far, all known results on parallel OR-tree searches showed a near-linear speedup for only a small number of processors.

Anomalies are studied with respect to the assumption that all idle processors are used to expand active subproblems. In fact, detrimental anomalies cannot happen if some processors can be kept idle in the presence of active subproblems. The number of processors to be kept idle is problem dependent and is very difficult to find without first solving the problem.

Some anomalies on parallel depth-first OR-tree searches are illustrated here. A single list of subproblems is assumed. The

behavior of using multiple lists is analogous to that of a centralized list. An example of an acceleration anomaly with an approximate depth-first or best-first search is shown in Figure 1a. When three processors are used, the optimal solution is found in the second iteration, and P_4 and P_5 are eliminated. If two processors are used, subtrees T_4 and T_5 have to be expanded. $T(2,0.1)/T(3,0.1)$ will be much larger than 3/2 if T_4 and T_5 are very large. Figure 1b illustrates a detrimental anomaly under an approximate best-first or depth-first search with ϵ=0.1. When two processors are used, $f(P_8)$, the optimal solution, is found in the fourth iteration. Assuming that the lower bounds of nodes in T_3 are between 8.2 and 9, all nodes in T_3 will be eliminated by lower-bound tests with P_8 since $[9/(1+\epsilon)]<8.2$. When three processors are used, P_3 is expanded in the third iteration. P_5, P_6, and P_7 are generated and will be selected in the next iteration. If T_3 is large, $T(2,\epsilon) < T(3,\epsilon)$ will occur. A detrimental anomaly may occur even when lower-bound tests are inactive and is illustrated in Figure 1c. A similar example can be derived for acceleration anomalies.

In the last section, we have derived the performance bounds with respect to depth-first OR-tree searches. From these results, we can develop the relative efficiency between using k_1 and k_2, $1<k_1<k_2$, processors. First, we derive a sufficient condition to assure the monotonic increase in computational efficiency with respect to the number of processors. To simplify the sufficient condition, the following bounds on $T_d(k,0)$ are used.

$$\frac{T_b'(1,0)}{k} \leqslant T_d(k,0) \leqslant \left\lfloor \frac{T_d(1,0)}{k} + (c+1)*h \right\rfloor.$$

Corollary 2:** Let $r_d' = T_b'(1,0)/T_d(1,0) \leqslant 1$. In a parallel depth-first search that satisfies the assumptions of Theorem 1, $T_d(k_2,0) \leqslant T_d(k_1,0)$ when

$$\frac{T_d(1,0)}{h} \geqslant \frac{(c+1)k_1k_2}{r_d'k_2 - k_1} \quad \text{and} \quad r_d' > \frac{k_1}{k_2}, \quad 1<k_1<k_2 \qquad (7)$$

where c is the number of the distinct incumbents obtained during the serial depth-first search.

From Corollary 2, we can conclude that the existence of parallel-to-parallel detrimental anomalies in depth-first searches depends on $T_b'(1,0)$, r_d', and c. If $r_d' \approx 1$, c is small, and $T_b'(1,0)$ is very large, then Eq. (7) will be satisfied. Our simulation results reveal that for some problems, such as the 0-1 knapsack and vertex-cover problems, $T_d(1,0)$ is close to $T_b'(1,0)$, hence $r_d' \approx 1$. Moreover, if the feasible-solution values must be integers, then c is often small. For this kind of problems, detrimental anomalies can be prevented for parallel depth-first searches when $T_b'(1,0)$ is large and k_2 is relatively small. However, the range of parallel processing within which no detrimental anomalies occur for depth-first searches is smaller than that for best-first searches [13].

From Theorem 1, we can also derive a necessary condition for acceleration anomalies with respect to k_1 and k_2 processors.

Corollary 3:** In a parallel depth-first search that satisfies the assumptions of Theorem 1, $T_d(k_1,0)/T_d(k_2,0) > k_2/k_1$ only if

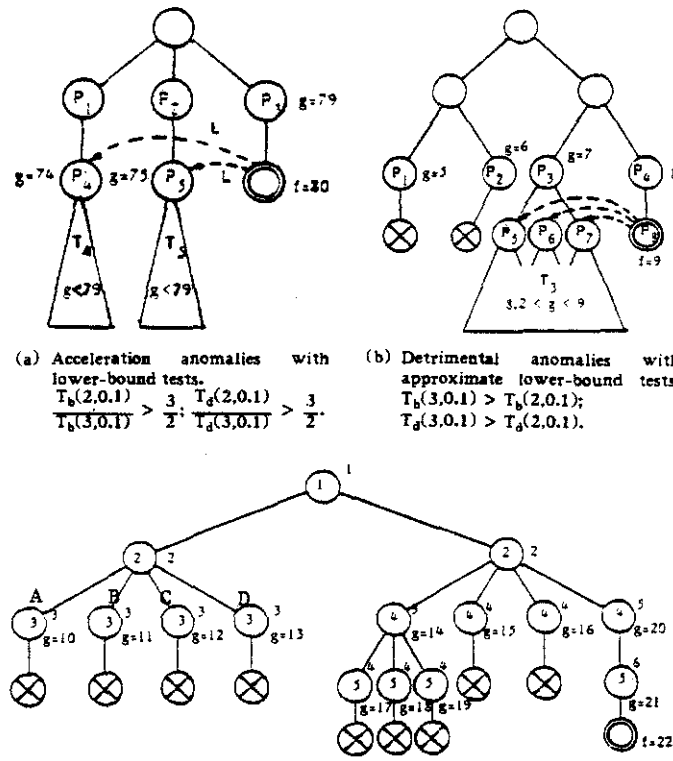
$$\left\lfloor T_d(1,0) - T_b'(1,0) \right\rfloor > \left\lfloor k_2 - 1 - (k_1-1)[(c+1)*h - c] \right\rfloor \quad 1<k_1<k_2 \qquad (8)$$

If all solutions are located at the bottommost level of the OR-tree, then the corresponding necessary condition is simplified by Corollary 1 as

$$\left\lfloor (c+1)*h - c \right\rfloor > \frac{k_2-1}{k_1-1} \qquad 1<k_1<k_2. \qquad (9)$$

Obviously, the necessary condition in Eq. (8) is readily satisfied, and $T_d(k_1,0)/T_d(k_2,0)$ may be much greater than k_2/k_1.

** The proof is omitted due to space limitation and can be found elsewhere [13, 16].



(a) Acceleration anomalies with lower-bound tests. $\frac{T_b(2,0.1)}{T_b(3,0.1)} > \frac{3}{2}$; $\frac{T_d(2,0.1)}{T_d(3,0.1)} > \frac{3}{2}$.

(b) Detrimental anomalies with approximate lower-bound tests. $T_b(3,0.1) > T_b(2,0.1)$; $T_d(3,0.1) > T_d(2,0.1)$.

(c) Detrimental anomalies without lower-bound tests in a depth-first or best-first search, T(4,0)=5, T(5,0)=6. (Number inside node is the evaluation order using four processors; number outside node is the evaluation order using five processors.)

Figure 1. Examples of anomalies.

Usually, if k_1 and k_2 are close to each other and h is large, then acceleration anomalies may occur quite often.

When a suboptimal solution is sought, the following corollary shows the required sufficient conditions.

Corollary 4:** In parallel depth-first searches that satisfy the assumptions of Theorem 1 with the exception that $\epsilon>0$, $T_d(k_2,\epsilon) \leqslant T_d(k_1,\epsilon)$ when

$$\frac{T_d(1,\epsilon)}{h} \geqslant \frac{(c+1)k_1k_2}{r_dk_2 - k_1} \text{ and } r_d > \frac{k_1}{k_2}, \quad 1<k_1<k_2 \qquad (10)$$

where $r_d = MINT_b(\epsilon)/T_d(1,\epsilon)$. $T_d(k_1,\epsilon)/T_d(k_2,\epsilon) > k_2/k_1$ when

$$\left\lfloor T_d(1,\epsilon) - MINT_b(\epsilon) \right\rfloor > \left\lfloor k_2 - 1 - (c+1)(k_1-1)*h \right\rfloor \qquad (11)$$

If all feasible solutions are located at the bottommost level of the OR-tree, the necessary condition to allow acceleration anomalies is the same as that stated in Eq. (9). Further, a weaker sufficient condition to eliminate detrimental anomalies can be derived from Corollary 1.

$$\frac{T_d(1,\epsilon)}{h} > \frac{(c+1)k_1k_2}{k_2 - k_1} \qquad (12)$$

3. ORDERED DEPTH-FIRST SEARCH FOR EVALUATING LOGIC PROGRAMS

In our previous paper [12] we have developed an optimal search strategy to evaluate logic programs modeled as AND/OR-trees using the heuristic information p(x), the success probability of a subgoal (or clause) x, and c(x), the estimated overhead of evaluating the subgoal (or clause). The heuristic information to guide the search is defined as follows.

$$\Phi_a(x) = \frac{p(x)}{c(x)} \qquad \begin{matrix}(x \text{ is descendant} \\ \text{of an OR—node})\end{matrix} \qquad (13)$$

$$\Phi_o(x) = \frac{1 - p(x)}{c(x)} \qquad \begin{matrix}(x \text{ is descendant} \\ \text{of an AND—node})\end{matrix} \qquad (14)$$

The logic program is first transformed from the AND/OR-tree representation into a two-level AND/OR-tree. The root of the transformed tree is an OR-node and represents the selection of clauses, and its descendants are AND-nodes and represent different solution trees in the logic program. The descendents of the OR-node are ordered according to decreasing values of Φ_o, and the descendants of the AND-nodes are ordered according to decreasing values of Φ_a.

Although the above strategy minimizes the expected search time, there are two implementation problems. First, the transformed AND/OR-tree significantly expands the number of nodes in the original AND/OR-tree. In fact, the number of potential solution trees is a hyper-exponential function of the height of the tree. To apply the above search strategy on the original AND/OR-tree, a global list is required to maintain the order of all possible solution trees, and the storage overhead is prohibitively large [12]. Second, if two solution trees T_1 and T_2 have nearly equal Φ_a or Φ_o, then exchanging the search order of T_1 and T_2 may not significantly improve the expected search overhead. As an example, suppose that the success probabilities and the estimated overheads of all solution trees rooted at a non-terminal node are uniformly distributed between 0.01 and 0.99 and 1 and 10 units of cost, respectively, and that there are a million possible solution trees from this node. Suppose further that two solution trees can be viewed as having nearly equal Φ_a or Φ_o if their difference is less than 0.001. Then, approximately, every thousand solution trees have nearly equal Φ_a or Φ_o. Obviously, it is unnecessary to store the exact order of all solution trees.

In this section we will address two problems. First, given an ordered depth-first search strategy and assuming that all sibling nodes in the AND/OR-tree are independent, what is the order to search the nodes in each level of the AND/OR-tree to minimize the expected search time? Second, for a logic program with shared variables and clauses, how should the subgoals and clauses be ordered to minimize the average search cost of a depth-first search?

3.1 Assumptions

In a logic program, if there are n clauses whose heads match (sub-)goal A, then they can be ordered according to the given heuristic values. Likewise, if there are m subgoals in the body of a clause B, $B :- B_1, \ldots, B_m$, then the m subgoals can also be ordered.

The assumptions made in the search strategy are described here.

(1) For a given representation of the AND/OR-tree, a depth-first search is used. When nodes in each level are ordered according to the heuristic values, the search is called an *ordered depth-first search*.

(2) A *producer-consumer* model is used to bind values to variables. A variable is a producer if it has not been bound to any value, otherwise, it is a consumer. For each variable not defined in the head, only its leftmost occurence can be the producer, as a depth-first search is used. All other occurences of this variable in this clause are consumers. For example, in the clause $A(x,y)$:-$B(x,z)C(z,y)D(x,y)$, variable z in subgoal B must be a producer, while variable z in subgoal C is a consumer. Depending on whether a variable defined in the head is a producer or a consumer, the variable in the corresponding subgoal will be a producer or a consumer. For example, if x is a producer in A, then x in B is a producer, while x in D is a consumer. We use a subscript '+' to indicate that the mode of a variable is a producer and a '−' to indicate that its mode is a consumer. As an example, $A(x_+,y_-) :- B(x_+,z_+)C(z_-,y_-)D(x_-,y_-)$. When a variable in a subgoal is a consumer, it is necessary to verify in this subgoal whether the subgoal is TRUE or FALSE for such a binding of value. In contrast, when a variable in a subgoal is a producer, it is necessary to find a binding of value to the variable such that this subgoal is TRUE.

(3) The probability of a subgoal to return TRUE and the average minimum overhead to determine whether a subgoal is TRUE or FALSE are independent of the bound values.

(4) The overhead to test whether a subgoal in a clause is TRUE or FALSE for a given binding of values to variables or to generate a binding of values to variables is assumed to be independent of other subgoals in this clause, provided that the modes of its variables are unchanged. Likewise, the overhead to verify the head of a clause is independent of other clauses with the same head when the modes of its variables are unchanged. These assumptions are valid when results in one subgoal or clause are passed to other subgoals or clauses through the binding of values to variables.

(5) The probability that a subgoal in a clause is TRUE for a given binding of values to variables is assumed to be independent of other subgoals in this clause. Similarly, the probability that the head of a clause is TRUE is independent of other clauses with the same head. These assumptions are not valid in general logic programs because subgoals have shared clauses and variables, but are made here to simplify the model.

3.2. Optimal Ordering of Depth-First Searches in AND/OR-Trees

In this section we discuss a special case in the optimal ordering of depth-first searches for AND/OR-trees, assuming that the success probabilities and expected overheads of all nodes are independent of each other, and that a node, once evaluated, will not be evaluated again. This special case exists in a logic program when it does not have any logic variables and shared clauses. For each node in the AND/OR-tree, suppose that it has n descendent nodes, then there are n! possible evaluation orders for a depth-first search. Our objective is to select the optimal order of descendents for each node in the AND/OR-tree such that the average overhead to verify the root to be TRUE or FALSE is minimized.

Various heuristic functions can be used to arrange the order of descendent nodes. Examples include the success probability, the lower bound on cost, and the number of immediate descendents. The following theorem shows that Φ_a and Φ_o (for AND-nodes and OR-nodes, respectively) are the heuristic functions to order the search such that the expected search cost is minimized.

Lemma 1: Suppose that node K is an OR-node (resp. AND-node) with n (resp. m) immediate descendent AND-nodes (resp. OR-node) ordered as $K_1, \ldots, K_n$ (resp. $K_1, \ldots, K_m$), and K_i is searched before K_{i+1} in a depth-first search. Let p_i and c_i be the success probability and search cost of node K_i, and $q_i=(1-p_i)$. If all p_is and c_is are independent of each other and $p_i/c_i < p_{i+1}/c_{i+1}$ (resp. $q_i/c_i < q_{i+1}/c_{i+1}$), $1 \leqslant i \leqslant n$, then the expected search cost can be reduced when K_{i+1} is searched before K_i.

Proof: Let C and C' be the expected costs of searching the descendents of node K in the order $K_1, \ldots, K_n$ and that in the order with K_i and K_{i+1} interchanged. Assume that node K is an AND node. Then

$$C = \sum_{k=1}^{n} \left(\prod_{j=1}^{k-1} q_j \right) \cdot c_k \qquad \text{and} \qquad (15)$$

$$C' = \sum_{k=1}^{i-1} \left(\prod_{j=1}^{k-1} q_j \right) \cdot c_k + \left(\prod_{j=1}^{i-1} q_j \right) \cdot (c_{i+1} + q_{i+1} c_i) + \sum_{k=i+1}^{n} \left(\prod_{j=1}^{k-1} q_j \right) \cdot c_k \qquad (16)$$

Subtracting Eq. (15) from (16) yields

$$C' - C = \left(\prod_{k=1}^{i-1} q_k \right) \cdot (p_i c_{i+1} - p_{i+1} c_i) > 0$$

The proof when node K is an OR-node is analogous. □

Some special cases of this ordering strategy have been observed by Simon and others [21, 3, 1].

Theorem 2: Assume that a depth-first search is used to search an AND/OR-tree, that the probabilities of success and search costs of all sibling nodes are independent of each other, and that a node, once evaluated, will not be evaluated again. The ordered sequence in which all OR-nodes, x_is, are ordered by decreasing $p(x_i)/c(x_i)$ and all AND-nodes, y_is, are ordered by decreasing $q(y_i)/c(y_i)$ will minimize the expected search cost over all possible ordered sequences, where $p(x)$, $q(x)$, and $c(x)$ are the success and failure probabilities and average search cost for node x.

Proof: Without loss of the generality, assume that the root (in Level 0) is an OR-node and that each OR-node (resp. AND-node) has n (resp. m) immediate descendent AND-nodes (resp. OR-nodes). For the n AND-nodes (resp. m OR-nodes), there are n! (resp. m!) possible oredered sequences, $s_1, \ldots, s_{n!}$ (resp. $s_1, \ldots, s_{m!}$). Let $c^j_{j,AND}$ (resp. $c^j_{j,OR}$) be the minimum expected cost of the j'th AND-node (resp. OR-node) in sequence s_i over all possible ordered sequences of descendents of this node. Let $c_{r,OR}$ be the minimum expected cost of a depth-first search of the root over all possible ordered depth-first searches of the given AND/OR-tree. Since the expected search cost of a node is the cost of searching the subtree rooted at this node to return TRUE or FALSE, it is independent of the search order of other sibling nodes. Hence, if all nodes in the k'th level have been ordered optimally, then this optimal order remains unchanged when determining the optimal order in levels smaller than k. That is, the principle of optimality is satisfied. The minimum expected cost of the root r can be found from a dynamic programming formulation.

$$c_{r,OR} = \min_{s_i \in \{s_1,\ldots,s_{n!}\}} \left\{ \sum_{j=1}^{n} \left[\prod_{k=1}^{j-1} q^i_k \right] * c^j_{j,AND} \right\} \quad \text{where} \tag{17}$$

$$c^j_{j,AND} = \min_{s_u \in \{s_1,\ldots,s_{m!}\}} \left\{ \sum_{v=1}^{m} \left[\prod_{w=1}^{v-1} p^u_w \right] c^u_{v,OR} \right\} \tag{18}$$

where p^i_k and q^i_k are, respectively, the success and failure probabilities of the k'th node in the i'th ordered sequence s_i. $c^u_{v,OR}$ can be evaluated in a similar fashion as in Eq. (17). Eq's (17) and (18) can be solved by a bottom-up evaluation.

For any nonterminal OR-node (resp. AND-node), K, since all its immediate descendents $K_1, \ldots, K_n$ (resp. $K_1, \ldots, K_m$) are independent of each other, then from Lemma 1 and applying adjacent pairwise interchanges, the optimal search order should satisfy $p(K_i)/c(K_i) > p(K_{i+1})/c(K_{i+1})$ (resp. $q(K_i)/c(K_i) > q(K_{i+1})/c(K_{i+1})$). □

The above ordering strategy only holds when all nodes are independent. In general, a logic program has shared variables and shared clauses. Hence, the subgoals and clauses have dependent search costs and success probabilities. Moreover, a subgoal may be searched more than once because a given binding of values to variables may succeed with this subgoal but fail with other subgoals. In the next section, we will discuss a heuristic method to find an efficient search order.

3.3. Ordered Depth-First Search of Logic Programs

To find an appropriate order of depth-first search in a logic program, the main problem is to develop a function to compute the expected search cost and success probability of a clause or a subgoal, assuming that the costs and success probabilities of all its immediate descendents in the AND/OR-tree representation are known. The difficulty lies in the shared variables and clauses in different subgoals of a logic program. The search cost of a subgoal may depend on the modes of its variables and cannot be evaluated as in Eq. (15). For a subgoal with a producer variable, it is necessary to generate one (or all) binding of value for the given variable; whereas a subgoal with a consumer variable has to test whether the given binding is TRUE. The latter cost is usually larger than the former one. The cost functions are more complicated when there are multiple variables. Here, a subgoal can have a combination of producer and consumer variables.

Owing to the distinction between producers and consumers and that a clause may be used with their variables set in different modes, the success probabilities and costs must be defined for all combinations of modes of variables. For example, there are four success probabilities and four expected search costs for clause with head A(x,y), namely, $p_A(x_+,y_+)$, $p_A(x_+,y_-)$, $p_A(x_-,y_+)$, $p_A(x_-,y_-)$, $c_A(x_+,y_+)$, $c_A(x_+,y_-)$, $c_A(x_-,y_+)$, and $c_A(x_-,y_-)$, where a subscript '+' indicates that a variable is a producer, and '−' indicates that it is a consumer. Let $\mathbf{L}$ be the set of variables in a subgoal, and $\mathbf{L}_+$ and $\mathbf{L}_-$ be the subsets of producer and consumer variables. For a clause with head $A(\mathbf{L}_+,\mathbf{L}_-)$, all variables in $\mathbf{L}_-$ have been bound (called a *binding-set*) before this clause is searched, whereas all variables in $\mathbf{L}_+$ must be bound after the subtree rooted at clause A has been searched.

In Figure 2 we have shown a Prolog program to query granddaughter(•,•). In Table 2 the average search costs for various modes of variables X and Y in granddaughter are shown. For different modes, the orders in which the depth-first search should be performed may be different. We have shown the order that minimizes the search cost for two of these combinations of modes. The structures for the other two combinations are different. The values in Table 2 illustrate that the difference in costs between the best and the worst orders can be a factor of one to seven.

For node $A(\mathbf{L})$, $p_A(\mathbf{L}_+,\mathbf{L}_-)$ is defined as the probability to successfully generate a binding-set of $\mathbf{L}_+$ under the condition that the given binding-set of $\mathbf{L}_-$ is TRUE, namely,

$$p_A(\mathbf{L}_+,\mathbf{L}_-) = p_A(\mathbf{L}_-)\, p_A(\mathbf{L}_+ \mid \mathbf{L}_-) \tag{19}$$

```
mother(theresa,martha).
mother(jane,martha).
mother(michael,mary).
mother(susan,jane).
mother(edward,jane).
wife(john,martha).
wife(paul,mary).
wife(michael,jane).
female(theresa).
female(susan).
female(X):-wife(_,X).
father(X,Y):-mother(X,Z),wife(Y,Z).
parent(X,Y):-mother(X,Y).
parent(X,Y):-father(X,Y).
grandparent(X,Y):-parent(Z,Y),parent(X,Z).
granddaughter(X,Y):-female(Y),grandparent(Y,X).
```

Figure 2. Minimum-cost Prolog program on family tree with granddaughter(−,+) or granddaughter(−,−) as the goal.

Modes of X,Y in granddaughter	*Minimum Avg. Cost*	*Maximum Avg. Cost*	*Mean Avg. Cost*	*Standard Deviation*
−,−	20.6	97.8	46.9	28.9
−,+	14.1	97.9	47.4	31.6
+,−	20.1	130.8	81.6	36.0
+,+	11.5	20.6	16.6	4.4

Table 2. Average Costs of evaluating granddaughter(X,Y) in Figure 2 for all combinations of bindings of variables and all possible solutions returned. (Each traversal of a subgoal or clause has unit cost. Each producer variable only produces one binding at a time.)

$p_A(L_-)$ is the probability that the binding-set of L_- on A is true. $p_A(L_+ | L_-)$ is defined as $n/(n+1)$, where n is expected number of binding-sets of L_+ in subgoal A for a given binding-set of L_-. In this case we are approximating the distribution on the number of distinct binding-sets of L_+ for a given binding-set of L_- as a geometric distribution with parameter p. For such a geometric distribution, its expected value is $p/(1-p)$, which implies that $p=n/(n+1)$. In the special case when all variables in L are producers, then $p_A(L_+) = m/(m+1)$, where m is the total number of generated binding-sets.

For $A(L_+,L_-)$, its expected cost, $c_A(L_+,L_-)$, is defined as the expected cost of generating a successful binding of variables in L_+, given the binding of variables in L_-. If all variables in L are consumers, then $c_A(L_-)$ is the expected cost of testing whether a binding-set is TURE.

For clarity, we illustrate a heuristic method to compute the various costs. In this method all probabilities are assumed to be independent. For a clause $A(x,y) :- B(x,z),C(z,y)$ with known costs and probabilities for subgoals B and C, the expected cost of A can be computed by modeling the test process as an absorbing Markov chain [26]. If *one solution is sought*, then the absorbing Markov chain in Figure 3a is used. The two sink nodes (s_0 and s_1) represent the states of success and failure. After a finite number of steps, the process must enter one of these absorbing states. To find the expected cost, we need to calculate the expected number of times that the process is in transient states s_2 and s_3. In this example P, the transition matrix, Q, one of its submatrices denoting the process in the transient states, and R, another submatrix denoting the transitions from the transient states to the absorbing states, are

$$P = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & q_2 & 0 & p_2 \\ p_3 & 0 & q_3 & 0 \end{vmatrix} = \begin{vmatrix} I & 0 \\ R & Q \end{vmatrix}; \quad Q = \begin{vmatrix} 0 & p_2 \\ q_3 & 0 \end{vmatrix}; \quad R = \begin{vmatrix} 0 & q_2 \\ p_3 & 0 \end{vmatrix} \qquad (20)$$

Let n_j be the expected total number of times that the process is in state s_j, and $M_i[n_j]$ be the mean of n_j when the chain is started in s_i. From the theory of absorbing Markov chains [26], $N = \{M_i[n_j]\} = (I - Q)^{-1}$. N is called the *fundamental matrix*. In our example

$$N = \begin{vmatrix} \frac{1}{1-p_2q_3} & \frac{p_2}{1-p_2q_3} \\ \frac{q_3}{1-p_2q_3} & \frac{1}{1-p_2q_3} \end{vmatrix} \qquad (21)$$

As a result, the expected cost is $(c_2+p_2c_3)/(1-p_2q_3)$, where c_i is the cost associated with state s_i. If B is searched before C, then A has expected cost

$$c_A(x_+,y_+) = \frac{c_B(x_+,z_+) + p_B(x_+,z_+)*c_C(z_-,y_+)}{1 - p_B(x_+,z_+)*q_C(z_-,y_+)} \qquad (22)$$

If subgoal C is searched before B, then A has expected cost

$$c_A'(x_+,y_+) = \frac{c_C(z_+,y_+) + p_C(z_+,y_+)*c_B(x_+,z_-)}{1 - p_C(z_+,y_+)*q_B(x_+,z_-)} \qquad (23)$$

Comparing c_A and c_A', the order with the smaller cost is used. Expected costs of clause A with variables in other modes can be computed similarly.

When *all solutions* in a subgoal have to be found, the process can also be modeled as an absorbing Markov chain. Figure 3b shows the absorbing Markov chain for the above example.

To compute the success probability of a clause, if $b_{i,j}$ is the probability that the process starting in transient state s_i ends up in absorbing state s_j, then from the theory of absorbing Markov chains, $\{b_{i,j}\} = B = N \times R$. In our example, $b_{2,0} = p_2p_3/(1-p_2q_3)$, and $b_{2,1} = q_2/(1-p_2q_3)$. If subgoal B is searched first, then the success probability of node A is

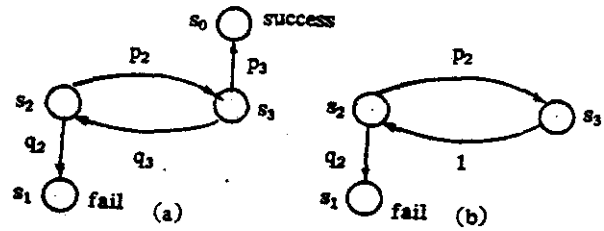


Figure 3. Example to compute the search cost and probability using an absorbing Markov chain.

$$p_A(x_+,y_+) = \frac{p_B(x_+,z_+)p_C(z_-,y_+)}{1 - p_B(x_+,z_+)q_C(z_-,y_+)} \qquad (24)$$

In general, if a subgoal has k variables, then 2^k combinations of probabilities and costs corresponding to all combinations of modes of the variables have to be found.

The above example illustrates the use of an absorbing Markov chain to order the search of descendents of an AND-node, which represents the evaluation of a clause. In contrast, to order the descendents of an OR-node, which represents the selection of multiple clauses with the same head, it is observed that once a descendent of an OR-node has been searched for a given binding-set, it will not be searched again. Unlike descendents of AND-nodes, there is no backtracking involved for a given binding-set. According to Theorem 2, descendents of an OR-node should, therefore, be searched in decreasing order of ratios of success probability to cost. The cost and probability of an OR-node can be computed in a similar fashion as in Eq. (15) when it has at least one consumer variable. When all its variables are producers, the average cost is taken as the average cost of each of its descendents weighted by the fraction of the total number of binding-sets that can be generated.

The basic idea in a systematic method to determinate an appropriate ordering of the subgoals and clauses is to associate with each subgoal and clause a table of the expected costs and probabilities for all combinations of modes of variables, and to use the appropriate costs and probabilities depending on the modes set for the variables. The best order with the minimum expected cost is chosen from all possible permutations of descendents. The number of permutations may be large. In this case heuristic information, such as the number of variables in a subgoal, can be used to eliminate inefficient candidate permutations. Note that the cost of each node in the AND/OR-tree representation depends only on the costs and probabilities of its descendents, provided that the descendents only depends on each other through shared variables. Here, the selection of the best order in a given level does not influence the computation of costs in levels above. That is, the computation of the minimum cost satisfies the principle of optimality, and the optimal order can be found by dynamic programming. In practice, subgoals are generally dependent on each other through shared clauses, which results in over-estimation of the costs. The proposed scheme is still applicable as a heuristic method to arrange the order in the search process. Statistic sampling has to be used to estimate the cost and probability of a node after the order of its descendents is determined. This reduces the accumulation of errors as nodes in higher levels of the AND/OR-tree are ordered.

A final point on the ordering of nodes in the AND/OR-tree representation of logic programs is that different orders may be found depending on the modes of the variables. Either an 'average' order may be used or multiple program statements may be generated for different cases to reflect the preferred order.

4. CONCLUSIONS

In this paper we have studied the computational efficiency of parallel and ordered depth-first searches to solve optimization

and decision problems. The performance bounds and conditions to cope with anomalies in searching optimization problems represented as OR-trees have been derived and verified by simulations. Speedups have been found to be related to the problem complexity and the number of incumbents obtained during the search process. For a problem with a high complexity and a small number of incumbents, such as integer programming problems, a near-linear speedups can be achieved with respect to a large number of processors.

An ordered depth-first search strategy has been studied with respect to decision problems represented as AND/OR-trees. When the success probabilities and costs of sibling nodes are independent of each other, and a node, once searched, will not be searched again, the sibling nodes should be ordered according to ratios of probability and cost to minimize the expected total search cost. Due to shared clauses and variables in a Prolog program and that backtracking is allowed, it is difficult to find the optimal depth-first search order. An absorbing Markov chain to model the effects of backtracking and a dynamic programming method to order the search have been developed.

ACKNOWLEDGEMENTS

The authors would like to thank Mr. Mark Gooley for generating the results in Figure 2 and Table 2.

REFERENCES

[1] J. Barnett, "Optimal Searching from AND Nodes," *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 786-788, William Kaufman, Inc., Los Altos, CA, 1983.

[2] A. Ciepielewski and S. Haridi, "Execution of Bagof on the OR-Parallel Token Machine," *Proc. Int'l Conf. Fifth Generation Computer Systems*, pp. 551-560, ICOT and North-Holland, 1984.

[3] M. Garey, "Optimal Task Sequencing with Precedence Constraints," *Discrete Mathematics*, vol. 4, no. 1, pp. 37-56, 1973.

[4] T. Ibaraki, "Theoretical Comparisons of Search Strategies in Branch-and-Bound Algorithms," *Int'l J. of Computer and Information Sciences*, vol. 5, no. 4, pp. 315-343, Plenum Press, 1976.

[5] T. Ibaraki, "Computational Efficiency of Approximate Branch-and-Bound Algorithms," *Math. of Oper. Research*, vol. 1, no. 3, pp. 287-298, Inst. of Management Sciences, 1976.

[6] M. Imai and T. Fukumura, "A Parallelized Branch-and-Bound Algorithm Implementation and Efficiency," *Systems, Computers, Controls*, vol. 10, no. 3, pp. 62-70, Scripta Publishing, June 1979.

[7] R. Kowalski, *Logic for Problem Solving*, North-Holland, 1979.

[8] V. Kumar and L. N. Kanal, "A General Branch and Bound Formulation for Understanding and Synthesizing And/Or Tree Search Procedures," *Artificial Intelligence*, vol. 21, no. 1-2, pp. 179-198, North-Holland, 1983.

[9] T. H. Lai and S. Sahni, "Anomalies in Parallel Branch-and-Bound Algorithms," *Comm. of the ACM*, vol. 27, no. 6, pp. 594-602, ACM, June 1984.

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

[11] G.-J. Li and B. W. Wah, "How to Cope with Anomalies in Parallel Approximate Branch-and-Bound Algorithms," *Proc. National Conf. on Artificial Intelligence*, pp. 212-215, AAAI, 1984.

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

[13] G.-J. Li, *Parallel Processing of Combinatorial Search Problems*, Ph.D. Dissertation, Purdue University, W. Lafayette, IN, Dec. 1985.

[14] G.-J. Li and B. W. Wah, "Optimal Granularity of Parallel Evaluation of AND-Trees," *Proc. 1986 Fall Joint Computer Conference*, ACM-IEEE, Nov. 1986.

[15] G.-J. Li and B. W. Wah, "Coping with Anomalies in Parallel Branch-and-Bound Algorithms," *Trans. on Computers*, vol. C-35, no. 6, pp. 568-573, IEEE, June 1986.

[16] G.-J. Li and B. W. Wah, *Computational Efficiency of Combinatorial OR-Tree Searches*, to appear in IEEE Trans. on Software Engineering, 1986.

[17] J. Mohan, "Experience with Two Parallel Programs Solving the Traveling-Salesman Problem," *Proc. 1983 Int'l Conf. on Parallel Processing*, pp. 191-193, 1983.

[18] T. Moto-oka, H. Tanaka, H. Aida, K. Hirata, and T. Maruyama, "The Architecture of a Parallel Inference Engine (PIE)," *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pp. 479-488, ICOT and North-Holland, 1984.

[19] D. Nau, V. Kumar, and L. Kanal, "General Branch and Bound and its Relation to A* and AO*," *Artificial Intelligence*, vol. 23, no. 1, pp. 29-58, North-Holland, 1984.

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

[21] H. A. Simon and J. B. Kadane, "Optimal Problem-Solving Search: All-or-None Solutions," *Artificial Intelligence*, vol. 6, no. 3, pp. 235-247, North-Holland, 1975.

[22] B. W. Wah and Y. W. E. Ma, "MANIP--A Parallel Computer System for Implementing Branch and Bound Algorithms," *Proc. 8th Annual Symp. on Computer Architecture*, pp. 239-262, ACM, 1981.

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

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

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

[26] J. G. Kemeny and J. L. Snell, *Finite Markov Chains*, D. Van Nostrand Company, Inc., New York, NY, 1965.