EFFICIENT BRANCH-AND-BOUND ALGORITHMS ON A TWO-LEVEL MEMORY SYSTEM

Chee-Fen Yu and Benjamin W. Wah
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

Abstract:
In this paper we have investigated the efficient evaluation of branch-and-bound algorithms with a best-first search on a two-level memory hierarchy. Three alternatives are compared here. First, the algorithm is studied with respect to a conventional virtual memory with a LRU replacement algorithm. Second, a specialized virtual memory is designed to execute the algorithm exactly. Lastly, the requirements of the algorithm are relaxed to enhance the locality of accesses, and the data structure is implemented as multiple stacks. It is found that there is over a 100-time reduction in disk overhead for integer-programming problems with the last approach as compared to a direct implementation on a conventional virtual memory; and there is a 60-time improvement over the second approach. The improvement is related to the memory size for storing a subproblem. For problems that have smaller subproblems, the improvement is even greater.


1. 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. 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, warehouse-location, job-shop-scheduling, knapsack, vertex-cover and integer-programming problems.

A general technique for solving combinatorial-search problems is the branch-and-bound (B&B) algorithm [LAW66]. A B&B algorithm is a partitioning algorithm that decomposes a problem into smaller subproblems and repeatedly decomposes until infeasibility is proved or a solution is found. Many theoretical properties of serial B&B algorithms have been developed [NIL80, KOH74, IB76a, IB76b, IB77, IB78]. It has also been recognized that the B&B algorithm is a generalization of many heuristic search procedures [KUM83] such as A’ [NIL71], AO* [MAR78], SSS’ [STO83], B’ [B&F79], alpha-beta [KNU75] and dynamic programming [DR77].

This research was supported by National Science Foundation Grant ECS81-05968.

1984 IEEE Computer Software & Applications Conference

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 let \( f(P_i) \) be the value of the best solution obtained by evaluating all the 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:
(a) \( g \) is a lower-bound estimate of \( f \);
(b) \( g \) is exact when \( P_i \) is feasible;
(c) 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, the solution value becomes the incumbent \( z \). The incumbent represents the best solution obtained so far in the process. In minimization problems if the lower bound of a subproblem, \( P_i \), exceeds the value of the incumbent, \( P_i \) can be pruned because it will not lead to an optimal solution. The decomposition process continues until all subproblems are either expanded or eliminated. In other words, \( P_i \) is terminated during the computation if:
\[
g(P_i) \geq z
\]

The above elimination rule for obtaining an exact optimal solution can be relaxed in order to obtain a suboptimal solution with a guaranteed accuracy [LAW66]. Suppose it were decided that a deviation of 10% from the optimum was tolerable. If a feasible solution of 150 is obtained, 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% from 150. This technique significantly reduces the amount of intermediate storage and time needed to arrive at a suboptimal solution. Define an allowance function \( \epsilon(z) \): \( \mathbb{R} \to \mathbb{R} \) (set of reals) such that \( P_i \) is terminated if:
\[
g(P_i) \geq z - \epsilon(z)
\]

The final incumbent value \( z_f \) that is obtained by using the modified lower-bound test deviates from the optimal solution value \( z_0 \) by [IB76a]:
\[
z_f - \epsilon(z_f) \leq z_0 \leq z_f
\]

Examples of often used allowance functions are:
\[
\epsilon(z) = \frac{z_f - z_0}{z_f} \geq 0, z_0 \geq 0 \text{ (relative error deviation)} \tag{4}
\]
\[
\epsilon(z) = \frac{z_f - z_0}{1 + \epsilon} \geq 0, z_0 \geq 0 \text{ (relative error deviation)} \tag{5}
\]
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, the algorithm is called a breadth-first search. If the list is maintained in a last-in/first-out order, the algorithm is called a depth-first search. Lastly, if the list is maintained in increasing order of lower bounds, the algorithm is called a best-first search. The B&B algorithm may be summarized as:

\[
\begin{align*}
\text{incumbent} &= 0; \\
\text{list of subproblems} &= \{P_0\}; \\
\text{while list of subproblems is not empty do} & \quad \text{apply selection rule to list of subproblems;} \\
& \quad \text{expand selected subproblem by the branching rule;} \\
& \quad \text{for } P \in \text{set of son subproblems generated} & \quad \text{if subproblem } P \text{ is a feasible solution then} \\
& \quad & \quad \text{update incumbent;} \\
& \quad & \quad \text{apply elimination rule; } \\
& \quad & \quad \text{else if value } < z^*(P) \text{ then} \\
& \quad & \quad \quad \text{insert subproblem } P \text{ into the list of subproblems.}
\end{align*}
\]

To illustrate the B&B algorithm, the evaluation of an integer-programming problem [MIT70] is shown here. Integer-programming problems may be expressed as a constrained optimization:

\[
\begin{align*}
\text{Minimize} & \quad CX \\
\text{subject to} & \quad AX \geq B \\
& \quad X^T = (x_1, x_2, \ldots, x_n) \\
& \quad x_i \geq 0, i = 1, 2, \ldots, n;
\end{align*}
\]

A, B, C are constant matrices.

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

One approach to the problem is the following. Apply the dual simplex method to a subproblem and solve it as a linear program. If the optimal solution is integral, a feasible solution has been generated; otherwise, create two new subproblems as follows. Choose a variable that has a non-integer value (say \(x_1 = 4.4\)) and restrict that variable to the next lower integral value for one problem \(x_1 \leq 4\) or \(x_1 \leq 4\) and to the next higher integral value \(x_1 \geq 4\) or \(x_1 \geq 5\) 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 \(x_i \geq a_i\) was increased. The down penalty is similar, except that it is associated with the constraint \(x_i \leq a_i\). 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.

Figure 1(b) shows the B&B tree for the problem in Figure 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 \(x_1 = 1\). The lower bounds are calculated as shown in Figure 1(b). The dual simplex method is applied again to the subproblem with the smallest lower bound, and a feasible solution is generated with all variables having integral values. This constitutes an optimal solution since the lower bound of the remaining subproblem is greater.

It has been shown that a best-first search has a predictable and the best time efficiency if \(g[P_i] = f^*\) when \(P_i\) is not an optimal solution node and \(f^*\) is the optimal-solution value [L84]. This result has been derived in conjunction with a computer model that consists of a CPU and a main memory. In many computers the main memory is limited in size and has to be extended in size by a secondary memory. Although the best-first search requires the minimum execution time, it needs an exponential amount of memory space and may use the secondary memory for extended storage. In this case...
there is a mismatch in access efficiency and hence a degradation in performance.

In a depth-first search the list of subproblems is stored in a last-in/first-out stack. The size of this stack is the maximum depth of the search tree. For many optimization problems, this is equal to the number of variables in the problem and is reasonably small. A depth-first search can, thus, be run entirely in main memory and avoid the delays of the secondary storage. However, the saving in time may be offset by the increased number of subproblems to be examined. A breadth-first search has a similar behavior. On the other hand, a best-first search requires a large memory space and is generally efficient in the number of subproblems expanded. Hence if an efficient memory management system is available so that all the secondary-storage accesses are overlapped with the computations, a best-first search will be more efficient.

Several researchers have addressed the problem of improving the performance of non-search algorithms in a computer with a two-level memory hierarchy managed by a paging virtual-memory system. These include rules for enhancing locality [MOR68], automatic restructuring of programs [BAE76, HAT77, FER74, FER75, FER76], matrix manipulation [MCK69, FIS70], sorting [BRO70, ALA84] and database operations [SMI78].

In this paper we have studied the efficient evaluation of best-first searches in a computer model with a two-level memory hierarchy that comprises of a CPU, a main memory, a slower secondary memory and the corresponding secondary-memory controller. The CPU and the secondary-memory controller overlap in processing. The time required for an algorithm is measured by the expected completion time on the two-level memory-computer model when the search process is run in a single-user mode.

2. DIRECT IMPLEMENTATION OF BEST-FIRST SEARCHES

One implementation of the best-first search uses a list of subproblems and a priority queue of pointers to these subproblems. In each iteration, the subproblem with the minimum lower bound is deleted from the list. This generates new subproblems that are inserted back. The priority queue of pointers is used to maintain the ordering required for a best-first search since restoring a sorted list of pointers is less costly than maintaining a sorted list of subproblems.

As subproblems are not ordered by lower bounds in the list, the subproblem chosen for expansion is equally likely to be anywhere in the list. There will be a wait if the chosen subproblem is in the secondary memory. On the other hand, once the main memory is full, subproblems must be moved to the secondary storage before the process can continue. This introduces another wait. To avoid these waits, we have investigated two approaches to support best-first searches: modification of the virtual memory that manages the distribution of subproblems between the main memory and the secondary memory, and modification of the search algorithm. In this section we present the implementation of the algorithm in a modified virtual memory.

The proposed virtual-memory system is depicted in Figure 2. Subproblems in the secondary storage are organized as a B*-tree.2 Each leaf of the tree is a page and contains one or more subproblems. Since the non-terminal 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 disk accesses required to access a subproblem in the secondary storage.

The main memory contains a partial list of subproblems and a heap of pointers to subproblems in the partial list. Newly generated subproblems are inserted into the memory list. When the main-memory space is exhausted, subproblems are moved to the B*-tree in the secondary storage. In a best-first search, the subproblem with the smallest lower bound is always expanded. It is prohibitively expensive to compare the sub problem with the smallest lower bound in the memory list with the subproblem having the smallest lower bound in the B*-tree each time a subproblem is expanded. This can be solved by keeping a portion of the first B*-tree page in the main memory.

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

---

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 A B-tree of order m is a search tree that is either empty or satisfies the following properties [COM79]: (i) the root node has at least two children; (ii) each node contains at least m keys and m + 1 pointers; and (iii) each node contains at most 2m keys and 2m + 1 pointers. A B*-tree is a 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.
use a bit map in main memory to show the status of blocks of all the pages. 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 operating system depends on the replacement algorithm and the page size [YU83]. 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. Analysis and simulation results have shown that for integer-programming problems, between 70% to 90% of the subproblems in main memory with the largest lower bounds should be replaced each time. For knapsack problems, between 30% to 50% of the subproblems should be replaced. The page size should be chosen to minimize the disk traffic. A suitable page size for both integer-programming and knapsack problems is found to be between 60% to 90% of the maximum size of the subproblem list in main memory.

3. MODIFIED BEST-FIRST B&B ALGORITHM

Although effective, modification of the virtual-memory system is problem dependent and is thus inflexible. On the other hand, the modification of the search strategy is easier to be managed by the designers. In this section we describe a modified B&B algorithm that improves the locality of access to the secondary storage without sacrificing the performance.

3.1. Description of the Algorithm

Let \( g(P_0) \) be the lower bound of the initial problem. For any subproblem \( P_i \), let \( h(P_i) = g(P_i) - g(P_0) \). As depicted in Figure 3, the range of possible \( h \) is partitioned into a disjoint region, \( \{[u_0, u_1], [u_1, u_2], \ldots, [u_{n-1}, u_n] \} \) where \( u_0 = 0 \) and \( u_n = \infty \). Each region \([u_i, u_{i+1}]\)

\[0 \leq i \leq n-1,\] is implemented as a stack. The list in each stack is usually large and may have to be stored partially in the secondary memory. It is thus inefficient to access a subproblem with the smallest lower bound in a list due to high overhead for secondary storage accesses. To solve this problem, subproblems within a list are not ordered by lower bounds and are expanded in a depth-first manner. To minimize the access overhead, the top portion of each stack resides in a buffer in the main memory while the remaining part resides in the secondary memory. To increase the overlap between the main-memory and secondary-storage accesses, there are disk buffers for storing temporary data.

The modified selection rule chooses the subproblem on top of the non-empty buffer with the smallest index for expansion. This is the buffer that contains the subproblem in main memory with the smallest lower bound. Since subproblems within a list are not sorted, the lower-bound elimination rule also has to be modified. Let the value of the incumbent be \( z \) and the allowance function be \( \epsilon(z) \). Suppose that \( z - \epsilon(z) \) lies in the range of stack \( k \), then all stacks with indices greater than \( k \) are eliminated. There may be subproblems in stack \( k \) whose lower bounds are greater than \( z - \epsilon(z) \) but are not eliminated immediately. These subproblems will be eliminated when they are moved to the main memory during the expansion of list \( k \). As a result, it will be necessary to carry out the lower-bound test on each selected subproblem before it is expanded. The modified B&B algorithm is summarized as follows:

\[
\text{incumbent} = 0; \\
\text{list of subproblems} = \{P_0\}; \\
\text{while number of subproblems} > 0 \text{ do} \\
\quad \text{let } k \text{ be the smallest index of the non-empty lists;}
\]

\[
\text{if buffer} k \text{ is empty and no load request is outstanding then} \\
\quad \text{initiate load from stack} k \text{ into buffer} k; \\
\text{if all buffers are empty then} \\
\quad \text{wait until load from stack} k \text{ into buffer} k \text{ is completed;}
\]

\[
\text{apply modified selection rule;}
\]

\[
\text{if lower bound of selected subproblem} < z - \epsilon(z) \text{ then} \\
\quad \text{if all buffers are full then} \\
\quad \quad \text{wait until subproblems stored;}
\]

\[
\text{expand selected subproblem;}
\]

\[
\text{if expanded subproblem is a feasible solution then} \\
\quad \text{update incumbent;}
\]

\[
\text{apply modified elimination rule;}
\]

\[
\text{else if value} < z - \epsilon(z) \text{ then} \\
\quad \text{push subproblems into corresponding stacks;}
\]

\[
\text{for} i \in \text{non-empty buffer with index} > k \\
\quad \text{if buffer} i \text{ is full then} \\
\quad \quad \text{initiate store from buffer} i \text{ to stack} i
\]

More subproblems may be expanded in the modified B&B algorithm than in a direct implementation. Let the value of the optimal solution falls within the range of stack \( k \). Since subproblems are not ordered by lower bounds in a stack, subproblems in stack \( k \) with lower bounds greater than the value of the optimal solution may be expanded if they are smaller than the value of the incumbent. Some subproblems from stacks with indices greater than \( k \) may also be expanded due to the modified selection rule. These subproblems are never expanded in a best-first search.

The increase in the number of subproblems expanded depends on the assignment of boundaries between adjacent stacks. Suppose we have an estimate
of the optimal solution, \( v_F \), a lower limit, \( a_L \), and an upper limit, \( u_L \), on the accuracy of the estimate. A simple argument shows that there should be at least three stacks. When three stacks, \( i = 0, 1, 2 \), are used, the stack boundaries should be \( u_0 = 0 \), \( u_1 = v_F/a_L - \varepsilon (v_F/a_U) \), \( u_2 = v_F/a_L - \varepsilon (v_F/a_U) \) and \( u_3 = \infty \). The optimal solution resides in Stack 1, and all subproblems in it must be examined. To limit the number of subproblems examined when this happens, Stack 1 should further be partitioned into smaller stacks. In the next section we derive an analytical procedure for determining the boundaries of stacks that will minimize the average number of subproblems examined for any given number of stacks. This will also serve as a guide in determining the number of stacks to be used. We will only consider the case where \( \varepsilon (z) \) is zero. The results for the case when \( \varepsilon (z) \) is not zero can be obtained by replacing the terms \( v_F/a_U \) and \( v_F/a_L \) in the derived equations by \( v_F/a_U - \varepsilon (v_F/a_U) \) and \( v_F/a_L - \varepsilon (v_F/a_L) \) respectively.

### 3.2 Analysis of Stack Boundaries

Let the main memory allocated to the modified R&B algorithm be \( b \), and the size of a buffer be \( n_b \). Then the number of buffers (and hence the number of stacks), \( n \), will be \( \lfloor b/n_b \rfloor \). As a simplification, when a stack is terminated, its buffer is not reallocated to another stack. When a buffer is full, its contents are copied to a disk buffer of size \( n_b \) and subsequently transferred to the stack on disk as a single disk cluster. It is assumed that there is always a free disk buffer.

Let \( v_A \) be the value of the optimal solution. Assume that \( v_A \) is uniformly distributed over the range \( (v_F/a_U, v_F/a_L) \). Let \( (u_l, v_A=v_l) \), \( 0 \leq l \leq n \), be the optimal assignment of \( u_l \) when \( v_A=v_l \), and let \( (w_l, v_A=v_l) = (u_l+1, v_F - v_l) - (u_l, v_A=v_l) \) be the width of stack \( l \). Suppose that \( v \) falls within the range of stack \( k \). Let \( M_k \) be the total number of subproblems written to stack \( k \) in the secondary storage for the duration of the R&B process. By assuming that:

1. The increase in lower bound when a subproblem is expanded is a random variable that is exponentially distributed with parameter \( \lambda \) (the distribution has been verified experimentally to be gamma with \( \alpha < 1 \) for integer-programming and knapsack problems [WAI92]: the exponential distribution is used here as an approximation),
2. Each subproblem expansion generates \( s \) subproblems,
3. All expanded subproblems are independent of each other,

the expected value of \( E[M_i] \) given that \( v_A=v \) has been shown in Appendix A (Eq. (A-12)) to be equal to:

\[
E[M_i | v_A=v] = \begin{cases} 
(1 - e^{-\lambda(v_F - v)}) s e^{-\lambda(v_F - v)} & \text{for } 1 \leq i \leq k \\
(1 - e^{-\lambda(v_F - v)}) s e^{-\lambda(v_F - v)} e^{\lambda v} & \text{for } n+1 \geq i \geq k 
\end{cases}
\]

(7)

The boundaries \( u_l, i=1, ..., n-2 \), should be chosen so that the disk overhead for each stack between \( v_F/a_U \) and \( v_F/a_L \) when the process terminates is approximately the same. This will minimize the expected number of subproblems expanded when \( v_A \) is uniformly distributed over the range \( (v_F/a_U, v_F/a_L) \). A subproblem in stack \( i \), \( i \leq k \), undergoes two disk transfers since the subproblem has to be written from buffer \( i \) into stack \( i \) and later read from stack \( i \) into buffer \( i \). A subproblem in stack \( i \), \( i > k \), undergoes only one disk transfer since the subproblem is never read from stack \( i \) into buffer \( i \). The disk overhead for the stacks will be the same if:

\[
E[M_i | v_A=v] = \cdots = E[M_{n-1} | v_A=v] = \cdots = E[M_n | v_A=v] = 2 E[M_{n+1} | v_A=v] = \ldots = 2 E[M_{n-1} | v_A=v]
\]

(8)

This will be satisfied if the assigned boundaries satisfy:

\[
1 - e^{-\lambda(u_l|v_A=v)} = e^{-\lambda(u_l|v_A=v)} e^{\lambda (v_F - v)}
\]

for \( i \leq k \)

\[
1 - e^{-\lambda(u_l|v_A=v)} = e^{-\lambda(u_l|v_A=v)} e^{\lambda (v_F - v)}
\]

for \( i = k + 1 \)

(9)

\[
1 - e^{-\lambda(u_l|v_A=v)} = e^{-\lambda(u_l|v_A=v)} e^{\lambda (v_F - v)}
\]

for \( i \geq k + 2 \)

In addition, since \( u_1 = v_F/a_L \), \( u_n-1 = v_F/a_U \), we have:

\[
\sum_{i=1}^{n-1} (w_l, v_A=v_l) = v_F - \frac{v_F}{a_L}
\]

(10)

An iterative scheme may be used to compute the values of \( (w_l, v_A=v_l), i=1, ..., n-2, \) and \( (u_l|v_A=v_l), i=2, ..., n-2, \) that will satisfy Eq's (9) and (10).

Given the values of \( (u_l|v_A=v_l), i=2, ..., n-2, \) the assumption on the distribution of \( v_A \) can then be used to determine the values of \( u_l \):

\[
u_l = \int_{v_F/a/U}^{v_F/a/L} \frac{1}{(v_F/a_U - v_F/a_L)} dv
\]

(11)

It is not possible to obtain a closed form for Eq. (11). Instead, a discrete approximation may be used in which \( v_A \) is chosen from \( v_i = v_F - \frac{v_F}{a_L} \) with probability \( \frac{1}{M} \).

The expected disk overhead due to the assignment \( (u_l, v_A=v_l) \) may be evaluated. Let \( E[T_D | v_A=v] \) be the expected disk overhead given that \( v_A=v \). Suppose that \( u_k \leq v \leq u_{k+1} \). Let \( E[P_i | v_A=v], i=2, ..., n-1, \) be the number of disk clusters that make up stack \( i \).

\[
E[P_i | v_A=v] = \frac{E[M_i | v_A=v]}{n_b}
\]

(12)

The assumption on the distribution of \( v_A \) can then be used to determine the expected disk overhead based on the assignment \( (u_1, ..., u_{n-1}) \).
the above equation subject to the constraint of Eq. (10). The boundaries can then be computed by:

$$u_i = \frac{1}{v_{E} / a_{L} - v_{E} / a_{U} + 1} \sum_{v_{A} = v}^{v_{E}} E[u_{i} | v_{A} = v]$$

As in Section 3.2, the expected disk overhead when

$$v_{A} = v, E[T_{D} | v_{A} = v],$$

may be computed for $v = v_{E} / a_{L}, v_{E} / a_{U} + 1, \ldots, v_{E} / a_{L}$, and the expected disk overhead,

$$E[T_{D}]$$

is:

$$E[T_{D}] = \frac{1}{v_{E} / a_{L} - v_{E} / a_{U} + 1} \sum_{v_{A} = v}^{v_{E}} E[T_{D} | v_{A} = v]$$

3.4 Simulation Results

We have carried out simulations for the modified B&B algorithm with integer-programming and vertex-cover problems. The effectiveness of the schemes developed in Sections 3.2 and 3.3 may be evaluated by observing $M_{1}$, the actual number of subproblems written to stack $i$ in the secondary storage when the B&B process terminates. When $n$ stacks are used and stack $k$ contains the optimal solution, the values of $M_{1}$ should satisfy:

$$M_{1} \approx \cdots M_{k} = 2 M_{k+1} \approx \cdots = 2 M_{n-2}$$

Some results are shown in Table 1 for integer-programming problems and in Table 2 for vertex-cover problems. We have presented these results in the form of the mean, standard, maximum and minimum deviations of the set of values $M_{1}, \ldots, M_{k}, 2 M_{k+1}, \ldots, 2 M_{n-2}$. It can be seen that the deviations from the mean are reasonable. In contrast, if a simple scheme such as equal-width stacks had been used, the values would show an exponential increase.

Our simulations suggest that for both the integer-programming and vertex-cover problems, three stacks should be used to minimize the disk overhead. This can be clearly seen from Table 3(a) that shows the variation of disk overhead with the number of stacks for five vertex-cover problems. The disk overhead calculated is based upon a disk with an average delay of 30.333 msec.

Table 1. Statistical measures for the number of subproblems written to the stacks for five 20-variables, 20-constraints integer programming problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Measure</th>
<th>4 stacks</th>
<th>5 stacks</th>
<th>6 stacks</th>
<th>7 stacks</th>
<th>8 stacks</th>
<th>9 stacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mean</td>
<td>570.0</td>
<td>424.0</td>
<td>312.3</td>
<td>240.0</td>
<td>194.2</td>
<td>167.0</td>
</tr>
<tr>
<td></td>
<td>std dev</td>
<td>248.0</td>
<td>75.5</td>
<td>50.3</td>
<td>67.3</td>
<td>74.1</td>
<td>74.4</td>
</tr>
<tr>
<td></td>
<td>max dev</td>
<td>248.0</td>
<td>88.0</td>
<td>75.7</td>
<td>88.0</td>
<td>138.8</td>
<td>164.0</td>
</tr>
<tr>
<td></td>
<td>min dev</td>
<td>248.0</td>
<td>12.9</td>
<td>14.7</td>
<td>23.0</td>
<td>2.367</td>
<td>17.9</td>
</tr>
<tr>
<td>2</td>
<td>mean</td>
<td>423.5</td>
<td>302.9</td>
<td>227.5</td>
<td>177.6</td>
<td>147.2</td>
<td>158.3</td>
</tr>
<tr>
<td></td>
<td>std dev</td>
<td>60.5</td>
<td>65.6</td>
<td>135.0</td>
<td>101.4</td>
<td>211.8</td>
<td>232.7</td>
</tr>
<tr>
<td></td>
<td>max dev</td>
<td>60.5</td>
<td>69.0</td>
<td>5.0</td>
<td>9.5</td>
<td>6.2</td>
<td>7.3</td>
</tr>
<tr>
<td></td>
<td>min dev</td>
<td>60.5</td>
<td>5.0</td>
<td>2.5</td>
<td>3.5</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>3</td>
<td>mean</td>
<td>318.0</td>
<td>144.5</td>
<td>110.5</td>
<td>84.6</td>
<td>64.0</td>
<td>55.0</td>
</tr>
<tr>
<td></td>
<td>std dev</td>
<td>89.0</td>
<td>15.8</td>
<td>14.2</td>
<td>27.7</td>
<td>27.9</td>
<td>27.2</td>
</tr>
<tr>
<td></td>
<td>max dev</td>
<td>89.0</td>
<td>33.5</td>
<td>25.3</td>
<td>35.4</td>
<td>38.0</td>
<td>37.0</td>
</tr>
<tr>
<td></td>
<td>min dev</td>
<td>89.0</td>
<td>7.7</td>
<td>2.5</td>
<td>10.6</td>
<td>8.0</td>
<td>9.0</td>
</tr>
<tr>
<td>4</td>
<td>mean</td>
<td>482.0</td>
<td>325.7</td>
<td>240.0</td>
<td>193.2</td>
<td>171.0</td>
<td>231.7</td>
</tr>
<tr>
<td></td>
<td>std dev</td>
<td>304.0</td>
<td>112.8</td>
<td>64.6</td>
<td>45.5</td>
<td>48.1</td>
<td>51.2</td>
</tr>
<tr>
<td></td>
<td>max dev</td>
<td>304.0</td>
<td>148.7</td>
<td>91.0</td>
<td>55.8</td>
<td>83.0</td>
<td>107.3</td>
</tr>
<tr>
<td></td>
<td>min dev</td>
<td>304.0</td>
<td>24.3</td>
<td>31.0</td>
<td>16.2</td>
<td>8.0</td>
<td>8.7</td>
</tr>
<tr>
<td>5</td>
<td>mean</td>
<td>162.0</td>
<td>227.7</td>
<td>185.2</td>
<td>134.0</td>
<td>148.8</td>
<td>144.3</td>
</tr>
<tr>
<td></td>
<td>std dev</td>
<td>57.0</td>
<td>62.3</td>
<td>24.4</td>
<td>10.0</td>
<td>21.4</td>
<td>27.9</td>
</tr>
<tr>
<td></td>
<td>max dev</td>
<td>57.0</td>
<td>105.5</td>
<td>38.3</td>
<td>33.0</td>
<td>29.8</td>
<td>55.7</td>
</tr>
<tr>
<td></td>
<td>min dev</td>
<td>57.0</td>
<td>9.7</td>
<td>4.3</td>
<td>3.0</td>
<td>0.2</td>
<td>4.3</td>
</tr>
</tbody>
</table>
Table 2. Statistical measures for the number of subproblems written to the stacks for five 80-node vertex-cover problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Measure</th>
<th>4 stacks</th>
<th>5 stacks</th>
<th>6 stacks</th>
<th>7 stacks</th>
<th>8 stacks</th>
<th>9 stacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mean</td>
<td>224.5</td>
<td>805.9</td>
<td>1078.0</td>
<td>625.6</td>
<td>820.2</td>
<td>501.6</td>
</tr>
<tr>
<td></td>
<td>std</td>
<td>153.5</td>
<td>101.9</td>
<td>445.8</td>
<td>223.7</td>
<td>355.5</td>
<td>252.1</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>153.5</td>
<td>228.0</td>
<td>705.0</td>
<td>283.5</td>
<td>498.2</td>
<td>355.6</td>
</tr>
<tr>
<td></td>
<td>min</td>
<td>153.5</td>
<td>95.0</td>
<td>25.0</td>
<td>155.4</td>
<td>10.2</td>
<td>105.8</td>
</tr>
<tr>
<td>2</td>
<td>mean</td>
<td>418.5</td>
<td>512.9</td>
<td>1103.0</td>
<td>928.1</td>
<td>792.1</td>
<td>680.1</td>
</tr>
<tr>
<td></td>
<td>std</td>
<td>272.5</td>
<td>291.4</td>
<td>483.3</td>
<td>460.0</td>
<td>338.3</td>
<td>479.1</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>272.5</td>
<td>412.0</td>
<td>562.0</td>
<td>782.2</td>
<td>792.8</td>
<td>689.2</td>
</tr>
<tr>
<td></td>
<td>min</td>
<td>272.5</td>
<td>2.0</td>
<td>185.0</td>
<td>82.8</td>
<td>47.8</td>
<td>152.1</td>
</tr>
<tr>
<td>3</td>
<td>mean</td>
<td>250.0</td>
<td>135.7</td>
<td>1201.3</td>
<td>1254.2</td>
<td>1085.0</td>
<td>985.9</td>
</tr>
<tr>
<td></td>
<td>std</td>
<td>252.0</td>
<td>284.9</td>
<td>520.2</td>
<td>570.4</td>
<td>517.1</td>
<td>509.0</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>252.0</td>
<td>396.7</td>
<td>710.3</td>
<td>801.2</td>
<td>798.0</td>
<td>903.1</td>
</tr>
<tr>
<td></td>
<td>min</td>
<td>252.0</td>
<td>137.3</td>
<td>284.5</td>
<td>39.8</td>
<td>97.0</td>
<td>189.9</td>
</tr>
</tbody>
</table>

Table 3. Effect of the number of stacks on (a) the actual and predicted disk overheads and (b) the number of subproblems expanded for vertex-cover problems.

<table>
<thead>
<tr>
<th>Number of Stacks</th>
<th>Disk overhead (ms)</th>
<th>Number of subproblems expanded</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Actual</td>
<td>Predicted</td>
</tr>
<tr>
<td>3</td>
<td>303.0</td>
<td>455.0</td>
</tr>
<tr>
<td>4</td>
<td>252.0</td>
<td>284.9</td>
</tr>
<tr>
<td>5</td>
<td>252.0</td>
<td>284.9</td>
</tr>
</tbody>
</table>

(a)

(b) and a capacity of 20 Kbytes per track. Table 3(b) shows the number of expanded subproblems for the same vertex-cover problems. As expected, the maximum number of subproblems are expanded when three stacks are used. However, the number is not excessively large as compared to those when more stacks are used. Similar results are shown in Tables 4(a) and 4(b) for five integer-programming problems. Tables 3(a) and 4(a) also compare the predicted disk overhead with the actual simulated disk overhead. For the vertex-cover problems, the predicted results are fairly close to the actual simulated results in spite of the many simplifying assumptions made. For the integer-programming problems, the predicted results are much smaller than the actual simulated results. One possible reason is the use of an exponential distribution instead of the more accurate gamma distribution for the increase in lower bounds when a subproblem is expanded. However, it is also seen that most of the predicted values differ from the actual values by factors that do not vary greatly for each problem. In addition, both the predicted and actual values increase with the number of stacks. This allows us to use the analytical expressions to predict the number of stacks to be used and their boundaries.

Our simulations also show that the performance of the modified B&B algorithm appears to be relatively insensitive to the uncertainty in the estimate of the optimal solution value for the integer-programming and vertex-cover problems. This increases the robustness of our method because the optimal solution value may be difficult to estimate before the algorithm terminates.

The modified B&B algorithm for the integer-programming and vertex-cover problems is easy to implement when three stacks are used. Only an estimate of the value of the optimal solutions and the bounds of accuracy of this estimate are necessary. For other problems, it will be necessary to study sample problems to obtain an indication of the number of stacks that should be used. If more than three stacks are to be used, then it will be necessary to actually run the algorithm for a short period (say, a few hundred iterations) to obtain an esti-
mate on the distribution of the increase in lower bounds and the branching factor. The stack boundaries may then be computed and the problem solved by the modified B&B process.

4. COMPARISON OF VARIOUS SCHEMES

Simulation results for two 17-variable, 17-constraint integer-programming problems are shown in Figures 4 and 5. Each subproblem requires a memory size of 1476 bytes. A main-memory size of 128 Kbytes (10% of the virtual space) is assumed. Only the results with the optimal page sizes are shown for the modified virtual-memory scheme and the direct implementation. With large pages, there is a 100-time reduction of disk overhead for the modified B&B search as compared to a direct implementation, and a 60-time reduction as compared to a modified virtual-memory scheme.

Some simulation results for 80-node vertex-cover problems are shown in Figures 6 and 7. Each subproblem requires 330 bytes, and a main memory of 128 Kbytes is assumed. The modified B&B algorithm exhibits close to a 100-time reduction of the disk overhead as compared to a direct implementation and close to a 10-time reduction as compared to a modified virtual-memory scheme.

In conclusion, modification of the virtual memory to tailor to the characteristics of the search algorithm is effective but inflexible. The modification of the search algorithm to enhance the amount of locality exhibited is found to be a viable approach.

**Figure 4** Simulation results for B&B algorithms on a 17 by 17 integer-programming problem (main memory size is 128 Kbytes).

**Figure 5** Simulation results for B&B algorithms on a 17 by 17 integer-programming problem (main memory size is 128 Kbytes).

**Figure 6** Simulation results for B&B algorithms on a 80 node vertex-cover problem (main memory size is 128 Kbytes).
Figure A-1 The model of the B&B process under best-first search.

been shown in [WAH82] that the expected values of \( N(x) \) and \( M(x) \) are:

\[
E[N(x)] = \frac{\lambda}{s-1} \sum_{i=1}^{s-1} e^{(s-1)x} - \frac{1}{s-1} \quad (A-1)
\]

and

\[
E[M(x)] = s E[N(x)] + 1 \quad (A-2)
\]

Let \( \delta(P_i) \) be the optimal solution, and \( \delta(P_i) = v \). The total number of subproblems expanded is \( N(v) \). Let \( \delta(P_i) \) be identically and independently distributed. Suppose that \( \delta(P_i) \) has the value \( \delta_i \). The expected number of random variables \( \delta_i \) with value \( \delta_i \leq x \) given \( v \) and \( N(v) \), \( E[K(x)|v,N(v)] \), is:

\[
E[K(x)|v,N(v)] = N(v) P(x|v,N(v)) \quad (A-3)
\]

Without loss of generality, let \( z_1 \leq \ldots \leq z_{N(v)} \). Assuming that the expanded subproblems are independent of each other, \( \delta(P_i) \) may be approximated by \( z_i \) provided that:

\[
E[K(x)|v,N(v)] = E[N(x)] \quad (A-4)
\]

Substituting Eq's (A-1) and (A-3) into Eq. (A-4) yields:

\[
F(x|v,N(v)) = \frac{E[N(x)]}{N(v)} \quad (A-5)
\]

Differentiating Eq. (A-5) yields the density function:

\[
f(x|v,N(v)) = \frac{1}{N(v)} \left[ \frac{e^{(s-1)x}}{s-1} - \frac{e^{(s-1)x}}{s-1} \right] \quad (A-6)
\]

Let the front wall be at position \( x \leq u_1 \). The expected number of subproblems generated will be \( E[M(x)] \), and the expected number of subproblems expanded will be \( E[N(x)] \). Let \( P_i \) be one of the \( E[M(x)] \) subproblems generated and \( \Pr(u_1 \leq \delta(P_i) < u_{i+1} | v, N(v)) \) be the probability that \( \delta(P_i) \) lies within \([u_i, u_{i+1})\). Out of the
E(M(x)) generated subproblems, the expected number of subproblems in the region [u_i, u_i+1] given v and N(v), E(W(x)|v, N(v)), is given by:
\[ E[W(x)|v, N(v)] = E[M(x)] Pr(u_i \leq \delta(P) < u_i+1 | v, N(v)) \]
\[ = (s E[N(x)] + 1) Pr(u_i \leq \delta(P) < u_i+1 | v, N(v)) \]
\[ = s E[N(x)] Pr(u_i \leq \delta(P) < u_i+1 | v, N(v)) \] (A-7)

Let P_k be a subproblem generated by the expansion of P_h and I(P_k) be the increase in lower bound due to this expansion.
\[ \delta(P_k) = \delta(P_h) + I(P_k) \approx Z_h + Y_k \] (A-8)
Since I(P_k) has been assumed to be an exponentially distributed random variable,
\[ Pr(u_i \leq \delta(P_k) < u_i+1 | Z_h = x, v, N(v)) \]
\[ = \int_{x}^{x+1} \lambda e^{-\lambda y} dy = e^{-\lambda u_i} - e^{-\lambda u_i+1} \] (A-9)
Unconditioning and applying Eq. (A-6), we have:
\[ Pr(u_i \leq \delta(P_k) < u_i+1) \]
\[ = \int_{0}^{x} \left( e^{-\lambda u_i} - e^{-\lambda u_i+1} \right) \frac{\lambda}{N(v)} \frac{1}{\lambda} \frac{1}{s-1} e^{-1}(s-1) dz \] (A-10)
Substituting Eq. (A-10) into Eq. (A-7) yields:
\[ E[W(x)|v, N(v)] \approx \int_{0}^{x} \left( e^{-\lambda u_i} - e^{-\lambda u_i+1} \right) s^2 \lambda e^{-\lambda v} dz \]
\[ = s e^{-\lambda u_i} \left( 1 - e^{-\lambda w} \right) \left( e^{-\lambda v} - 1 \right) \] (A-11)

Eq. (A-11) may be used to derive approximate expressions for the number of subproblems written into stack i, M_i, in the modified B&B algorithm. If u_i \leq v then M_i is the number of subproblems in stack i when it becomes the current stack (the first non-empty stack with the minimum index) since at this point, subproblems in stack i are expanded in a depth-first fashion and no new subproblems will be written to stack i in the secondary memory. Thus, E[M_i|v] \approx E[W(u_i)|v, N(v)] if the number of subproblems chosen for expansion from buffer i before stack i becomes the current stack is small. If u_i > v then M_i is the number of subproblems in stack i when stack i is terminated. If no feasible solution is generated before the optimal solution is found, then E[M_i|v] \approx E[W(v)|v, N(v)]. For problems such as integer programming and vertex cover, it has been found that the number of feasible solutions generated is small, and hence the above approximations are good. When the number of feasible solutions generated before the process terminates is significant as in the knapsack problems, then E[M_i|v] < E[W(v)|v, N(v)] and the approximations do not hold. Thus,
\[ E[M_i|v] = \begin{cases} E[W(u_i)|v, N(v)] & u_i \leq v \\ E[W(v)|v, N(v)] & u_i > v \end{cases} \]
\[ = \begin{cases} s e^{-\lambda u_i}(1 - e^{-\lambda w})(e^{-\lambda u_i} - 1) & u_i \leq v \\ s e^{-\lambda u_i}(1 - e^{-\lambda w})(e^{-\lambda v} - 1) & u_i > v \end{cases} \]
\[ = \begin{cases} (1 - e^{-\lambda w}) s e^{-\lambda u_i} & u_i \leq v \\ (1 - e^{-\lambda w}) s e^{-\lambda v} & u_i > v \end{cases} \] (A-12)

REFERENCES


