On a Search Problem Related to Branch-and-Bound Procedures

R. M. Karp,* M. Saks,** A. Wigderson***

0. INTRODUCTION

Branch-and-bound procedures are commonly used in practice for the solution of NP-hard combinatorial optimization problems (see e.g., [PS, LW]). We describe a model that captures the essential elements common to all branch-and-bound procedures. Storage limitations in these procedures raise in a natural way an interesting search problem. We present deterministic and probabilistic algorithms for this problem. These imply that branch-and-bound can be performed with very small storage and only slightly superlinear time.

In section 1 we describe the search problem and state our results. In section 2 we describe the algorithms. Section 3 elaborates on the relation of our search problem to branch-and-bound procedures.

1. A SEARCH PROBLEM

The input will be a valued, ordered, infinite binary tree T. By this we mean

(*i*) every vertex $v \in T$ has a left child and a right child, denoted $L(v)$ and $R(v)$ respectively. The root of T is r.

(*ii*) every vertex has a value, $val(v)$, and the function $val: T \to N$ satisfies $val(v) \leq val(L(v))$ and $val(v) \leq val(R(v))$. (In other words, values along any path from the root increase, so the tree is a heap).

(For simplicity we shall assume that values on the tree are distinct, i.e. $val(v) \neq val(u)$ for each $u,v \in T$, $u \neq v$. This assumption does not affect any of the results).

*University of California and MSRI, Berkeley, CA

**Bell Communications Research, Morristown, NJ and Department of Mathematics, Rutgers University, New Brunswick, NJ

***MSRI, Berkeley, CA and Institute of Mathematics and Computer Science, The Hebrew University, Jerusalem

0272-5428/86/0000/0019$01.00 © 1986 IEEE

The computation is performed by a RAM M (e.g. AHU) that "explores" the tree. At any time step M resides at a vertex $v \in T$. Only then can it read $val(v)$. As atomic steps, M may decide to move to either $L(v)$, $R(v)$ or $F(v)$ (the father of v) if it exists. Initially, M is placed at the root r of the tree, and is given a separate integer input n. The task of the computation is to find the n^{th} smallest value in the set $\{val(v) | v \in T\}$. We call this problem $SELECT(n)$.

Every traversal of an edge costs one unit. For simplicity we assume that all other computation is free. (Charging for the arithmetic operations as well will only affect our upper bounds by a constant factor). Hence, the **time** of a computation is defined to be the total number of edge traversals (the length of the tour of M) during the computation. The **space** is defined to be the total number of registers used by M. For a fair account of space we allow a register to contain only one tree value, i.e. these values are atomic and can only be compared. The program, however, may apply the usual arithmetic operations to other integer variables.

For a given (randomized) algorithm A, let $T_A(n)$ and $S_A(n)$ denote, respectively, the (expected) time and space complexity of $SELECT(n)$ for the worst case function $val: T \rightarrow N$.

Before stating our results we recommend that the reader verify the following easy fact: $SELECT(n)$ requires $\Omega(n)$ steps, and can be solved in $O(n^2)$ steps.

Our main results are three algorithms for $SELECT(n)$,

(1) A randomized Las Vegas algorithm B, with
$T_B(n) = n \cdot 2^{O(\sqrt{\log n})}$,
$S_B(n) = O(\sqrt{\log n})$.

(2) A deterministic algorithm C, with $T_C(n) = n \cdot 2^{O(\sqrt{\log n})}$,
$S_C(n) = O(\log^{2.5} n)$.

(3) For every fixed ε, a randomized algorithm D_ε with
$T_{D_\varepsilon}(n) = O(n^{1+\varepsilon})$,
$S_{D_\varepsilon}(n) = O(1)$.

2. ALGORITHMS

Both the deterministic algorithm C and the randomized algorithm B will be variants of algorithm A, described in subsection 2.1. Algorithm A will not be space efficient ($S_A(n) = O(n)$), but will serve to illustrate the ideas for achieving the time bound $T_A(n) = n \cdot 2^{O(\sqrt{\log n})}$. In subsection 2.2 we will show how to reduce space using randomization (Algorithm B). In subsection 2.3 we describe the

deterministic small space algorithm C. In 2.4, we show how to trade space for time.

In this section, the algorithms and the analysis are described informally. However, we add an appendix that contains the precise description of Algorithm A in a PASCAL-like language for scrutiny.

2.1. The Deterministic Algorithm A

This algorithm is best thought of as a branch-and-bound algorithm. Let NTH be the n^{th} smallest value in the tree. The algorithm will proceed in phases. The i^{th} phase will start with lower and upper bounds L_{i-1} and U_{i-1} on NTH (i.e. $L_{i-1} \leq NTH \leq U_{i-1}$) and terminate with better bounds L_i and U_i (i.e. $L_{i-1} < L_i \leq NTH \leq U_i < U_{i-1}$). For the first phase, assume $L_o = val(r)$, $U_o = \infty$.

To understand a phase, we must understand the "roots" of this phase. Let T_i be the tree of all vertices whose values are $\leq L_{i-1}$, (i.e. T_i contains the "good" values found before the i^{th} phase where good means $\leq NTH$). A vertex u is a root for this phase if it is the child of a leaf in T_i and $val(u) \leq U_{i-1}$. Note that the remaining good values to be found must be in subtrees rooted at these roots.

Let $R = \{r_1, r_2, ..., r_s\}$ be the set of roots for phase i. This phase will produce a total of $g = g(n) \leq n$ (a parameter to be determined later) new good values from the subtrees rooted at R.

Assume that somehow we knew that the smallest n_j values in the subtree rooted at r_j are good (i.e. all are $\leq NTH$), and further that $\sum_{j=1}^{s} n_j = g$. Then we could call our algorithm recursively from r_j asking for the smallest n_j from this subtree, for $1 \leq j \leq s$. The complexity of this phase will be then bounded by $|T_i| + \sum_{j=1}^{s} f(n_j)$, where $f(n)$ is the time to find the n^{th} smallest using this algorithm. The overhead $|T_i|$ is for traveling between the roots. After phase i we shall set L_i to be the largest good value found in this phase (U_i is not needed in this hypothetical situation), and proceed to the next phase.

Of course, we do not know the n_j in advance. However, using an observation we shall describe soon, we will be able to bound the total cost of recursive calls to root r_j by $c_1 f(n_j)$, and the total overhead by $c_2 |T_i| \log^2 n \leq c_2 n \log^2 n$, where c_1, c_2 are absolute constants, independent of n. As each phase will provide

$g(n)$ new good values, the number of phases will be about $\frac{n}{g(n)}$, and hence

$$f(n) \leq \frac{c \cdot n}{g(n)} \left[\sum_{\Sigma n_j = g(n)} f(n_j) + n \log^2 n \right]$$

where $c = \max\{c_1, c_2\}$. We will show that for a suitable choice of $g(n)$, there is a constant K such that $f(n) = nK^{\sqrt{\log n}}$ satisfies this recurrence.

The key (simple) observation is that given a value Z, it is easy ($O(n)$ time) to test if it is good (i.e., whether or not $Z \leq NTH$). Perform depth-first search, retreating each time a value greater than Z is found, and count the number of values $< Z$ found, stopping at n. If the counter reaches n, then Z is bad, else it is good. (This is the procedure "good" in the appendix).

Moreover, given s values, $Z_1, Z_2, \ldots, Z_s$, we can test which of them is bad in time $O(n \log s)$, by sorting them and using binary search to find the largest good value amongst them. (This is procedure "bounds" in the appendix).

Now the idea will be to call the algorithm recursively to find the k^{th} smallest in the tree of r_j, $1 \leq j \leq s$, with $k = 1,2,4,8,\ldots$. (This is the inner while loop in procedure "findbest" in the appendix). After each such set of calls the k^{th} best from each r_j, say Z_j will be returned, and we shall test which values on this set $S = \{Z_1, Z_2, \ldots Z_s\}$ are good. For the next iteration, (doubling k) we will proceed only with roots r_j such that Z_j was good, which we call live roots. This has the effect that if r_j eventually supplied n_j good values, the total time spent in recursive calls to it is $f(1) + f(2) + f(4) + \ldots + f(2^t)$, such that $2^{t-1} \leq n_j < 2^t$, and f will satisfy that this sum is bounded by $c_1 f(n_j)$. The overhead in finding after each iteration which roots are alive is $O(n \log n)$, and as there are at most $\log g(n) \leq \log n$ iterations, this time is bounded by $c_2 n \log^2 n$ steps. At the end of the phase, L_i is taken to be the largest good value found so far, and U_i the smallest bad one found so far.

Analysis

$$f(n) \leq \frac{cn}{g(n)} \left[\sum_{\Sigma n_j = g(n)} f(n_j) + n \log^2 n \right]$$

We will choose a convex function f, so

$$f(n) \leq \frac{cn}{g(n)} [f(g(n)) + n \log^2 n].$$

Set $f(n) = n \log^2 n \, h(n)$. Then

$$h(n) \leq c \left[h(g(n)) + \frac{n}{g(n)} \right].$$

$$\leq c \frac{n}{g(n)} + c^2 \frac{g(n)}{g(g(n))} + \ldots + c^t \frac{g^{[t-1]}(n)}{g^{[t]}(n)}$$

where $g^{[t]}(n) = 1$. Given t this expression is minimized when all of its terms are equal, and in this case their common value is $c^{\frac{t+1}{2}} n^{\frac{1}{t}}$. Choosing $t = \sqrt{2 \log_c n}$, corresponding to

$$g(n) = \frac{n}{c^{\sqrt{2\log_c n}}} - 1,$$

we obtain

$$h(n) \leq \sqrt{2\, c\, \log_c n}\ \ c^{\sqrt{2\log_c n}} = 2^{O(\sqrt{\log n})}$$

Hence

$$f(n) = n \log^2 n\ 2^{O(\sqrt{\log n})} = n 2^{O(\sqrt{\log n})}$$

2.2. The Randomized Algorithm B

The deterministic algorithm A maintains a constant number of variables per level of recursion, except for the set $S = \{Z_1, Z_2, \ldots, Z_s\}$ from which it should extract the live roots for the next iteration (doubling k). Algorithm A sorted S and then used binary search to find which Z_i are good ($\leq NTH$). We now describe how to use randomness to perform this "binary search" using only a constant number of registers, in expected time $O(n \log s)$.

First, assume that $r_1, r_2, \ldots, r_s$ are ordered in the way they occur in a depth first traversal of the tree T_i. Then Z_j can be computed given only L_{i-1}, U_{i-1}, j and k in this level of recursion (plus a recursive call to the tree of r_j), simply using a counter to identify the j^{th} root r_j. Furthermore, the procedure "good" to test whether $Z_j \leq NTH$ also requires only one counter.

Now, the binary search is done using random splitters. We keep X_L and X_U, lower and upper bounds on the largest good value in S. Initially, $X_L = -\infty$ and $X_U = \infty$. When $X_L = X_U$ we are done. If not, the following iteration is repeated:

(1) Count the number (say l) of Z_p such that $X_L \leq Z_p \leq X_U$. (This is procedure "roots" in the appendix).

(2) Pick a random j, $1 \leq j \leq l$.

(3) Find the j^{th} root (say r_m) among those roots r_p for which $X_L \leq Z_p \leq X_U$, in the ordering $r_1, r_2, \ldots, r_s$.

(4) If Z_j is good, then $X_L \leftarrow Z_m$ else $X_U \leftarrow Z_m$.

It is easy to see that the expected time until $X_L = X_U$ is $O(\log s)$, and since each goodness test is linear, the expected time to completion is $O(n \log s)$.

The time analysis of algorithm B is identical to that of A, so

$T_B(n) \le nK'^{\sqrt{\log n}}$. At every level of recursion only a fixed number of variables is needed, and since $g(n) < \frac{n}{2^{\sqrt{\log n}}}$ there are at most $O(\sqrt{\log n})$ levels of recursion, so $S_B(n) = O(\sqrt{\log n})$.

2.3 A Small Space Deterministic Algorithm C

In light of the previous section, we can now distill an abstract problem, a small space solution to which will provide a small space algorithm.

We wish to perform binary search on an unordered set $Z_1, Z_2, \ldots, Z_s$, which (restricting ourselves to depth first search) can be accessed only sequentially in left to right order (perhaps several times). This can be done if we have a selection algorithm which finds the i^{th} smallest element in the set for any $1 \le i \le s$. But this is exactly the "external selection" model discussed in [*MP*]. They show, that using only $O(\log^2 s)$ space, selecting the i^{th} smallest from the set $\{Z_1, \ldots, Z_s\}$ takes only $O(\log s)$ left-to-right passes over the set. As $s \le n$ and scanning $Z_1, \ldots, Z_s$ in left to right order takes $O(n)$ time (depth first search), the total overhead in every phase, even if we use $O(\log^2 n)$ variables per level of recursion, is $O(n \log^3 n)$.

So incorporating the Munro-Paterson selection algorithm into Algorithm *A* results in a deterministic algorithm *C* for the tree selection problem with $T_C(n) \le n \cdot K''^{\sqrt{\log n}}$ (same analysis) and $S_C(n) \le O(\log^{2.5} n)$ (each level of recursion requires $O(\log^2 n)$ space).

2.4. Trading Space for Time

The choice of $g(n)$ in the analysis of section 3.1 was made to minimize the running time of the algorithm. Note, however, that $g(n)$ determines also the depth of recursion, and hence the total space used.

The analysis of section 2.1 shows, that for every t, $2 \le t \le \sqrt{\log n}$, if we choose $g(n) = n^{1-\frac{1}{t}}$, we obtain an algorithm D_t whose running time is $T_{D_t}(n) = O(t\, n^{1+\frac{1}{t}} c^t)$, and whose maximum depth of recursion is $O(t)$. Hence if D_t uses randomization in the binary search, also $S_{D_t}(n) = O(t)$ (if D_t were deterministic, space is $O(t \log^2 n)$.) This means that even with constant space, we have for any fixed $\varepsilon > 0$ an algorithm for SELECT(n) whose running time is bounded by $O(n^{1+\varepsilon})$!

3. RELATION TO BRANCH-AND-BOUND PROCEDURES

Branch-and-bound procedures are widely used in practice for the solution of NP-hard combinatorial optimization problems. The main results of the present paper permit branch-and-bound procedures to be implemented with a very small amount of working storage, at the cost of a relatively modest increase in execution time. In order to explain this consequence of our results, we describe the essential elements common to all branch-and-bound procedures.

We consider combinatorial optimization problems of the form: minimize $g(x)$, where x ranges over $\{0,1\}^m$. Each coordinate of x corresponds to a zero-one variable, and the branch-and-bound approach is based on considering certain restrictions of the original problem, obtained by fixing some of these zero-one variables to constant values. A *restriction* is defined as an element $s \in \{0,1,*\}$. The coordinates of s containing a zero or a one are called *fixed*, and those containing a * are called *free*. Associated with any restriction s is the domain $Dom(s)$, consisting of those solutions x which agree with s in all the fixed coordinates; i.e.,

$Dom(s) = \{x \in \{0,1\}^m \mid s_i \neq * \Rightarrow x_i = s_i\}$.

The subproblem associated with restriction s is to minimize $g(x)$, where x ranges over $Dom(s)$. In particular, the original optimization problem is the subproblem associated with the restriction $*^m$, in which every coordinate is free.

Fundamental to each branch-and-bound procedure is a subroutine for computing lower bounds. On the input s this subroutine returns a value $A(s)$ such that $A(s) \leq \min_{x \in Dom(s)} g(x)$. Often, this subroutine derives its lower bound by solving a relaxed version of the subproblem associated with restriction s; for example, the relaxation might replace an integer programming problem by a linear programming problem; this is done by allowing the variables corresponding to free coordinates to range over the interval [0,1], rather than the discrete set $\{0,1\}$. We assume the following monotonicity property of the function $A(s)$: if restriction s' is obtained from restriction s by fixing one or more free coordinates, then $A(s') \geq A(s)$. In other words, the lower bound cannot decrease as the domain of the restriction shrinks.

Certain restrictions are designated as terminal restrictions; every terminal restriction s has the property that the associated lower bound is tight; i.e., $A(s) = \min_{x \in Dom(s)} g(x)$. We assume that there is an efficient pro-

cedure for testing whether a given restriction s is terminal and that, in particular, every restriction in which all coordinates are fixed is terminal.

The final element in a branch-and-bound formulation of a combinatorial optimization problem is a *branching rule*. This rule associates with each nonterminal restriction s, a pair of restrictions $L(s)$ and $R(s)$ (the left child and right children of s) obtained by fixing some free coordinate of s to the values 0 and 1, respectively. In other words, there is some coordinate i such that $s[i] = *$ $L(s)_i = 0$ and $R(s)_i = 1$. For any other coordinate j, $s_j = L(s)_j = R(s)_j$.

The object of a branch-and-bound procedure is to determine $\min_x g(x)$ by finding a terminal restriction s for which $A(s)$ is a minimum. At any stage in the execution of a branch-and-bound procedure, let us say that a restriction is *active* if it has been created but not yet branched from, so that its children have not yet been created. A typical branch-and-bound procedure maintains a list of active restrictions. For each active restriction it keeps track of the associated lower bound $A(s)$, together with an indication of whether s is terminal or nonterminal. Initially the only active restriction is $*^m$, the restriction in which all coordinates are free. At each step the procedure selects some nonterminal active restriction s and replaces it by its two children, $L(s)$ and $R(s)$. This step is called *expanding* the restriction s. The computation terminates when the smallest lower bound, among all the active restrictions, corresponds to some terminal restriction s. In this case it is easy to check that

$$A(s) = \min_x g(x) = \min_{x \in Dom(s)} g(x),$$

so that the combinatorial optimization problem is solved.

The following rule defines the branch-and-bound procedure known as best-first-search: among all the active restrictions, expand the one whose lower bound is smallest. In the case where the function $A(s)$ is one-to-one, so that no two restrictions have the same lower bound, this procedure is optimal; every restriction that it expands must be expanded by every branch-and-bound procedure. Among branch-and-bound procedures, best-first-search is the method of choice when enough space is available to store the list of active restrictions. However, this amount of storage is typically exponential in m, the size of the problem. In such cases space may be too limited for best-first-search to be implemented.

The algorithms given in the previous section allow branch-and-bound procedures to be executed using a very small amount of space, at the cost of a modest increase in execution time. To make explicit the connection between our results and the branch-and-bound method, let us think of the branching rule as determining a finite, rooted, ordered binary tree whose nodes correspond to restrictions. The root of the tree corresponds to the restriction $*^m$; if node s is not terminal then its children are the restrictions $L(s)$ and $R(s)$. The leaves of the tree correspond to terminal restrictions. The goal of the branch-and-bound procedure is to find a terminal restriction of least value. Because of the monotonicity property of the lower bound function $A(s)$, the values on any path directed away from the root form a nondecreasing sequence. Thus the tree conforms to the assumptions of our tree search model, with two exceptions: the tree is finite, and the same value may occur on more than one node. These deviations from the model are easily dealt with. The finiteness of the tree can only make the tree search problem easier. In the presence of equal values a simple tiebreaking rule can be used to obtain a total ordering of the nodes of the tree. Hence, the branch-and-bound problem can be solved by repeated use of any of our tree search algorithms, doubling the input parameter at each step, and halting when the terminal node of least value is found.

The following table compares the time and space requirements of best-first-search with those of the algorithms in the previous section when the terminal node of least value is the n^{th}-smallest node in the branch-and-bound tree.

	Time	Storage
best-first-search	$O(n)$	$O(n)$
Algorithm B	$n\ 2^{O(\sqrt{\log n})}$	$O(\sqrt{\log n})$
Algorithm C	$n\ 2^{O(\sqrt{\log n})}$	$O(\log^{2.5} n)$
Algorithm D	$n^{1+\epsilon}$	$O(1)$

REFERENCES

[AHU] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1975.

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

[MP] J. I. Munro and M. S. Paterson, "Selection and Sorting with Limited Storage", *Proceedings of the 19th FOCS Conference*, pp. 253-258, 1978.

[PS] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, 1982.

APPENDIX

All our algorithms will be described in informal PASCAL. Every procedure will have as a first input parameter the vertex in which it resides. A procedure residing at v can (recursively) call a subroutine that will reside in either v,$L(v)$ or $R(v)$. When such a subroutine terminates, it returns to the vertex from which it was called. Hence, the time analysis, we shall count the number of subroutine calls to a child.

We shall use upper case variables for vertex values, and lower case variables for vertex names and integer counters. The names of procedures appear in boldface. Let T_v will denote the tree rooted at vertex v, and $T^{(k)}$ be the k^{th} smallest value in the tree T.

```
Algorithm A (r,n)        (returns a value)
   return (findbest (r,n))

findbest (v,n)           (returns a value)
   LOWER ← val(v), UPPER ← UP
   while good (v,n, LOWER) < n do;
      lr ← roots (v,1, LOWER, UPPER)
      k ← 1; NEW_LOWER ← LOWER;
      while lr ≠ 0 and [good (v,n, NEW_LOWER)
          - good (v,n, LOWER) < g(n)] do;
          S ← expand (v,k, LOWER, UPPER);
          (L,U) ← bounds (v,n,S);
          if L ≠ − ∞ then NEW_LOWER ← L;
          if U ≠ ∞ then UPPER ← U;
          k ← 2k; lr ← roots (v,k, LOWER, UPPER);
          if lr = 0 then NEW_LOWER ← UPPER.
      end
      LOWER ← NEW_LOWER;
   end
   return select (v,n, LOWER)

expand (v,k, LOW,UP)     (returns a set of values)
   if val(v) ≤ LOW then return
      expand (L(v),k, LOW, UP). ∪ expand (R(v),k, LOW, UP)
   else if good (v,k,UP) ≥ k then
      return (findbest (v,k))

good (v,n,K)             (returns an integer)
   if val(v) ≤ K then do.
      y ← 1 + good (L(v), n − 1, K)
      if y < n then return (y + good (R(v), n − y K).
      else return (n)
   end
   else return (0).

roots (v,k, LOW,UP)      (returns an integer)
   if val(v) ≤ LOW then return
      roots (L(v),k, LOW, UP) + roots (R(v),k, LOW, UP)
   else if good (v,k, UP) < k then return (0)
                  else return (1).

bounds (v,n,S)           (returns two values)
      Sort S.
      using binary search find:
   L ← largest value L' in S∪{−∞} s.t. good (v,n,L') ≤ n
   U ← smallest value U' in S∪{∞} s.t. good (u,n,U') ≥ n
   return (L,U)

select (v,n,K)   (n ≤ good (v,∞,K) < 5n)      (returns a value)
   simple binary search on the
   values ≤ K in T_v to find X s.t. good (v,∞,X) = n
   return (X).
```