Efficient Branch-and-Bound Algorithms on a Two-Level Memory System

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

Abstract—**In this paper, we have investigated the efficient evaluation of branch-and-bound algorithms in a system with a two-level memory hierarchy. An efficient implementation depends on the disparities in the numbers of subproblems expanded between the depth-first and best-first searches as well as the relative speeds of the main and secondary memories. A best-first search should be used when it expands a much smaller number of subproblems than that of a depth-first search, and the secondary memory is relatively fast. In contrast, a depth-first search should be used when the number of expanded subproblems is close to that of a best-first search. The choice is not as clear for cases in between. The Iterative Deepening A* (IDA*) algorithm has been shown to be asymptotically optimal in space, time, and cost. However, for the conditions that we have assumed, IDA* does not result in the optimal space–time tradeoff for minimizing the completion time. In this paper, we study the space–time tradeoff by proposing and analyzing two strategies: a specialized virtual-memory system that matches the architectural design with the characteristics of the existing algorithm, and a modified branch-and-bound algorithm that can be tuned to the characteristic of the problem and the architecture. The latter strategy illustrates that designing a better algorithm is sometimes more effective than tuning the architecture alone. Guidelines have also been developed to select appropriate *a priori* values for the parameters of the modified B&B algorithm.**

Index Terms—**Best-first search, branch-and-bound algorithm, disk overhead, modified branch-and-bound algorithm, two-level memory hierarchy, virtual-memory.**

I. Introduction

THE search for solutions in a combinatorially large problem space is important in artificial intelligence and operations research. Search problems may be classified as either decision or optimization problems [11]. In a decision problem, one attempts to determine the existence of at least one solution that satisfies a given set of constraints. Examples include theorem-proving, expert systems, and some permutation problems. An optimization problem is characterized by an objective function to be minimized or maximized and a set of constraints to be satisfied. Examples include the traveling-salesman, job-shop-scheduling, knapsack, vertex-cover, and integer-programming problems.

Manuscript received January 14, 1986; revised October 28, 1986. This work was supported by the National Science Foundation under Grants ECS81-05968 and MIPS85-19649.

C. F. Yu is with the Intel Corporation, M.S. SC9-22, 2250 Mission College Blvd., Santa Clara, CA 95052.

B. W. Wah is with the National Science Foundation, 1800 G Street, NW, Washington, DC 20050, on leave from the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

IEEE Log Number 8822456.

A general technique for solving combinatorial-search problems is the *branch-and-bound (B&B) algorithm* [20]. A B&B algorithm is a partitioning algorithm that decomposes a problem into smaller subproblems and repeatedly decomposes them until infeasibility is proved or a solution is found. Many theoretical properties of serial B&B algorithms have been developed [13]–[16], [18], [27], [41]. It has also been recognized that the B&B algorithm is a generalization of many heuristic search procedures [19] such as the A* [26], AO* [22], SSS* [32], B* [3], alpha-beta [17], and dynamic programming [6] algorithms. Parallel computer architectures for evaluating B&B algorithms have also been studied [35].

B&B algorithms are characterized by four constituents: branching rule(s), selection rule(s), elimination rule(s), and termination condition(s). The first two rules are used to decompose problems into simpler subproblems and appropriately order the search. The last two rules are used to eliminate generated subproblems that are not better than the ones already known.

In a B&B algorithm, let P_i be a subproblem, and $f(P_i)$ be the value of the best solution obtained by evaluating all subproblems decomposable from P_i. Each subproblem is characterized by a value that is computed from a lower-bound function g. The lower-bound function satisfies the following properties:

1) g is a lower-bound estimate of f;
2) g is exact when P_i is feasible;
3) lower bounds of descendant nodes always increase.

A *lower bound* is calculated for a subproblem when it is created. If a subproblem is a feasible solution with the best objective-function value so far, then the solution value becomes the *incumbent* z. In minimization problems, if the lower bound of P_i exceeds the incumbent, then P_i can be pruned because it will not lead to a better solution value than the incumbent. In other words, P_i is terminated during the computation if

$$g(P_i) \geq z. \tag{1}$$

The decomposition process continues until all subproblems are either expanded or eliminated.

The above elimination rule for obtaining an exact optimal solution can be relaxed to obtain a suboptimal solution with a guaranteed accuracy [20]. Suppose it were decided that a deviation of 10 percent from the optimum was tolerable. If a feasible solution of 150 is obtained,

then all subproblems with lower bounds of 136.4 [or $150/(1 + 0.1)$] or more can be terminated since they cannot lead to a solution that deviates by more than 10 percent from 150. This technique significantly reduces the amount of intermediate storage and time needed to arrive at a suboptimal solution. Define an *allowance function* $\epsilon(z)$: $\boldsymbol{R} \rightarrow \boldsymbol{R}$ (set of reals) such that P_i is terminated if

$$g(P_i) \geq z - \epsilon(z). \tag{2}$$

The final incumbent z_F obtained by the modified lower-bound test is related to the optimal solution value z_0 by [13]

$$\left(z_F - \epsilon(z_F)\right) \leq z_0 \leq z_F. \tag{3}$$

Examples of often used allowance functions are

$$\epsilon(z) = \epsilon \qquad \epsilon \geq 0 \text{ (absolute error deviation), and} \tag{4}$$

$$\epsilon(z) = \frac{\epsilon z}{1 + \epsilon} \qquad \epsilon \geq 0,\ z \geq 0 \text{ (relative error deviation)} \tag{5}$$

The selection rule examines the list of active subproblems and selects one for expansion. If the list is maintained in a first-in/first-out order, then the algorithm is called a *breadth-first search*. If the list is maintained in a last-in/first-out order, then the algorithm is called a *depth-first search*. Lastly, if the list is maintained in an increasing order of lower bounds, then the algorithm is called a *best-first search*.

The iterative deepening A* (IDA*) algorithm [42], [43] starts from the root and performs a depth-first search using a heuristically assigned value as the initial incumbent. This depth-first search is repeated with successively increasing estimates for the initial incumbent until the optimal solution node is expanded. The first initial incumbent is taken to be the lower bound of the root. After each iteration, the next estimate of the initial incumbent is the smallest of the lower bounds that exceed the current estimate of the incumbent.

The B&B algorithm is summarized as follows:

```
procedure Branch.and.Bound;
[  incumbent = ∞;
   list of subproblems = {P0};
   while list of subproblems ≠ ∅ do [
      apply selection to rule to list of subproblems;
      expand selected subproblem by the branching rule;
      for P ∈ {child subproblems generated} do [
         if P is a feasible solution then [
            update incumbent;
            apply elimination rule ]
         else if ( g(P) < (z − ε(z))) then
            insert P into the list of subproblems
      ]
   ]
]
```

To illustrate the B&B algorithm, the evaluation of an integer-programming problem [24] is shown here. Integer-programming problems may be expressed as a constrained optimization.

$$\begin{aligned} &\text{Minimize} && CX \\ &\text{subject to} && AX \geq B \\ & && X^T = (x_1, x_2, \cdots, x_n); \\ & && x_i\text{: nonnegative integer, } i = 1, 2, \cdots, n; \\ & && A, B, \text{ and } C \text{ are constant matrices.} \end{aligned} \tag{6}$$

These problems differ from ordinary linear-programming problems in that the variables are restricted to nonnegative integer values.

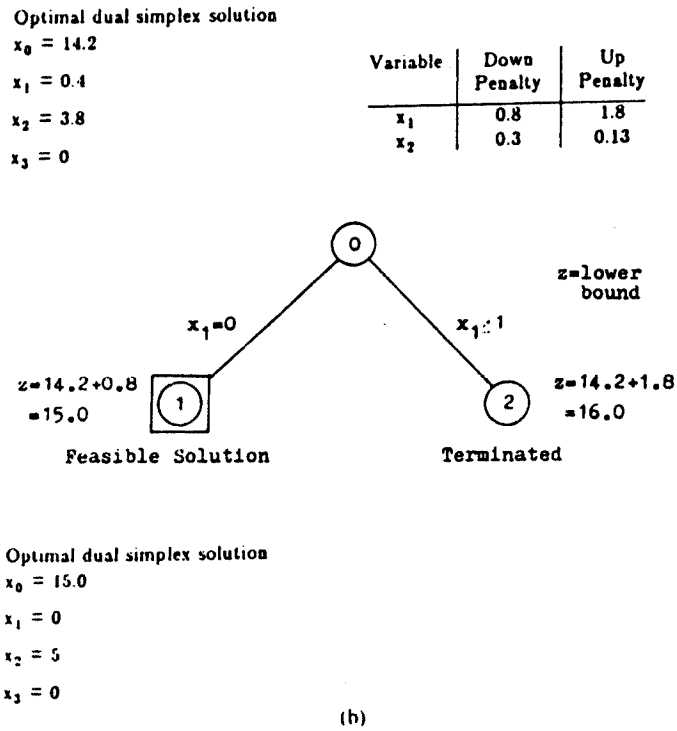
One approach to solve this problem is to first apply the dual simplex method to a subproblem and solve it as a linear program. If the optimal solution is integral, then a feasible solution has been generated; otherwise, create two new subproblems as follows. Choose a variable in the subproblem that has a nonintegral value (say $x_i = 4.4$) and restrict that variable to the next smaller integral value for one subproblem ($x_i \leq \lfloor 4.4 \rfloor$ or $x_i \leq 4$) and to the next larger integral value ($x_i \geq \lfloor 4.4 \rfloor$ or $x_i \geq 5$) for the other. The variable chosen is the one with the greatest up or down penalty. The up penalty for a variable x_i, having a value of a_i, is the estimate of the amount by which the solution to the current subproblem would increase if the integral constraint $x_i \geq \lfloor a_i \rfloor$ was introduced. The down penalty is similar, except that it is associated with the constraint $x_i \leq \lfloor a_i \rfloor$. The lower bound of a new subproblem is the sum of the optimal simplex solution and the associated penalty. This entire process is repeated on the new subproblems until either a feasible integral solution is found or all subproblems have been expanded.

Fig. 1(b) shows the B&B tree for the problem in Fig. 1(a). The dual simplex method gives an optimal solution of 14.2 for the original problem. Since the variables are not integral, a feasible solution has not been generated. Up and down penalties are calculated for the variables. x_1 has the greatest penalty ($U = 1.8$). Two new subproblems are then created, one with $x_1 = 0$, and the other with $x_1 \geq 1$. The lower bounds are calculated as shown in Fig. 1(b). The dual simplex method is applied again to the subproblem with the smallest lower bound, and a feasible solution is generated in which all variables are integral. This constitutes an optimal solution since the lower bound of the remaining subproblem is greater.

Among the alternative search strategies, depth-first search, breadth-first search, and IDA require a small memory space and can be evaluated entirely in the main memory. In a depth-first search, the list of subproblems is stored in a last-in/first-out stack with a maximum size equal to the height of the search tree. For many search problems, this height is equal to the number of variables in the problem and is reasonably small. A breadth-first search has a similar behavior. On the other hand, a best-first search expands the minimum number of subproblems

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

(a)



(b)

Fig. 1. (a) An example of an integer-programming problem and (b) the corresponding B&B solution.

if no node other than an optimal-solution node has a lower-bound value equal to the optimal-solution value [20], [21]. Hence a best-first search will have the best time-efficiency if the main memory is large enough to accommodate all its active subproblems. However, analytical and simulation studies have shown that the average maximum memory space required to obtain an optimal solution for many problems by a best-first search is exponential with a sublinear exponent [34], [36], [40] unless the lower-bound function is very accurate in leading to the optimal solution. In many computers, the main memory is limited in size, and a slower secondary memory will have to be used with the main memory to solve any nontrivial problem by a best-first search. Although IDA* is asymptotically optimal when the problem size is infinite, for finite problem sizes it expands at best a constant multiple of the number of nodes expanded in a best-first search [42]. IDA* performs redundant node expansions because it uses a stack of constant size. If memory space is available in main and secondary memory, a better space-time tradeoff can be made by storing the nodes and not regenerating them as well as overlapping the accesses to main and secondary memory such that the average access time is close to that for main memory. In this case, the time required to expand a subproblem by a best-first search may exceed that by a depth-first search, and the latter strategy may have a better time-efficiency. Although IDA* is asymptotically optimal when the problem size is infinite, for finite problem sizes it expands at best a constant multiple of the number of nodes expanded in a best-first search [42]. IDA* performs redundant node expansions because it uses a stack of constant size. If memory space is available in main and secondary memory, a better space-time tradeoff can be made by storing the nodes and not regenerating them as well as overlapping the accesses to main and secondary memory such that the average access time is close to that for main memory. In this paper, we seek to determine the most efficient search strategy for a two-level memory hierarchy.

Studies have been carried out on non-search algorithms in a computer with a two-level memory hierarchy managed by a paged virtual-memory system. These include rules for enhancing locality [25], automatic restructuring of programs [12], [7]–[9], [2], matrix manipulation [23], [10], sorting [4], [1], and database operations [30].

In the remainder of this paper, we will present alternative search strategies that are useful in a *single-user computer system*. In such a system, the time-efficiency may be characterized by the expected completion time of the algorithm. In Section II, we will present analytical and simulation results on the number of subproblems expanded under the best-first and depth-first strategies. In Sections III and IV, we will present the design of a specialized virtual-memory system for the best-first search and a modified B&B algorithm [39], which is a compromise between the depth-first and best-first searches. The modified algorithm illustrates that *designing new architectures to fit an existing algorithm may be less beneficial than designing a better algorithm to fit an existing architecture*.

Both of these alternatives are of limited value in multiuser systems. It is extremely difficult to modify the virtual-memory system without adversely affecting other users in the system. Moreover, it is very difficult to determine a suitable metric to characterize the time efficiency of an algorithm to the degree that it may be used as an optimization measure. The completion time cannot be used to characterize the efficiency of the algorithm as it will depend not only on the algorithm itself but also on the system load during its execution. Other possible metrics such as the processing time and the sum of the processing and secondary-storage times do not adequately describe the tradeoff and the overlap between computations and secondary-memory accesses.

II. B&B Algorithms in a Two-Level Memory Hierarchy

To decide between using the best-first and depth-first strategies in a system with a two-level memory hierarchy, it is necessary to compare the relative numbers of subproblems expanded (Section II-A) and the relative speeds of the main and secondary memories (Section II-B).

A. Number of Expanded Subproblems

An analytical model of the search strategies depends on

the accuracy of the lower-bound function that predicts the best solution derivable from a given node. To compare the performance of search strategies for all possible distributions of the accuracy would be a nontrivial, if not impossible, problem. The effects of the accuracy of the lower-bound function on the average performance of the A^*-algorithm (best-first search) and informed backtracking (depth-first search) have been studied for decision problems [28]. Other models of the cutoff mechanism which do not explicitly consider the accuracy of the lower bounds have also been used to study the average performance of best-first [31] and depth-first [31, 33] searches. A mathematically tractable solution that assumes a constant distribution for the accuracy has been derived and published before [36]. The results of this derivation are summarized here.

Suppose that the state-space tree is a complete s-way tree of height h and that solution nodes exist at the leaves. It was assumed that all solution values were independently and identically distributed (iid) random variables with a truncated exponential distribution function. An approximate value of the expected number of nodes expanded was derived as a function of the parameters of the distribution function, the shape and size of the search tree, and the search strategy. The expected number of nodes expanded by a best-first search was found to be always less than that by a depth-first search. However, the difference was a function of the accuracy of the lower-bound function in predicting the solution value. When the lower-bound function is very inaccurate, depth-first, and best-first searches both perform poorly and expand a large portion of the state-space tree. In contrast, when the lower-bound function is very accurate, both depth-first and best-first searches perform equally well and expand a small portion of the state-space tree. Between these two extremes, a depth-first search always expands many more nodes than a best-first search.

The above behavior can be illustrated by the 0/1 knapsack, vertex-cover, and integer-programming problems. The lower-bound functions for the 0/1 knapsack and vertex-cover problems are greedy algorithms and are very accurate in pruning unnecessary expansions. In contrast, the integer-programming problem uses a lower-bound function that is a linear program with the integrality constraints removed for all unassigned variables. This function will generally be less accurate. Table I shows some results from our simulations with these problems. For the knapsack and vertex-cover problems, the number of subproblems expanded in a depth-first search was almost identical to that in a best-first search. For integer-programming problems, the ratio of the number of subproblems expanded in a depth-first search to that in a best-first search varies between 1.4 and 17.6.

TABLE I
NUMBER OF NODES EXPANDED IN BEST-FIRST (BFS) AND DEPTH-FIRST (DFS) SEARCHES FOR 30-OBJECT 0/1 KNAPSACK, 80-NODE VERTEX-COVER, AND 20-VARIABLE, 20-CONSTRAINT INTEGER-PROGRAMMING PROBLEMS. (PARAMETERS FOR THESE PROBLEMS WERE GENERATED RANDOMLY.)

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

B. Effects of Secondary Memory on the Total Overhead

A simplified analysis of the performance of a best-first search will indicate the appropriate search strategy to use. Suppose that the best-first search is implemented on a paged virtual-memory system with a list of subproblems and a priority queue[1] of pointers to these subproblems to maintain the required ordering. Subproblems are not ordered by lower bounds in the list. In each iteration, the subproblem with the minimum lower bound is found from the priority queue and deleted from the list, while newly generated subproblems are inserted into the list and the priority queue is updated. Let n_p subproblems be the page size, and n_m pages be the maximum amount of main memory allocated to the subproblem list. Also, assume that n_{bfs} (resp. n_{dfs}) subproblems are expanded in the best-first (resp. depth-first) search and that each subproblem expansion generates s subproblems. Let r_{bfs} and w_{dfs} be the expected number of page reads and page writes from and to the secondary memory, respectively. Since a pointer node is usually much smaller than a subproblem, the heap is assumed to be resident in the main memory, and the paging overhead due to the pointer heap will be ignored.

In the simplified analysis, it is assumed that the initial incumbent is set at $-\infty$, and that the first feasible solution generated becomes the optimal solution, hence subproblems generated are not pruned.[2] Manipulation of the list of subproblems will incur no page reads or writes if n_{bfs} does not exceed n_t, where

$$n_t = \left\lfloor \frac{n_m n_p - 1}{s - 1} \right\rfloor. \tag{7}$$

n_t measures the number of iterations that subproblems can be expanded without exceeding the size of the main memory. Initially, the memory space available is $(n_m n_p - 1)$ subproblems with the root of the B&B tree resident in the main memory. In each iteration, one subproblem is removed from the main memory and s subproblems are gen-

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

[2] In reality, when the first feasible solution is generated in a best first search, the optimal solution is close to be found. The initial incumbent used is problem dependent. In integer programming problems, the initial incumbent is set at $-\infty$. For other problems, a better initial incumbent can be used.

erated, resulting in an increase of $(s-1)$ subproblems in the main memory.

Since subproblems in the subproblem list are not ordered, it is assumed that the subproblem chosen for expansion is equally likely to be in any of the pages in use by the list. The actual numbers of page reads and writes will be less than the corresponding numbers when no subproblems are terminated prior to the generation of the optimal solution. Hence

$$r_{bfs} \le \sum_{i=n_t+1}^{n_{bfs}} \left(1 - \frac{n_m}{\left\lceil \frac{i(s-1)+1}{n_p} \right\rceil}\right) \qquad n_{bfs} > n_t \tag{8}$$

$$w_{bfs} \le \left\lceil \frac{n_{bfs}(s-1)+1}{n_p} \right\rceil - n_m \qquad n_{bfs} > n_t. \tag{9}$$

In (8), index i represents the current number of subproblems expanded. Each term in the summation is the probability that the problem chosen for expansion is residing in the secondary memory.

Page writes may be overlapped with subproblem expansions when sufficient buffers are available because the B&B process can continue once the page has been copied into a buffer. However, processing will have to be delayed in a page read until the page has been read from the secondary memory. Let h_c be the CPU time required to expand a subproblem, which is the same for both best-first and depth-first searches. Further, let h_d be the average page-read time, and Ξ be the ratio of the number of subproblems expanded in a best-first search to that in a depth-first search ($=n_{bfs}/n_{dfs}$). D_{bfs}, the overhead incurred by a best-first search, can be approximated as the summation of the total page-read times and the maximum of the CPU times and the page write times.

$$D_{bfs} = r_{bfs}h_d + \max\left(w_{bfs}h_d, n_{bfs}h_c\right) \tag{10}$$

From (8), $r_{bfs} \approx w_{bfs} \approx n_{bfs}$ when $n_{bfs} >> n_t$. In this case, a best-first search will be more efficient than a depth-first search if Ξ is less than Ξ_e, where

$$\Xi_e = \frac{h_c}{\max\left(h_c, h_d/n_p\right) + h_d/n_p} \tag{11}$$

and h_d/n_p is the page-read overhead for each subproblem expanded.

In practice, due to the many simplifications made in this analysis, a best-first search should be chosen only when $\Xi << \Xi_e$. Similarly, a depth-first search should be used when $\Xi >> \Xi_e$. For problems characterized by intermediate values of Ξ, the choice between the depth-first and best-first searches is less clear. Either a depth-first or a best-first search may be used, although neither is particularly suitable. In Section IV, we study a modified B&B algorithm that is a hybrid of best-first and depth-first searches and is more efficient than both when Ξ is close to Ξ_e.

In short, the difference between the number of subproblems expanded in the best-first and depth-first searches is problem dependent and varies considerably between problems. This suggests that no one strategy will be optimal for all problems in a two-level memory hierarchy and that the strategy should be chosen according to the predicted difference. A depth-first search should be used when there is a small difference between the number of subproblems expanded under the depth-first and best-first searches, such as the case in the 0/1 knapsack and vertex-cover problems. When this difference is large, a depth-first search will still be more efficient if the secondary memory is slow. On the other hand, a best-first search should be used when a depth-first search expands many more subproblems than the best-first search. When this disparity is moderately large, a best-first search should still be used if the computer has a fast secondary memory.

III. A Specialized Virtual-Memory System for Best-First Search

In this section, we show an approach that tailors the operating system to the characteristics of the best-first search. The proposed virtual-memory system is depicted in Fig. 2. Subproblems in the secondary storage are organized in a B$^+$-tree.[3] Each leaf of the tree is a page and contains subproblems. The B$^+$-tree organizes subproblems to be expanded in the order dictated by the best-first search. Since nonterminal nodes of the tree are pointer nodes that are much smaller than leaf nodes, a substantial portion of the pointer tree may be kept in the main memory. This reduces the number of secondary-memory accesses required to access a subproblem in the secondary storage.

The main memory contains a partial list of subproblems (*m-list*) and a heap of pointers to subproblems in the *m*-list. Newly generated subproblems are inserted into the *m*-list. When the *m*-list is full, subproblems are moved to the B$^+$-tree pages.

In a best-first search, the subproblem with the smallest lower bound is always expanded in each iteration. This requires the comparison of the subproblems with the smallest lower bound in the *m*-list and that in the B$^+$-tree. The process is prohibitively expensive if the first B$^+$-tree page resides in the secondary memory. Hence the first B$^+$-tree page should be kept in a fast buffer (*s*-buffer) that is partially in the main memory.

In inserting a subproblem into a normal B$^+$-tree, the appropriate page is read into the main memory, the subproblem is inserted into the page image in main memory, and the page image is written back to disk. A more efficient scheme is to set the block size to an integral number of disk sectors which is greater than or equal to the size

[3] A B-tree of order b is a search tree that is either empty or satisfies the following properties [5]: 1) the root node has at least two children; 2) each node contains at least b keys and $b+1$ pointers; and 3) each node contains at most $2b$ keys and $2b+1$ pointers. A B$^+$-tree is variant of the B-tree in which all records reside in the leaves. The upper levels are organized as a B-tree and serve as an index to locate a record. The leaf nodes are linked from left to right for easy sequential processing.

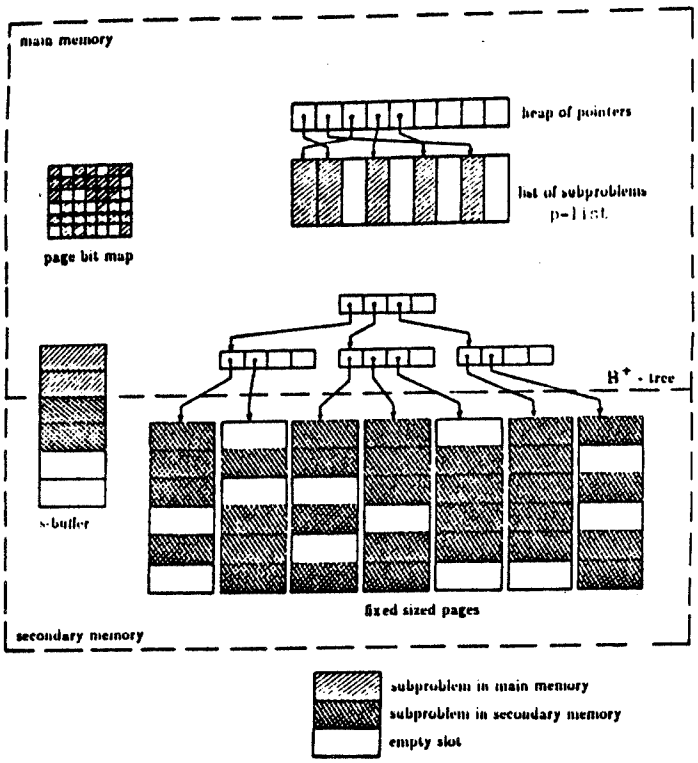


Fig. 2. Data structure in a modified virtual-memory system for supporting the best-first search.

of a subproblem and to use a bit map in main memory to show the status of all page blocks. Inserting a subproblem into a page, therefore, consists of searching for an empty block in the bit map and writing the block when the disk head is properly positioned.

The design of the virtual-memory system involves the selection of a replacement algorithm and the page size [38]. Unlike conventional best-first searches, the B^+-tree and a good replacement algorithm allow future disk accesses to be predicted, provided that there are enough disk buffers. The overhead of execution in a single-user environment can be approximated by the maximum of the computation and disk times. Since subproblems are expanded in a similar fashion as conventional best-first searches, the modified virtual-memory system has little effects on the computational time, and the objective of design becomes the minimization of the paging overhead. An efficient replacement algorithm should maximize the number of subproblems inserted into each page and avoid replacing subproblems that will be expanded in the immediate future. Let n_m be the maximum size of the m-list and $P_1, \cdots, P_{n_m}$, $g(P_{i_1}) \leq \cdots \leq g(P_{i_{n_m}})$, be the subproblems in the m-list when replacement is initiated. We have analyzed an effective replacement rule $S(\theta, \phi)$ that selects subproblems $P_{i_{\theta+1}}, \cdots, P_{i_{\theta+\phi}}$ for removal, where $\theta \geq 0$, $\theta + \phi \leq n_m$ [38]. This replacement rule tends to minimize the number of B^+-tree pages accessed. A special case of this rule is $S(n_m - \phi, n_m)$, or the *Back-Replacement policy*, in which the ϕ subproblems with the largest lower bounds are replaced.

The problem of selecting the proper replacement rule is reduced to the selection of θ (the index of the first subproblem to be replaced) and ϕ (the number of subproblems to be removed) to minimize η, the expected number of page accesses normalized by the expected number of subproblems expanded. In Appendix A, we have derived an expression for η (A-12) that is a function of θ, ϕ, s (the branching factor), n_p (the page size in number of subproblems), π_m (the probability that a subproblem expanded is chosen from the s-buffer), and $E(R)$ (the average number of B^+-tree pages). Equation (A-12) was verified experimentally on integer-programming problems, and a subset of the simulation results are shown in Table II. These results show that (A-12) is a reasonably good measure of η for integer-programming problems.

TABLE II
COMPARISON OF η TO η_{actual} FOR TWO 20-VARIABLE, 20-CONSTRAINT INTEGER-PROGRAMMING PROBLEMS ($n_m = 100$, $n_p = 50$)

θ	ϕ	Problem 1		Problem 2	
		Predicted η	Actual η	Predicted η	Actual η
0	50	0.80	0.80	0.47	0.46
10	50	0.65	0.73	0.37	0.41
20	50	0.56	0.68	0.31	0.39
30	50	0.51	0.64	0.29	0.36
40	50	0.50	0.61	0.28	0.33
50	50	0.54	0.59	0.28	0.31
80	20	0.65	0.73	0.34	0.41
70	30	0.60	0.68	0.30	0.36
60	40	0.56	0.63	0.29	0.35
50	50	0.54	0.60	0.28	0.31
40	60	0.53	0.57	0.28	0.31
30	70	0.53	0.54	0.29	0.30
20	80	0.54	0.54	0.30	0.31
10	90	0.55	0.55	0.31	0.31

It is difficult to choose θ and ϕ that will minimize η for a given set of parameters because some of these parameters are complex functions of θ and ϕ.[4] Instead, we optimize θ and ϕ iteratively. By using quadratic approximations of the variations, a nonlinear programming optimization indicates that a Back-Replacement policy should be used. The details of this optimization will not be shown here [38]. The use of the Back-Replacement policy is also suggested by statistics collected during simulations of integer-programming problems [Fig. 3(a)]. Only the results of four problems are shown here, but other problems simulated exhibit a similar behavior.

Assuming that the Back-Replacement policy is used, ϕ may be chosen similarly. It was found that ϕ should be set between $0.7\ n_m$ and $0.9\ n_m$ [38]. Statistics collected during simulations of integer-programming problems also suggest a similar result although the actual performance is problem-dependent [Figure 3(b)]. Other simulations also exhibit a similar behavior. In practice, any ϕ in the range of $0.7\ n_m$ to $0.9\ n_m$ is satisfactory as the number of page accesses in this region differs from the minimum by at most 10 percent.

Given the replacement algorithm, the optimal page size can be determined. The cost of accessing a page on a

[4]Simulations of integer-programming problems showed that the parameters π_m, λ_m, and λ, (defined in Appendix A) vary with both θ and ϕ [38].

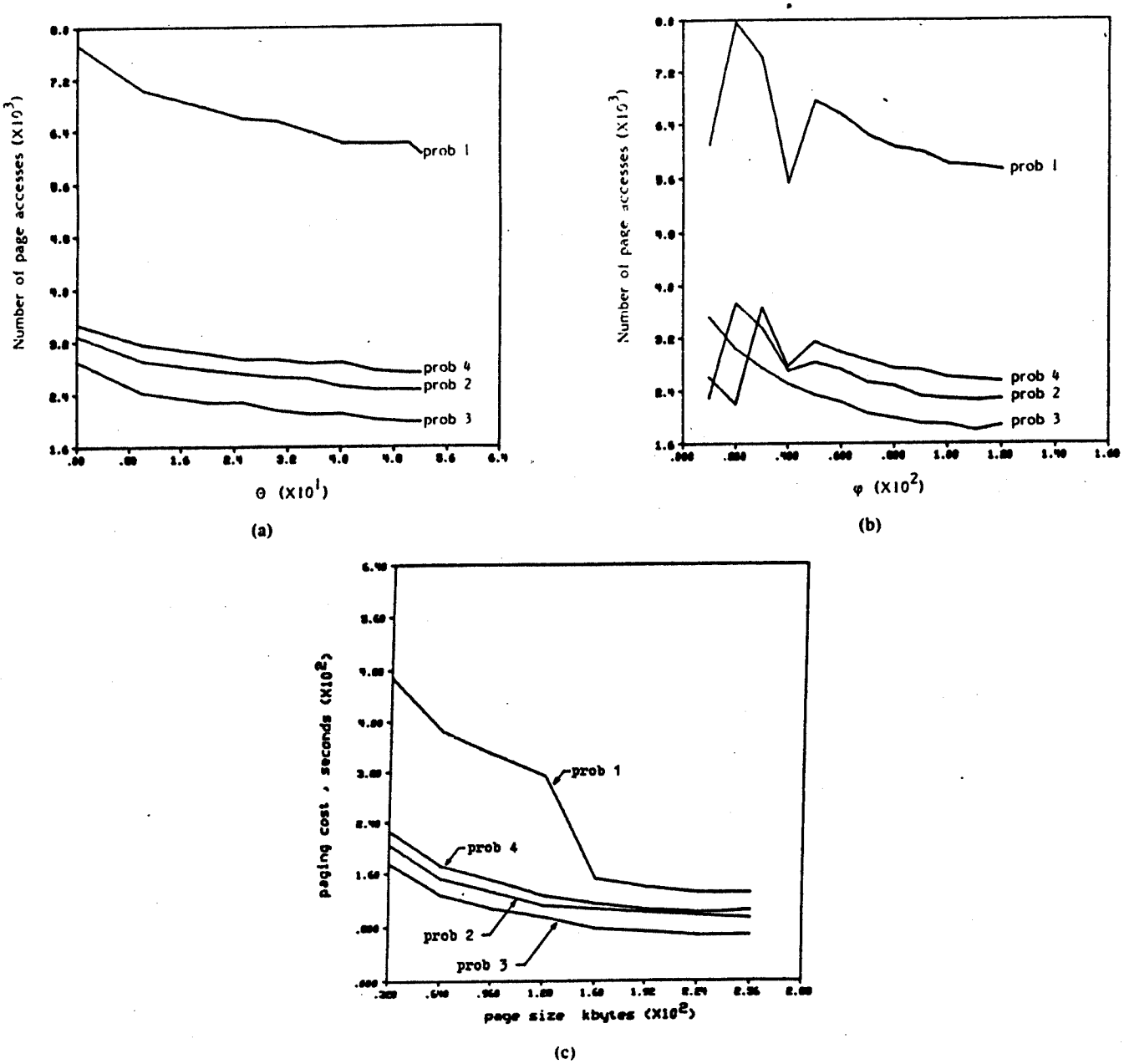


Fig. 3. (a) Variation of the number of page accesses with θ for four 20-variable, 20-constraint integer-programming problems ($n_m = 132$, $n_p = 33$, $\phi = 80$). (b) Variation of the number of page accesses with ϕ for four 20-variable, 20-constraint integer-programming problems ($n_m = 132$, $n_p = 33$, $\theta = n_m - \phi$). (c) Variation of the total paging cost with page size on a moving-head disk (seek time = 28 ms, rotational delay = 16.67 ms) with a disk density of 16 kbytes/track for four 20-variable, 20-constraint integer-programming problems ($n_m = 132$, $\phi = 120$, $\theta = n_m - \phi$, size of each subproblem = 1974 bytes).

noving-head disk is given by the sum of the average seek, otational, and transfer times. For integer-programming roblems, it was found that n_p (the page size) in the range f 0.65 n_m to 0.85 n_m should be used [Fig. 3(c)].

The performance comparison between a direct implementation of the B&B algorithm and the modified virtual memory will be shown after the discussion of the modified B&B algorithm.

IV. Modified Best-First B&B Algorithms [39]

In this algorithm, the range of possible lower bounds is artitioned into n disjoint regions, $\{[u_0, u_1), [u_1, u_2), \cdots, [u_{n-1}, u_n)\}$ (Fig. 4). Normally $u_0 = g_0$ and $u_n = g^e_{opt}$, where g_0 is the lower bound of the initial problem, nd g^e_{opt} is an estimate of the value of the optimal solution. The list of active subproblems in each region is usually large and so has to be partly stored in the secondary memory. Further, accesses to the secondary memory incurs a high overhead, and hence subproblems in each region $[u_i, u_{i+1})$, $i = 0, \cdots, n-1$, are not ordered by lower bounds but are kept in a last-in, first-out stack, S_i, and expanded in a depth-first manner.

This approach contrasts with the IDA* approach of using a constant stack size. Our approach seeks a space-time tradeoff that utilizes the available main and secondary memory as well as the overlapping of the accesses to main and secondary memories.

The efficiency of the algorithm depends on whether it is implemented on a file system or a virtual-memory system. We will describe the version for a file system here

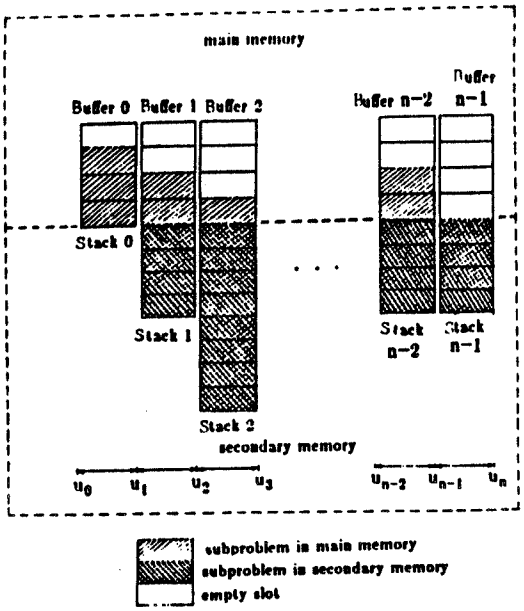


Fig. 4. Data structure to support the modified B&B algorithm.

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

Fig. 5. The modified branch-and-bound algorithm.

and note the needed changes for a virtual-memory implementation. The main-memory space allocated to the storage of subproblems is organized as a pool of fixed-sized frames. Stack S_i is partitioned into F_i, a secondary-memory file, and B_i, a main-memory buffer. Buffer B_i constitutes the top portion of the S_i and consists of a number of frames. Subproblems pushed onto stack S_i are written into B_i. When B_i is full, a new empty frame is requested from the frame pool in the main memory. When no free frames are available, subproblems in the bottom frame of B_i are written to F_i. It is assumed that secondary-memory buffers exists, and prefetching from the secondary-memory can be done in overlap with computations.

Let the *current stack* be the stack containing the active subproblem with the smallest lower bound, and the *solution stack* be the stack containing the optimal solution. Stack S_j is defined to be *smaller* than stack S_k if $j < k$, and larger than stack S_k if $i > k$. Similar relationships hold for the buffers B_j and B_k. Hence the current stack is also the *smallest stack*. An *active stack* is one that is larger than the current stack and has not been eliminated by lower-bound tests.

In the *modified selection rule*, a subproblem in the smallest nonempty main-memory buffer is chosen for expansion. Note that the main-memory buffer for the smallest-stack may be empty, and that the overhead may be too large to fetch and wait for a subproblem in the smallest stack from the secondary memory. In this case, the fetch should be initiated, but the expansion should proceed with any subproblem in the smallest nonempty main-memory buffer. This *No-Wait Policy* is based on the facts that the selection overhead is high and that the lower-bound heuristic may be fallible in guiding the search [37].

Let the incumbent be z and the allowance function be $\epsilon(z)$. Suppose that $(z - \epsilon(z))$ lies in the range of S_k. The *modified lower-bound elimination rule* eliminates all active subproblems with lower bounds greater than $(z - \epsilon(z))$ are not eliminated immediately because the overhead of accessing them in the secondary memory is large, and S_k may be eliminated when a future incumbent is found. Subproblems in F_k that are larger than the incumbent will be eliminated when they are staged to the main memory during the expansion of S_k. As a result, the lower-bound test has to be carried out on each selected subproblem before it is expanded. The *modified B&B algorithm* is summarized in Fig. 5.

More subproblems may be expanded in the modified B&B algorithm than in the conventional best-first search. Some subproblems in the solution stack S_r, with lower bounds greater than the value of the optimal solution as well as some subproblems from stacks $S_{r+1}, \cdots, S_{n-1}$ may be expanded due to the modified selection rule. These subproblems are never expanded in the conventional best-first search. The extra overhead of expanding these additional subproblems will be offset by the savings in waiting times due to page reads. Additional secondary-storage space to store subproblems that are not pruned immediately by the modified lower-bound elimination rule is also necessary in the modified B&B algorithm.

When one stack is used, the modified B&B algorithm is identical to a depth-first search; when infinitely many stacks are used, it is identical to a best-first search. In general, as the number of stacks increases, the number of expanded subproblems decreases and the secondary-memory overhead increases. The assignment of the stack boundaries and the number of stacks should be chosen to maximize the overlap between computations and secondary-memory accesses.

In Appendix B (B-20), we have derived an analytical expression for $E[D(n)]$, the expected overhead for a given assignment of the stack boundaries when n stacks are used. By enumerating $E[D(n)]$ over feasible assignments, the assignment that minimizes $E[D(n)]$ for a

TABLE III
NORMALIZED OVERHEADS FOR THE EQUAL-WIDTH (EWA) AND ANALYTICAL (AA) BOUNDARY ASSIGNMENTS FOR FIVE 80-NODE VERTEX-COVER PROBLEM AND FIVE 20-VARIABLE, 20-CONSTRAINT INTEGER-PROGRAMMING PROBLEMS (n_m = 16 (128 kbytes)). FOR INTEGER-PROGRAMMING PROBLEMS, n_p = 4, and $g^r_{ipp} = (1 - \log_r(x))\, g_{ipp}$, WHERE x IS A RANDOM NUMBER BETWEEN 0 AND 1. FOR VERTEX-COVER PROBLEMS, n_p = 2, AND g^r_{ipp} IS GENERATED BY A GREEDY ALGORITHM THAT ALWAYS CHOOSES THE VERTEX WITH LARGEST DEGREE.

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

assigning stack boundaries is to divide the range of possible lower bounds into equal-width regions.

We have carried out extensive simulations for the modified B&B algorithm with a number of integer-programming and vertex-cover problems using analytical (AA) and equal-width (EWA) stack-boundary assignments. Some of these results are shown in Table III. The values for the overhead have been normalized with respect to h_c, the mean computational time required to expand a subproblem. It can be seen that the performance of the heuristic and the analytical schemes are approximately the same.

These extensive simulations also suggest that for integer-programming and vertex-cover problems, *three to four stacks should be used to minimize the overhead*. This is clearly seen from Table III that shows the variations of overheads with respect to the number of stacks for a sample of integer-programming and vertex-cover problems. Moreover, an appropriate number of stacks and their boundaries can be predicted quite accurately for the given problem in general. However, stack assignments are problem dependent, and the analytical method proposed here can be used as a guideline to select the number of stacks and their boundaries *before the given problem is actually solved*.

In a virtual-memory implementation of the modified B&B algorithm, the stacks will have to be defined in virtual space, and the transfers of subproblems between the main and secondary memories are automatically managed by the operating system. The frame pool and its associated routines are no longer needed. However, it is unlikely that reads from the secondary memory to the main memory can be overlapped with computations, so the analysis of overheads must be altered accordingly. The analysis of such a scheme is very similar to that presented here and will not be shown.

V. COMPARISONS OF VARIOUS SCHEMES

Extensive simulations have been performed to compare the various schemes proposed in this paper. The simulation programs were written in C language and were implemented on the DEC VAX 11/780 computer. The simulations took over one hundred hours of CPU time to complete. In the simulator, a main memory of 128 kbytes (10 percent of the virtual space) was assumed. Due to space limitation, only a sample of the simulation results are shown here [39].

Simulation results for two 20-variable, 20-constraint integer-programming problems are shown in Fig. 6. Each subproblem required a memory space of 1974 bytes. The logical file block size of the modified B&B scheme and the page size of the direct implementation were fixed to be 8 kbytes. For the modified virtual-memory scheme, a page size of 16 kbytes was used. The page access time was adjusted according to the assumptions that the disk has a mean seek time of 28 ms and a mean rotational delay of 8.33 ms, and that track capacities of 16 and 64 kbytes were used. These correspond to transfer rates of approximately 1 and 4 Mbytes/s, respectively. Only the case of three stacks were simulated for the modified B&B scheme. The overhead is defined as the completion time of the algorithm normalized by h_c. For the range of h_d/h_c simulated, there is a two to fivefold reduction in the completion time for the modified B&B scheme as compared to any other search strategies Other simulation results also exhibit a similar behavior.

Some simulation results for two 80-node vertex-cover problems are shown in Fig. 7. Each subproblem required a memory space of 330 bytes. The modified B&B algorithm for the vertex-cover problem performs almost identically as a depth-first search for these problems because the lower-bound function is very accurate in guiding the search.

In conclusion, although a best-first search minimizes the total number of subproblems expanded, it will not always be the most time-efficient search strategy on a two-level memory hierarchy because of the extensive memory space required. On the other hand, a depth-first search is space efficient but requires more computational overhead than that of the best-first search. The most time-efficient strategy will depend on both the disparity between the numbers of subproblems expanded under the best-first (n_{bfs}) and depth-first searches (n_{dfs}), and the relative speeds of the main and secondary memories. In this paper, we have proposed a modified branch-and-bound algorithm that is a hybrid of the best-first and depth-first searches and have developed guidelines in choosing a suitable search strategy. These guidelines are summarized in Table IV.

This modified B&B algorithm [39] also compares favorably with IDA* [42], [43]. Both algorithms will find

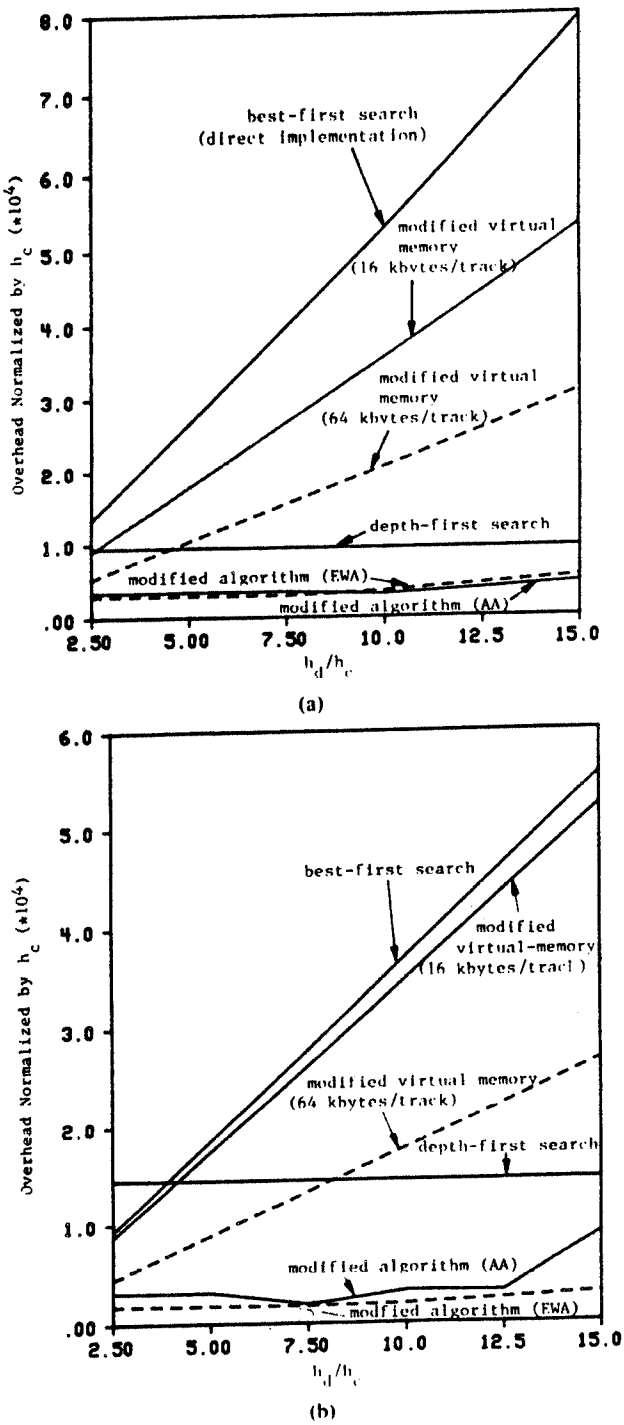


Fig. 6. Simulation results for B&B algorithms on two 20-by-20 integer-programming problems. (Main memory size is 128 kbytes.)

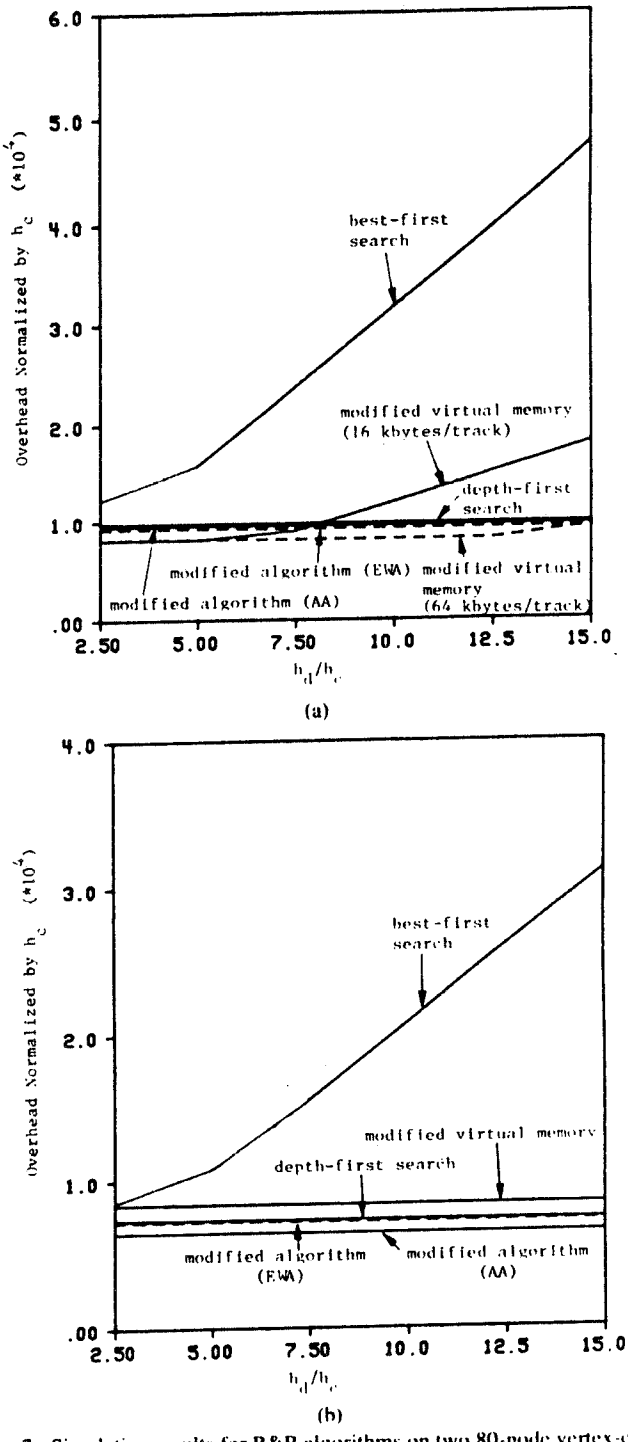


Fig. 7. Simulation results for B&B algorithms on two 80-node vertex-cover problems. (Main memory size is 128 kbytes. The completion time for the modified B&B scheme is the same for both 16 kbytes/track and 64 kbytes/track in the second problem.)

the optimal solution if one exists. IDA* is optimal in space. Modified B&B uses more space than IDA* to avoid multiple expansions of nodes. Both algorithms are asymptotically optimal in time but neither algorithm gives the minimum completion time for a given problem. Both algorithms expand more nodes than best-first search. IDA* does multiple expansions of nodes. Modified B&B may expand nodes in the last stack with lower bounds greater than the optimal solution. The better algorithm depends on the problem and the choice of parameters for modified B&B.

TABLE IV
SUMMARY OF SEARCH STRATEGIES TO BE USED IN DIFFERENT ENVIRONMENT (dfs: DEPTH-FIRST SEARCH; bfs: BEST-FIRST SEARCH; mbb: MODIFIED BRANCH-AND-BOUND ALGORITHM; n_{bfs}: NUMBER OF NODES EVALUATED IN bfs; AND n_{dfs}: NUMBER OF NODES EVALUATED IN dfs).

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

rithm in this paper illustrates one important characteristic of many nondeterministic search problems such as evaluating a logic program or searching for a good move in a chess game. These problems have an unpredictable number of tasks to be performed before the problem is solved. The nondeterminism arises because the search is guided a fallible heuristic function that does not always lead in the right direction and results in backtracking when a wrong decision is made. Therefore, the strict adherence to the heuristic function may not always be rewarding, especially when the overhead of doing so is high and the heuristic function is not accurate in guiding the search. In this case, the adherence to the heuristic function should be relaxed. This method has been applied to design the modified selection rule in this paper and a parallel computer system for searching branch-and-bound trees [35], [37].

Appendix A
Expected Number of Page Accesses in the Modified Virtual-Memory System

Let n_m be the maximum size of the m-list. Replacement is initiated when there are n_m subproblems in the m-list. Let P_i, $i = 1, \cdots, n_m$, $g(P_1) \le \cdots \le g(P_{n_m})$, be the subproblems in the m-list. Similarly, let E_i, $i = 1, \cdots, n_s$, $g(E_1) \le \cdots \le g(E_{n_s})$, be the subproblems in the secondary memory. We can then define $X_j = g(P_j) - g(P_1)$, $j = 1, \cdots, n_m$, and $Y_k = g(E_k) - g(E_1)$, $k = 1, \cdots, n_s$. It is assumed

1) that $g(P_1) = g(E_1)$;

2) that X_i, $i = 1, \cdots, n_m$ are iid random variables with an exponential density function $f_X(x) = \lambda_m e^{-\lambda_m x}$;

3) that Y_j, $j = 1, \cdots, n_s$ are iid random variables with an exponential density function $f_Y(y) = \lambda_s e^{-\lambda_s y}$.

Assumptions 2) and 3) have been verified experimentally to be gamma with $\alpha < 1$ for integer-programming and knapsack problems [38]. The exponential distribution is used here as an approximation. The ϕ subproblems selected for removal are $P_{\theta+1}, \cdots, P_{\theta+\phi}$, where $X_{\theta+1} = v$ and $X_{\theta+\phi} = w$.

Let ω_s be the number of subproblems in the secondary memory with $v < Y_i \le w$. The probability that $\omega_s = m$ constitutes a binomial distribution.

$$\Pr\{\omega_s = m\} = \binom{n_s}{m} (\Pr\{v < Y_i \le w\})^m \cdot (1 - \Pr\{v < Y_i \le w\})^{n_s - m}$$

where $\Pr\{v < Y_i \le w\} = \int_v^w \lambda_s e^{-\lambda_s x}\, dx$ according to Assumption 3). The expected value of ω_s given that $X_{\theta+1} = v$, $X_{\theta+\phi} = w$, and $n_s = n$ is

$$E[\omega_s \mid X_\theta = v, X_{\theta+\phi} = w, n_s = n] = n \cdot \Pr\{v < Y_i \le w\}. \quad \text{(A-1)}$$

Since n_s and $\Pr\{v < Y_i \le w\}$ are independent, (A-1) may be evaluated as

$$E[\omega_s] = E[n_s] \int_{w=0}^{\infty} \int_{v=0}^{\infty} \Pr\{v < Y_i \le w\} \cdot f_v(v) f_w(w)\, dv\, dw$$

$$= E[n_s] \left(\frac{n_m!}{(n_m - \theta)!} \frac{\Gamma(n_m - \theta + 1 + \zeta)}{\Gamma(n_m + 1 + \zeta)} - \frac{n_m!}{(n_m - \theta - \phi)!} \frac{\Gamma(n_m - \theta - \phi + 1 + \zeta)}{\Gamma(n_m + 1 + \zeta)} \right) \quad \text{(A-2)}$$

where $\zeta = \lambda_s / \lambda_m$.

Assume that the ω_s subproblems reside in R pages, and let $Q_1, \cdots, Q_R$ be the numbers of subproblems in these R pages. It is further assumed that $Q_1, \cdots, Q_R$ are iid random variables with a uniform distribution between $((n_p/2) + 1)$ and n_p, where n_p is the size of each page in subproblems. Since R is a stopping time for the renewal series Q_is, renewal theory and Wald's equation [29] give

$$E[\omega_s] = E[R]\, E[Q]$$

and

$$E[R] = \frac{4}{3n_p + 2} E[n_s] \left(\frac{n_m!}{(n_m - \theta)!} \cdot \frac{\Gamma(n_m - \theta + 1 + \zeta)}{\Gamma(n_m + 1 + \zeta)} - \frac{n_m!}{(n_m - \theta - \phi)!} \frac{\Gamma(n_m - \theta - \phi + 1 + \zeta)}{\Gamma(n_m + 1 + \zeta)} \right). \quad \text{(A-3)}$$

The derivation of the distribution function of R is extremely difficult. For mathematical tractability, the expected value of R will be used in the following analysis instead. Let the ϕ subproblems be actually inserted into G of the $E[R]$ pages. Assuming that these ϕ insertions are uniformly distributed among the $E[R]$ pages, K, the number of subproblems inserted into any page, satisfies the binomial distribution.

$$\Pr\{K = j\} = \binom{\phi}{j} \left(\frac{1}{E[R]}\right)^j \left(1 - \frac{1}{E[R]}\right)^{\phi - j}.$$

The expected number of pages into which subproblems are inserted is

$$E[G] = E[R] \Pr\{K \ge 1\} = E[R] \cdot \left| 1 - \left(1 - \frac{1}{E[R]}\right)^{\phi} \right|. \quad \text{(A-4)}$$

H of the G pages will overflow and have to be split. A page containing q subproblems will be split if more than $(n_p - q)$ subproblems are inserted into it. Let $\nu = \max((n_p/2) + 1, n_p - \phi + 1)$. π_s, the probability that a page has to be split, is

$$\pi_s = \sum_{q=\nu}^{n_p} \Pr\{\text{split page} \mid Q = q\} \cdot \Pr\{Q = q\}$$

$$= \sum_{q=\nu}^{n_p} \left(1 - \Pr\{K \le n_p - q\}\right) \frac{2}{n_p}$$

$$= \frac{2}{n_p}(n_p + 1 + \nu) - \frac{2}{n_p} \sum_{j=0}^{n_p-\nu} \left\{ (n_p + 1 - \nu - j) \cdot \binom{\phi}{j} \left(\frac{1}{E[R]}\right)^j \left(1 - \frac{1}{E[R]}\right)^{\phi - j} \right\}. \quad \text{(A-5)}$$

This page splitting process may be viewed as a binomial experiment with π_s as the probability of success, and the expected number of pages split, $E[H] = \pi_s \cdot E[G]$. For $(E[G] - E[H])$ pages that are not split, there is an overhead of one write per page since the system can examine the bit map to decide the locations for placing the replaced subproblems inside a page without reading the page into the main memory. For $E[H]$ pages that have to be split, it is necessary to read the page once to discover the median lower bound. This is followed by one page-read to retrieve the necessary subproblems to be split into the second page, one page-write to insert the replaced subproblems into the first page, and one page-write to write the second page to disk. Hence $E[T_s]$, the expected number of page accesses incurred during a replacement, is given by

$$E[T_s] = (E[G] - E[H]) + 4E[H]$$

$$= (1 + 3\pi_s) E[G]$$

$$= \left\{ 1 + \frac{6}{n_p}(n_p + 1 - \nu) - \frac{6}{n_p} \sum_{j=0}^{n_p-\nu} \left| (n_p + 1 - \nu - j) \cdot \binom{\phi}{j} \cdot \left(\frac{1}{E[R]}\right)^j \left(1 - \frac{1}{E[R]}\right)^{\phi - j} \right| \right\} \times E[R] \left| 1 - \left(1 - \frac{1}{E[R]}\right)^{\phi} \right|. \quad \text{(A-6)}$$

We will now derive an expression for the expected number of subproblems expanded between two consecutive replacements when the space reserved for the m-list and the s-buffer is fixed. Let π_m be the probability that a subproblem selected for expansion is from the s-buffer. Also, let D be a random variable indicating the number of subproblems expanded between two replacements. Of these D subproblems, let D_s subproblems be from the s-buffer and D_m subproblems be from the m-list. Thus, $D = D_s + D_m$ and

$$\Pr\{D_s = k \mid D\} = \binom{D}{k} (\pi_m)^k (1 - \pi_m)^{D-k}.$$

The expected values of D_s and D_m are given by

$$E[D_s] = E[E[D_s \mid D]] = \pi_m E[D] \quad \text{(A-7)}$$

and

$$E[D_m] = (1 - \pi_m) E[D]. \quad \text{(A-8)}$$

The s subproblems generated by expanding a subproblem are inserted into the m-list. The size of the m-list increases by s when the subproblem expanded is from the s-buffer, and increases by $(s - 1)$ when the subproblem expanded is from the m-list (since the expanded subproblem has been deleted from the m-list and an empty slot is created). Equating the increase in the size of the m-list between two replacements to the fixed number of subproblems removed during a replacement, we have

$$\phi = s \cdot E[D_s] + (s - 1) \cdot E[D_m]$$

$$= (s - 1 + \pi_m) E[D]$$

and

$$E[D] = \frac{\phi}{s - 1 + \pi_m}. \quad \text{(A-9)}$$

Let the D_s subproblems be residing in T_t pages of the secondary memory, and let $Q'_1, \cdots, Q'_{T_t}$ be the number of subproblems in each of these T_t pages. Q'_i have the same distribution as the Q_i. Since T_t is a stopping time for the renewal series Q'_i, renewal theory and Wald's equation give $E[D_s] = E[T_t] \cdot E[Q]$ or

$$E[T_t] = \frac{4E[D_s]}{3n_p + 2} = \frac{4}{3n_p + 2} \frac{\pi_m \phi}{s - 1 + \pi_m}. \quad \text{(A-10)}$$

$E[T_t]$ is the expected number of page accesses due to the loading of the s-buffer between two successive replacements.

From its definition, η may be written as

$$\eta = \frac{E[T_s] + E[T_t]}{E[D]}. \quad \text{(A-11)}$$

Substituting (A-6), (A-9), and (A-10) into (A-11) yields

$$\eta = \frac{s - 1 + \pi_m}{\phi} \left\{ \frac{4}{3n_p + 2} \frac{\pi_m \phi}{s - 1 + \pi_m} + \left| 1 + \frac{6(n_p + 1 - \nu)}{n_p} - \frac{6}{n_p} \sum_{j=0}^{n_p-\nu} (n_p - \nu + 1 - j) \binom{\phi}{j} \left(\frac{1}{E[R]}\right)^j \cdot \left(1 - \frac{1}{E[R]}\right)^{\phi - j} \right| \cdot E[R] \cdot \left(1 - \left(1 - \frac{1}{E[R]}\right)^{\phi}\right) \right\}. \quad \text{(A-12)}$$

Appendix B
Expected Overhead for the Modified B&B Algorithm

This is based on the wall model [34], [36] of a best-first search. In this model, the front wall represents the lower bound of the subproblem being expanded, while the back wall represents the incumbent. These values are nor-

malized by the lower bound of the initial problem, $g(P_0) = g_0$. For any subproblem P_j, let $\Delta(P_j) = g(P_j) - g_0$. Initially, the front wall is at $\Delta(P_0) = 0$, and the back wall is at infinity. The front wall always moves to the right in a best-first search as the lower bound of a subproblem is always greater than that of its parent. The back wall always moves to the left and moves to $\Delta(P_s)$ when a better feasible solution P_s is generated. Successive subproblem expansions cause the front and back walls to approach each other, and the process terminates when the two walls meet.

When the front wall is at position x, let $N(x)$ and $M(x)$ denote the numbers of expanded and generated subproblems, respectively. It is assumed that the difference between the lower bounds of a subproblem and its parent subproblem is an exponential random variable with density function, $f_y(y) = \lambda e^{-\lambda y}$ (this distribution has been verified experimentally to be gamma with $\alpha < 1$ for integer-programming and knapsack problems [34], [36]; the exponential distribution is used here as an approximation). Further, each subproblem expanded is assumed to generate s subproblems. The expected value $N(x)$ has been shown before [34], [36] to be

$$E[N(x)] \approx \frac{s}{s-1} e^{\lambda(s-1)x} - \frac{1}{s-1} \qquad \text{(B-1)}$$

and

$$M(x) = sN(x) + 1. \qquad \text{(B-2)}$$

Let $Z_1, \cdots, Z_{N(x)}$ be iid random variables with distribution $F_z(z \mid x, x \geq z)$. Suppose that Z_i has value z_i, and, without loss of generality, let $z_1 \leq \cdots \leq z_{N(x)}$. As the $N(x)$ expanded subproblems are independent of each other, it is assumed that Z_i is stochastically identical to $\Delta(P_i)$. The expected number of expanded subproblems that have values not greater than z, $z < x$, is $E[N(z)]$, where

$$N(x)\, F_z(z \mid x, x \geq z) = E[N(z)]. \qquad \text{(B-3)}$$

Substituting (B-1) into (B-3),

$$F_z(z \mid x, x \geq z) = \frac{(se^{\lambda(s-1)z} - 1)}{(s-1)\, N(x)}. \qquad \text{(B-4)}$$

Differentiating (B-4) yields the density function, $f_z(z \mid x)$.

$$f_z(z \mid x, x \geq z) = \frac{s\lambda e^{\lambda(s-1)z}}{N(x)} \qquad \text{(B-5)}$$

Let Π_j, $j = 1, \cdots, N(x)$ be the $N(x)$ expanded subproblems and Ψ_k, $k = 1, \cdots, M(x)$ be the $M(x)$ generated subproblems when the front wall is at x. Suppose that Ψ_k is generated by the expansion of subproblem Π_j, and that I_k is the increase in lower bound due to this expansion. Then,

$$\Delta(\Psi_k) = \Delta(\Pi_j) + I_k = Z_j + I_k. \qquad \text{(B-6)}$$

$\Delta(\Psi_1), \cdots, \Delta(\Psi_{M(x)})$ will be iid random variables, as both $Z_1, \cdots, Z_{N(x)}$ and $I_1, \cdots, I_{M(x)}$ are sets of iid random variables. Let $\gamma_i(x)$ be the probability that $\Delta(\Psi_k)$, $k = 1, \cdots, M(x)$ lies within the range of Stack i, (that is, $(u_i - g_0) < \Delta(\Psi_k) \leq (u_{i+1} - g_0)$) when the front wall is at x. By (B-6) and the assumption that I_k is an exponentially distributed random variable, $\gamma_i(x)$ for given Z_j is

$$(\gamma_i(x) \mid Z_j = z) = \int_{y = u_i - g_0 - z}^{u_{i+1} - g_0 - z} \lambda e^{-\lambda y}\, dy = (e^{-\lambda(u_i - g_0)} - e^{-\lambda(u_{i+1} - g_0)})\, e^{\lambda z}. \qquad \text{(B-7)}$$

Unconditioning (B-7) and applying (B-5), we have

$$\gamma_i(x) = \int_{z=0}^{x} (e^{-\lambda(u_i - g_0)} - e^{-\lambda(u_{i+1} - g_0)}) \frac{s\lambda e^{\lambda sz}}{N(x)}\, dz \approx \frac{1}{N(x)} (e^{-\lambda(u_i - g_0)} - e^{-\lambda(u_{i+1} - g_0)}) e^{\lambda sx}. \qquad \text{(B-8)}$$

Of the $M(x)$ generated subproblems, the expected number of subproblems in the region $[u_i - g_0, u_{i+1} - g_0)$ is

$$E[W_i(x)] = M(x)\, \gamma_i(x) \approx sN(x)\, \gamma_i(x). \qquad \text{(B-9)}$$

Substituting (B-8) into (B-9) yields

$$E[W_i(x)] \approx (e^{-\lambda(u_i - g_0)} - e^{-\lambda(u_{i+1} - g_0)}) se^{\lambda sx}. \qquad \text{(B-10)}$$

Suppose that g_{opt} is the value of the optimal solution and that S_r is the solution stack (i.e., $u_r < g_{opt} \leq u_{r+1}$). Let $E[q \mid r]$ be the expected number of expanded subproblems when the algorithm terminates with the optimal solution in S_r. In the modified B&B algorithm, S_r is expanded in a LIFO fashion, so the number of expanded subproblems when the algorithm terminates will be greater than $N(g_{opt})$. It is extremely difficult to model the LIFO expansion of subproblems in S_r. Moreover, g_{opt} is unknown before the problem is solved. A possible simplification is to assume that all subproblems in S_r are expanded. Then

$$E[q \mid r] = E[N(u_{r+1} - g_0)] \approx \frac{s}{s-1} e^{\lambda(s-1)(u_{r+1} - g_0)}. \qquad \text{(B-11)}$$

Let π_r be the probability that S_r is the solution stack. The value of the optimal solution is assumed to be uniformly distributed over the range $[g_0, g^{\epsilon}_{opt})$. This assumption is very difficult to verify and is made here for mathematical tractability. For a problem with real-valued solutions,

$$\pi_r = \frac{u_{r+1} - u_r}{g^{\epsilon}_{opt} - g_0}. \qquad \text{(B-12)}$$

Similarly, for a problem with integer-valued solutions,

$$\pi_r = \frac{(\lfloor u_{i+1} \rfloor - \lfloor u_i \rfloor)}{(\lfloor g^{\epsilon}_{opt} \rfloor - \lfloor g_0 \rfloor)}. \qquad \text{(B-13)}$$

$E[t_c]$, the expected computational overhead, is

$$E(t_c) = \sum_{r=0}^{n-1} E[q \mid r]\, h_c \pi_r$$

$$= \sum_{r=0}^{n-1} \left(\frac{s}{s-1} e^{\lambda(s-1)(u_{r+1} - g_0)} \right) h_c \pi_r \quad \text{(B-14)}$$

where h_c is the mean computational time for expanding a subproblem.

Only the secondary-memory overhead due to the transfer of subproblems will be considered in this analysis. Each frame holds n_p subproblems, and no more than n_m frames are allowed in the main memory. n_0 of these frames are reserved for the current stack to be used in a depth-first expansion.

Let m_i be the number of subproblems in S_i when it becomes either the current stack or an eliminated stack. Equation (B-10) may be used to derive an approximate expression for $E[m_i \mid r]$, the expected value of m_i given that S_r is the solution stack. For $1 \le i \le r$, m_i is the number of subproblems in S_i when it becomes the current stack. The approximation that $E[m_i \mid r] \approx E[W_i(u_i - g_0)]$ can be used if few subproblems in S_i have been chosen for expansion prior to S_i becoming the current stack. For $(r+1) \le i \le (n-1)$, m_i is the number of subproblems in S_i when S_i is eliminated. If no stacks are eliminated prior to the generation of the optimal solution, then $E[m_i \mid r] \approx E[W_i(u_{r+1} - g_0)]$. This is a good approximation when the number of feasible solutions generated is small, such as the integer-programming and vertex-cover problems. Thus,

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

$$\approx \begin{cases} (e^{-\lambda(u_i - g_0)} - e^{-\lambda(u_{i+1} - g_0)})\, s e^{\lambda s(u_i - g_0)} \\ \quad 1 \le i \le r \\ (e^{-\lambda(u_i - g_0)} - e^{-\lambda(u_{i+1} - g_0)})\, s e^{\lambda s(u_{r+1} - g_0)} \\ \quad (r+1) \le i \le (n-1). \end{cases} \quad \text{(B-15)}$$

Once S_i becomes the current stack, subproblems in S_i are expanded in a depth-first fashion. Since n_0 frames have been reserved for the current stack, subproblems pushed onto the current stack are not written to the secondary memory. Also, no subproblems will be pushed onto a stack once it has been eliminated. Thus, m_i is the maximum number of subproblems in Stack S_i that will be written to the secondary memory.

Denote the equivalent number of frames in S_i by b_i. Its expected value given the solution stack S_r is

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

$$\approx \begin{cases} \left\lceil \dfrac{(e^{-\lambda(u_i - g_0)} - e^{-\lambda(u_{i+1} - g_0)})\, s e^{\lambda s(u_i - g_0)}}{n_p} \right\rceil \\ \quad 1 \le i \le r \\ \left\lceil \dfrac{(e^{-\lambda(u_i - g_0)} - e^{-\lambda(u_{i+1} - g_0)})\, s e^{\lambda s(u_{r+1} - g_0)}}{n_p} \right\rceil \\ \quad (r+1) \le i \le (n-1). \end{cases} \quad \text{(B-16)}$$

Let c_i be the number of frames allocated to B_i when S_i becomes either the current stack or an eliminated stack. If $1 \le i \le r$, and no stack has been previously eliminated, then there will be $(n-i-1)$ nonempty buffers that occupy frames when this occurs. Similarly, if $(r+1) \le i \le (n-1)$, and no stack has been eliminated prior to the generation of the optimal solution, there there will be $(n-r-1)$ buffers occupying frames. Since subproblem expansions are assumed to be independent, the number of frames allocated to each buffer will be proportional to the number of subproblems in the corresponding stack. The expected value of c_i when the solution stack S_r is known can be expressed as

$$E[c_i \mid r] = \begin{cases} \dfrac{(e^{-\lambda(u_i - g_0)} - e^{-\lambda(u_{i+1} - g_0)})}{(e^{-\lambda(u_{\min(i,r+1)} - g_0)} - e^{-\lambda(u_n - g_0)})} (n_m - n_0) \\ \quad \displaystyle\sum_{j=\min(i,r+1)}^{n-1} E[b_i \mid r] \ge (n_m - n_0) \\ E[b_i \mid r] \\ \quad \displaystyle\sum_{j=\min(i,r+1)}^{n-1} E[b_i \mid r] < (n_m - n_0). \end{cases} \quad \text{(B-17)}$$

The equivalent of $(b_i - c_i)$ frames of subproblems will be written into file F_i. These subproblems will eventually be read back into the main memory only if S_i becomes the current stack, and will be discarded if S_i is eliminated. Thus the expected secondary-memory overhead for S_i given the solution stack S_r will be

$$E[d_i \mid r] = \begin{cases} 0 \quad i = 0 \\ 2(E[b_i \mid r] - E[c_i \mid r])\, h_d \\ \quad 1 \le i \le r \\ (E[b_i \mid r] - E[c_i \mid r])\, h_d \\ \quad (r+1) \le i \le (n-1) \end{cases} \quad \text{(B-18)}$$

where h_d is the overhead for each page read. Unconditioning the solution stack S_r yields

$$E[t_d] = \sum_{r=0}^{n-1} \sum_{i=0}^{n-1} E[d_i \mid r]\, \pi_r. \quad \text{(B-19)}$$

The expected overhead for n stacks and a boundary assignment $\{u_0, \cdots, u_n\}$ is

$$E[D(n)] = \max\left(E[t_c], E[t_d]\right). \qquad \text{(B-20)}$$

REFERENCES

[1] T. O. Alanko, H. H. A. Erkio, and I. J. Haikala, "Virtual memory behavior of some sorting algorithms," *IEEE Trans. Software Eng.*, vol. SE-10, no. 4, pp. 422-431, July 1984.

[2] J. L. Baer and G. R. Sager, "Dynamic improvement of locality in virtual memory systems," *IEEE Trans. Software Eng.*, vol. SE-2, no. 1, pp. 54-62, Mar. 1976.

[3] H. Berliner, "The B tree search algorithm: A best-first proof procedure," *Artificial Intell.*, vol. 12, no. 1, pp. 23-40, 1979.

[4] B. S. Brown, F. G. Gustavson, and E. S. Mankin, "Sorting in a paging environment," *Commun. ACM*, vol. 13, no. 8, pp. 438-494, Aug. 1970.

[5] D. Comar, "The ubiquitous B-tree," *ACM Comput. Surveys*, vol. 11, no. 2, pp. 121-137, June 1979.

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

[7] D. Ferrari, "Improving locality for critical working sets," *Commun. ACM*, vol. 17, no. 11, pp. 614-620, Nov. 1974.

[8] ——, "Tailoring programs to models of program behavior," *IBM J. Res. Develop.*, vol. 19, no. 3, pp. 244-251, May 1975.

[9] ——, "The improvement of program behavior," *Computer*, pp. 39-47, Nov. 1976.

[10] P. C. Fisher and R. L. Probert, "Storage reorganization techniques for matrix computation in a paging environment," *Commun. ACM*, vol. 22, no. 7, pp. 405-415, July 1979.

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

[12] D. J. Hatfield and J. Gerald, "Program restructuring for virtual memory," *IBM Syst. J.*, vol. 10, no. 3, pp. 168-192, 1971.

[13] T. Ibaraki, "Computational efficiency of approximate branch-and-bound algorithms," *Math. Oper. Res.*, vol. 1, no. 3, pp. 287-298, 1976.

[14] ——, "Theoretical comparisons of search strategies in branch-and-bound algorithms," *Int. J. Computer and Information Sciences*, vol. 5, no. 4, pp. 315-343, 1976.

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

[16] ——, "Branch-and-bound procedure and state-space representation of combinatorial optimization problems," *Inform. Contr.*, vol. 36, pp. 1-27, 1978.

[17] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial Intell.*, vol. 6, no. 4, pp. 293-326, 1975.

[18] W. Kohler and K. Steiglitz, "Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems," *J. ACM*, vol. 21, no. 1, pp. 140-156, 1974.

[19] V. Kumar and L. N. Kanal, "A general branch and bound formulation for understanding and synthesizing and/or tree search procedures," *Artificial Intell.*, vol. 21, no. 1-2, pp. 179-198, 1983.

[20] E. L. Lawler and D. W. Wood, "Branch and bound methods: A survey," *Oper. Res.*, vol. 14, pp. 699-719, 1966.

[21] G.-J. Li and B. W. Wah, "Computational efficiency of parallel approximate branch-and-bound algorithms," in *Proc. Int. Conf. Parallel Processing*, IEEE 1984, pp. 473-480.

[22] A. Martelli and U. Montanari, "Optimizing decision trees through heuristically guided search," *Commun. ACM*, vol. 21, pp. 1025-1039, 1978.

[23] A. C. McKellar and E. G. Coffman, Jr., "Organizing matrices and matrix operations for paged memory systems," *Commun. ACM*, vol. 12, no. 3, pp. 153-164, 1969.

[24] L. Mitten, "Branch and bound methods: General formulation and properties," *Oper. Res.*, vol. 18, pp. 24-34, 1970.

[25] J. E. Morrison, "User program performance in virtual storage systems," *IBM Syst. J.*, vol. 5, no. 2, pp. 216-237, 1966.

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

[27] ——, *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga, 1980.

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

[29] S. M. Ross, in *Applied Probability Models with Optimization Applications*. San Francisco, CA: Holden Day, 1970.

[30] A. J. Smith, "Sequentiality and prefetching in database systems," *ACM Trans. Database Syst.*, vol. 3, no. 3, pp. 223-247, Sept. 1978.

[31] D. R. Smith, "Random trees and the analysis of branch-and-bound procedures," *J. ACM*, vol. 31, no. 1, pp. 163-188, 1984.

[32] G. C. Stockman and L. N. Kanal, "Problem reduction representation fo the linguistic analysis of waveforms," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-5, no. 3, pp. 287-298, 1983.

[33] H. S. Stone, "The average complexity of depth-first search," IBM, Res. Rep. RC 10717, 1984.

[34] B. W. Wah and C. F. Yu, "Probabilistic modeling of branch-and-bound algorithms," in *Proc. COMPSAC*, IEEE, 1982, pp. 647-623.

[35] B. W. Wah and Y. W. E. Ma, "MANIP—A multicomputer architecture for solving combinatorial extremum-search problems," *IEEE Trans. Comput.*, vol. C-33, no. 5, pp. 377-390, May 1984.

[36] B. W. Wah and C. F. Yu, "Stochastic modeling of branch-and-bound algoirthms with best-first search," *IEEE Trans. Software Eng.*, vol. SE-11, no. 9, pp. 922-934, Sept. 1985.

[37] B. W. Wah, G.-J. Li, and C. F. Yu, "Multiprocessing of combinatorial search problems," *Computer*, vol. 18, no. 6, pp. 93-108, June 1985.

[38] C. F. Yu and B. W. Wah, "Virtual-memory support for branch-and-bound algorithms," in *Proc. COMPSAC*, IEEE, Nov. 1983, pp. 618-626.

[39] ——, "Efficient branch-and-bound algorithms on a two-level memory system," in *Proc. COMPSAC*, IEEE, Nov. 1984, pp. 504-514.

[40] C.-F. Yu, "Efficient combinatorial search algorithms," Ph.D. dissertation, School Elec. Eng., Purdue Univ., West Lafayette, IN, Dec. 1986.

[41] C.-F. Yu and B. W. Wah, "Learning dominance relations in combinatorial search problems," in *Proc. Computer Software and Applications Conf.*, IEEE, Chicago, IL, Oct. 1986.

[42] R. Korf, "Depth-first iterative deepening: An optimal admissible tree search," *Artificial Intell.*, vol. 27, pp. 97-109, 1985.

[43] ——, "Iterative deepening A*, an optimal admissible tree search," in *Proc. 9th Int. Joint Conf. Artificial Intelligence*, Los Angeles, CA, pp. 1034-1036.

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

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

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

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

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