PARALLEL BRANCH-AND-BOUND FORMULATIONS FOR AND/OR TREE SEARCH

V. Kumar and L. Kanal

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-83-14 August 1983

TR 83-14 August 1983

Parallel Branch-and-Bound Formulations for And/Or Tree Search

by

V. Kumar* and L. Kanal**

Abstract

This paper discusses two general schemes for performing branch-and-bound (B&B) search in parallel. These schemes are applicable in principle to most of the problems which can be solved by B&B. The schemes are implemented for SSS*, a versatile algorithm having applications in game tree search, structural pattern analysis and And/Or graph search. The performance of parallel SSS* is studied in the context of And/Or tree and game tree search. The paper concludes with comments on potential applications of these parallel implementations of SSS* in structural pattern analysis and game playing.

* Dept. of Computer Sciences, University of Texas at Austin, Austin, TX78712

** Dept. of Computer Sciences, University of Maryland, College Park, MD20742

I. Introduction

And/Or graphs provide graphical representations for problem reduction formulations. Due to the natural correspondence between context-free grammars and And/Or graphs [8], they have also been found quite useful as structural (descriptive) models for some types of patterns [12]. In the context of waveform parsing applications, Stockman [30], [32] developed a procedure called SSS* (state space search) which searches an And/Or graph for the description of a pattern. By developing alternate descriptions of a pattern in a "best-first manner", SSS* efficiently produces a largest merit description, when the merit is defined in a specific manner. Stockman noticed that, due to a structural correspondence between And/Or graphs and game trees, SSS* can also be used for performing minimax search of game trees. What came as a greater surprise was that SSS* outperforms the classical alpha-beta algorithm [13] for game tree search in terms of the number of nodes evaluated [31]. The relative performance of SSS* with respect to alpha-beta, according to various performance measures, is a subject of continuing investigations [31], [25], [28], [5].

Various heuristic search procedures for state space search (e.g., A* [23]), And/Or graph search (e.g., AO* [24]), and game tree search have been developed in Artificial Intelligence, and have been thought to be related to Branch and Bound(B&B) procedures [10], [18] of Operations Research; but the relationship between these two classes of procedures has been rather controversial (e.g., see Pohl [26]). In [15], [16], [22] we have presented a general formulation for B&B and have shown that a number of AI search procedures such as A*, AO*, SSS*, B* [4] and alpha-beta are special cases of our B&B formulation. By considering these procedures under B&B framework it is easy to understand their similarities and differences. The B&B formulation presents a unified approach to simulating and analyzing a number of search procedures.

Besides providing a better understanding of the nature of and the relationships among various search procedures, the B&B formulation also makes it easy to visualize parallel implementations of many search procedures. This paper discusses two general schemes for performing B&B search in parallel. These schemes are applicable in principle to most of the problems which can be solved by B&B. But the effectiveness of a parallel implementation is expected to be dependent

upon the nature of the specific problem being modeled. The schemes are implemented for SSS*, and the performance of the parallel implementations is studied in the context of And/Or tree and game tree search. A number of parallel implementations of alpha-beta and B&B have been proposed and implemented; the effectiveness of the parallel implementations of SSS* can be evaluated in the context of the performances of these implementations.

Section II surveys the work on parallel implementations of alpha-beta and B&B procedures. Section III briefly reviews And/Or trees and considers how they can be viewed in two different ways as models of two person, perfect information board games. Section IV presents a brief review of branch and bound. Section V shows how SSS* can be considered as a B&B procedure from two different view points. Section VI presents two general schemes for performing B&B procedures in parallel and implements them for SSS*. The simulation results of parallel SSS* applied to game trees of various heights and depth are also presented. Section VII comments on the potential applications of these parallel implementations of SSS* in structural pattern analysis and game playing, and concludes with suggestions for further work.

II. A Brief Survey of Previous Work

Various researchers [6], [11], [19] have proposed and investigated parallel implementations for Branch-and-Bound. Imai et. al. [11] proposed a parallel depth first B&B algorithm in which n processes work concurrently on n "deepest" (or n most recently generated) subproblems. The set of generated subproblems is kept in a common store accessible to all the processors so that the B&B processes running on them can choose the currently deepest subproblem in each cycle of branching and bounding; hence this implementation is suitable only for multiprocessor architectures. In a parallel implementation proposed by Ma and Wah [19], n concurrent B&B processes work on n currently "best" subproblems. In their implementation, processors do not share common memory; but the subproblems need to be compared and exchanged between different processes to ensure that the concurrent processes work on n currently best subproblems. They proved that after n cycles of compare and exchange, the processes will have the n currently best subproblems. Due to the considerable communication required between the parallel processes, this approach is also unsuitable for implementation on "general purpose" distributed architectures like cm* [33] or zmob [27].

Various parallel implementations of the well known alpha-beta minimax search procedure have been proposed and investigated in the literature [1], [3], [7], [5]. Baudet [3] proposed a parallel version of alpha-beta, in which multiple processes search the same game tree concurrently, but with different alpha-beta windows. The speed-up provided by this technique is limited by a constant even if a large number of processors are available. The reason is that even if a process is started with the correct alpha and beta bounds, the amount of search needed to establish that the bounds are correct still is, on the average, a major fraction of the search required by a process started with "uninformed" starting alpha and beta [-oo,+oo] values***. Hence this kind of implementation is suited only for multiprocessors with a few processors. In the parallel Alpha-Beta procedure of Akl et. al. [1], processes search disjoint parts of the game tree; but to avoid unnecessary node evaluations, certain processes are assigned higher priority over other processes. Their

*** This reasoning is supported by Baudet's analytical results [3] and by our experiments.

data showed that the speed-up is bounded from above by a constant (approximately 5). Finkel and Fishburn [7] implemented a somewhat similar scheme, in which "higher" processes were designated as Master, which scheduled "lower" slave processes to search disjoint parts of the look-ahead tree. With n processors, a speed-up of at least order of n**1/2 was predicted. Their parallel implementation becomes very effective if the game tree to be searched is strongly ordered, i.e., the best move from a position is in the beginning part of the heuristically ordered (in terms of merit) list of moves possible from that position. Marsland and Campbell [20] present a comprehensive survey of parallel algorithms for searching strongly ordered game trees.

III. And/Or trees and Game Trees

A detailed treatment of And/Or graphs can be found in [23], [24]. To keep the discussion simple, in this paper we limit our presentation to AND/OR trees. Most of the concepts and techniques presented are also applicable to And/Or graphs. In this section we briefly review And/Or trees and their correspondence with game trees.

A. And/Or trees

Each node of an And/Or tree represents a problem, and a special node called 'root' represents the original problem to be solved. Nodes having successors are called *nonterminal.* Each nonterminal node has all immediate successors either of type AND or of type OR. A problem whose (nonterminal) node has immediate successors of type OR is solved if any of the successors is solved; while a problem whose node has immediate successors of type AND is solved if all of the successors are solved. Nodes with no successors are called *terminal,* and each terminal node represents a primitive problem which is known to be either solved or unsolved. Given an And/Or tree representation of a problem, we can identify its potential solutions, each represented by a "solution tree".

A *solution tree* T of an And/Or tree G is a subtree of G with the following properties:

(i) The root node p of the And/Or tree G is the root node of the solution tree T.

(ii) If a nonterminal node of the And/Or tree G is in T then all of its immediate successors are in T if they are of type AND, and exactly one of its immediate successors is in T if they are of type OR.

****** Fig. 1 ******

Fig. 1 shows an And/Or tree and one of its solution trees. In many problems, a merit (or a cost) function is associated with solution trees, and a largest merit (or a least cost) solution tree is desired. There are various ways in which a merit function can be defined, but the one defined below is of special interest to us. Let c(n) denote the merit (or cost) of a terminal node n of G.

Merit function f: Let T be a solution tree of an And/Or tree G, then f(T) = min{ c(t) | t is a terminal node of T}.

B. Correspondence with Game trees

And/Or trees can also be used as models of two person, perfect information board games [23], [29]; e.g., the And/Or tree of Fig. 1 can be viewed as a game tree. The game is played between players MAX and MIN; in the corresponding And/Or tree, board positions resulting from MAX's moves are represented by OR nodes (circle nodes in Fig. 1), and the positions resulting from MIN's moves are represented by AND nodes (square nodes in Fig.1). Moves of the game proceed in strict alternation between MAX and MIN, until no further moves are allowed by the rules of the game. After the last move, MAX receives a payoff which is a function of the final board position (c-value of the corresponding terminal node). MIN has to forfeit the same amount. Thus, MAX always seeks to maximize the payoff while MIN does the converse. Assuming that the root node of the tree corresponds to the current position of the game from which MAX is to move, the objective is to find a move for MAX which guarantees the best payoff. The best payoff that MAX can be guaranteed for a game is given by the minimax value of the corresponding And/Or tree. This evaluation can be defined in a recursive manner. We first define a minimax function g for all nodes of an And/Or tree G as follows:

(i) If n has immediate successors of type OR:
$g(n) = \max\{g(n_i)\}$ for all immediate successors n_i of n.

(ii) If n has immediate successors of type AND:
$g(n) = \min\{g(n_i)\}$ for all immediate successors n_i of n.

(iii) If n is a terminal node of G:
$g(n) = c(n)$.

Then the minimax value of an And/Or tree G is defined a g(p), where p is the root node of G.

In [31], Stockman made a remarkable observation (with a formal proof) that the minimax value of a game tree is the same as the maximum f-value (as defined in section II.A) of all solu-

tion trees when a game tree is viewed as an And/Or tree. An intuitive explanation for this result is that a solution tree represents all possible responses by MIN to some particular sequence of moves made by MAX, i.e., it represents a particular strategy for MAX. MIN can always choose the sequence of moves leading to the minimum valued tip node and thus minimize the payoff for a strategy chosen by MAX. Thus the merit (or the guaranteed payoff) of a solution tree is taken as the minimum c-value of all its tip nodes. Each solution tree represents an alternative strategy for MAX. Since MAX is free to choose any possible strategy he would choose the one corresponding to the solution tree with the highest merit, to guarantee the maximum payoff by MIN.

The above intuitive explanation suggests that the AND/OR tree may also be viewed from MIN's point of view which is complementary to that of MAX described above. In this complementary formulation, a solution tree T of an And/Or tree G is defined as follows:

(i) The root node of the AND/OR tree G is the root node of the solution tree T.

(ii) If a nonterminal node of G is in T then all of its immediate successors are in T if they are of type OR and exactly one of its immediate successors is in T if they are of type AND.

We refer to such a solution tree as an "OR solution tree" in contrast to the previous solution tree which will, henceforth be termed an "AND solution tree." However, for the sake of convention and simplicity, we shall also continue to use the term "solution tree" to mean "And-solution trees".

An OR solution tree represents all possible sequences of responses by MAX to some possible sequences of moves made by MIN. Given a particular sequence of moves made by MIN, MAX is at liberty to choose the responses which lead to the highest valued tip node hence giving maximum payoff for that particular sequence by MIN. Therefore we define the value of an OR solution tree to be the maximum f-value (or cost) of all terminal nodes in T.

Each OR solution tree represents one possible sequence of moves by MIN, and MIN would choose the sequence corresponding to the OR solution tree with lowest value to guarantee minimum payoff. Hence the minimax evaluation of a game tree can also be defined as the minimum cost of all OR solution trees when a game tree is viewed as an AND/OR tree.

IV. Branch and Bound : A Brief Review

The class of problems solved by Branch and Bound procedures can be abstractly stated as follows:

> For a given arbitrary discrete set X ordered by a real valued merit function f: X-->R, find some x* G X such that: for all x G X, f(x*) $\geq$ f(x).***

Branch and Bound procedures decompose the original set into sets of decreasing size. The decomposition of each generated set (branching operation) is continued until tests reveal that it is either a singleton (then we measure its merit directly and compare with the currently best member's merit) or proved not to contain an optimum element (an element with greatest merit) of the set X, in which case, the set is "pruned" or eliminated from further consideration.***** If the decomposition process is continued (and satisfies some properties), we eventually find an optimum element. Often, only a small fraction of the total set need be generated.

In the earlier formulations of B&B [21], [2], only the lower and upper bounds on the merit values of the elements of (sub)sets (of X) were used for pruning. If two sets X_1 and X_2 are in the collection of sets under consideration, and the lower bound on the merits of elements in X_1 is no smaller than the upper bound on the merits of elements in X_2, then X_2 can be pruned. The use of bounds for pruning gave the procedure its name branch-and-bound.

The concept of pruning by bounds was later generalized to include pruning by "dominance" (see [14], [9], [10], [15], [22]). A dominance relation D is defined between subsets X_1, X_2 of X such that X_1 D X_2 if and only if an optimum element of X_1 is no worse than an optimum element of X_2. If two sets X_1, X_2 are in the collection under consideration and X_1 D X_2, then X_2 can be pruned.

There are various ways of selecting a subset of X (from the collection of subsets of X under consideration) for branching, leading to different performances in terms of time and space

*** Discussion in this section is also applicable (with appropriate modifications) to the case, when f denotes the cost of the elements of X, and a least cost element is desired.

***** More precisely, we need only to prove that even after eliminating the set in question, at least one of the remaining sets still under consideration contains an optimum element of X.

requirements. The following two selection strategies for branching are rather commonly used in B&B procedures.

Best-first Selection Strategy

In many problem domains it is possible to associate an upper bound*** with the subsets of X. This upper bound information can be fruitfully used in the selection of an element for branching. If the branching is always performed on that subset of X which has the largest upper bound of all other available subsets of X then the selection rule is called *best-first,* and the branch and bound procedure using such a strategy is called best-first branch and bound.

If these bounds are tight (i.e., if the upper bound of a subset of X is close to the merit of an optimum element of the subset, for all subsets of X) then best-first B&B becomes very efficient [18].

Depth-first Selection Strategy

Another way of selecting a set for branching from the active collection A, is to select a set from those sets which have been generated most recently as a result of branching [18]. This selection rule is called *depth-first.* The major advantage of the depth-first selection rule over the best-first selection rule is that, in general, it requires less storage. But the depth-first search, being an exhaustive search, can be much slower than the best-first search.

***or lower bound instead, if f(x) represents the cost of x, and we are interested in a least cost element of X

Section V. SSS as a B&B Procedure*

In this section we discuss how SSS* can be considered as a B&B procedure in two different ways. We assume familiarity with the SSS* algorithm as presented in [31].

V.A. SSS as a Best-first Search Procedure in the Space of AND Solution Trees*

SSS* can be interpreted as a B&B procedure searching for a largest merit solution tree of an And/Or tree G. The discrete set X is the set of all solution trees of G. The merit of function f is as defined in Section III.A. SSS* as presented in [31] maintains a list of states (n,s,h) called OPEN; each of these states can be considered as representation of a set of solution trees of G. The value h is essentially an upper bound on the merit value of the solution trees represented by the triple (n,s,h). Expanding a state corresponds to the operation of branching. Purging states from OPEN corresponds to pruning dominated subsets. The precise details of this interpretation can be found in [16]. Fig. 2 illustrates this informally through an example.

******* Fig. 2 **********

Fig. 2.a shows just the root node of the And/Or tree G of Fig. 1. It represents the set of all solution trees of G. Fig. 2.b shows the two disjoint set of solution trees resulting from the branching operation on the set of Fig. 2.a, or equivalently from the expansion of node 1 in SSS*. The branching operation splits the total set of solution trees into two disjoint subsets now denoted by T_1' and T_2'. Fig. 2.c shows the sets of solution trees T_{11}' and T_{12}' resulting from the branching operation on T_1', or equivalently from the expansion of node 2 (and later node 4) in SSS*. Fig. 2.d shows the sets of solution trees T_{11}' and T_{12}' after evaluation of node 16 of T_{11}' and nodes 18, 19 of T_{12}'. From lemma 3.5 given in [16], T_{12}' dominates T_{11}', hence T_{11}' is eliminated; equivalently, in SSS* node 17 is eliminated.

V.B. SSS as Depth First Search in the Space of OR solution Trees.*

SSS* can also be viewed as a depth first Branch & Bound search for a minimum cost solution in the space of OR solution trees (also called OR trees for brevity). Viewed this way, SSS* keeps partitioning the most recently generated subset until it contains only one OR tree. The value of this OR tree, which is defined as the maximum value of all tip nodes then works as an upperbound. Other alternate solution trees are then partially (or fully) generated. Any time a new partial OR tree is found to have a lower bound greater than the cost of the current OR solution tree, the complete set represented by the partial OR tree is eliminated. If a better OR tree is found, it replaces the previous best-known solution. This process is repeated until the complete space of OR trees represented the And/Or tree has been explored.

A formal development of a depth first B&B procedure equivalent to SSS* can be done along the lines developed in [16]. For the sake of brevity, we shall demonstrate this informally by an example. In the following, we describe some steps in the search of And/Or tree G of Fig. 1 by the SSS* algorithm, and then reinterpret the operation of SSS* as that of a depth first B&B procedure searching in the space of OR solution trees.

Fig. 3.a shows the And/Or tree G after its four tip nodes have been explored by SSS*. Each arrow corresponds to a state (3-tuple representation of a partial tree) on the ordered list OPEN maintained by SSS*. We can also regard the four arrows as pointing to the tip nodes of the complete OR-solution tree (of value 78) represented in boldface. The OR-tree space is thus divided into two parts: the completely explored OR-tree, and the rest of the OR-trees. By exploring node 19 we will be pulling in another OR-tree (nodes 16, 19, 24, 26) from the second subset.

In Fig. 3.b, after evaluating node 19, we have an OR-tree (nodes 16, 19, 24, 26) of value 52, which is better (lower cost) than the previous OR-tree, which is now rejected. Also rejected are all OR-trees containing node 18, as they can never have value less than 78.

In Fig. 3.c, node 19 has the highest value and all AND-successors of its parent node 9 are solved. Therefore, in SSS*, the solved node 19 is replaced on OPEN by solved node 9 with the

same value. In terms of OR-trees, the OR-tree (nodes 16,9,24,26) now represents the previous OR-tree (nodes 16,19,24,26).

In Fig. 3.d, the solved node 9 is an OR-successor of its parent and has the highest value on OPEN. It is replaced by its parent, node 4. All successors of node 4 (node 16 in this case) are purged from OPEN. In terms of OR-trees, any OR-solution tree having either node 16 or 17 will also have (from the definition of OR-solution trees) node 9 in it, hence its value is bounded from below by 52. These solution trees are dominated by the already found OR-solution tree (nodes 16, 19, 24, 26) which is represented along with the value by (nodes 4, 24, 26).

VI. Parallel Implementations

This section presents two approaches to implement SSS* in parallel, which in fact are parallel implementations of the general B&B procedure. Hence both can be universally applied in any context in which B&B is applicable. In the first approach multiple processes perform depth first search concurrently. At any time at least one process has the property that if it terminates it returns an optimum solution; the other processes conduct a look-ahead search. In the second approach, the total search space is divided into several disjoint parts and each part is searched concurrently in depth first manner by a different process. Both approaches require very little inter-process communication and, therefore, are ideally suited for implementation on loosely coupled distributed systems (e.g. zmob [27], cm* [33]).

VI.A. Parallel Implementation of SSS - I*

Let us consider the case of a depth first B&B procedure searching for a least cost solution***. While searching depth first, a large portion of the search space can be pruned if a solution close to the optimum is found early in the search process. To speed up the search process, we can assume the existence of some hypothetical solution with a value which we hope is larger than but close to the optimum solution. We then keep discarding any partial solution (or set of potential solutions) whose lower bound is greater than the value of the hypothetical solution. If the optimum value is really smaller than the value of the hypothetical solution, the best solution will be found during the search by exploring a much smaller part of the total solution space. However, if the assumed solution has smaller cost than all other existing solutions, they will eventually be eliminated in favor of the assumed one, and the search will terminate without finding the best solution.

In Section V we argued that SSS* can be viewed as a depth first search for a least cost OR solution tree. In SSS* [31] the search is started by associating +oo as a bound with the starting state. In terms of depth first B&B search in OR tree space, this is equivalent to starting with a hypothetical solution of value +oo. If the starting bound (associated with the initial state) is chosen to be smaller than +oo but larger than the cost of an optimum OR solution tree then the procedure finds an optimum OR solution tree by searching a smaller part of the total space. Fig. 4 shows the number of nodes expanded vs. starting bound in the search of a random uniform game tree**** (RUG tree) of degree 2 and depth 10. The terminal values of the RUG tree are independently drawn from a uniform distribution between 0 and 100.

If we have multiple processors we start several (otherwise sequential) Branch & Bound processes, one on each processor, with different starting bounds (the assumed solutions). To avoid disappointment we start one B&B process with the most pessimistic bound (e.g. +oo). To speed

*** Discussion in this chapter is applicable to depth-first B&B search for largest merit solution with obvious modifications.

**** A game tree is called a random uniform game tree (or *RUG* tree) if all of its terminal nodes are at the same depth and if its terminal values are independently drawn from a common distribution.

up the search, the other B&B processes are started closer to the expected value of the optimum solution. Fig. 5 shows depth first B&B processes searching for an optimum solution with different starting bounds when the costs of all solutions lie between 0 and m.

In the beginning only one process (with the pessimistic bound) is guaranteed to provide the optimum solution; the others can be considered as look-ahead processes. If at any time, any of the look ahead processes finds a better solution than the hypothetical one it started with, we know that the process, if allowed to continue, would terminate with the optimum solution. Now any other process whose current best known (actual or hypothetical) solution is worse than the recently discovered solution can be given a more optimistic (hypothetical) solution for further search. Thus at any time, at least one process is guaranteed to terminate with the optimum solution and the others conduct a look ahead search. This goes on until the optimum solution is found by one of the processes, assuming the solution exists.

This kind of search can be particularly efficient if some probabilistic information about the optimum value (e.g. the probability distribution) is available. This type of implementation requires very little inter-process communication as processes need to communicate only when a process finds a better solution than the one previously known. Furthermore, processes never wait to receive some input from other processes but continue or terminate according to whatever information is currently available.

Simulation Results

We implemented a version of parallel SSS*, along the lines proposed above. Concurrent search processes were obtained by simple modifications in sequential SSS* by including steps for updating local and global bounds (a bound for a process denotes the merit of the best actual or hypothetical OR-solution tree found so far by the process). Simulation runs for a two processor system were conducted for uniform trees of different depths and degree of branching. Tip values were assigned from a uniform distribution. Using a simple, adhoc procedure for selecting assumed bounds (i.e., knowledge of the probability distribution of the minimax value of the AND/OR tree

was not used in selecting the bounds), a speedup of up to 30% over the sequential case was estimated for the two processor system.

The speedup achieved by this approach is very much dependent on how much pruning can be done by "hypothesizing" a good solution (i.e., how steep is the curve in Fig. 4 around the optimum value). Also, to be able to make a proper selection of look-ahead bounds, one needs some probabilistic information (e.g., expected value) about the optimum value. This kind of approach did not turn out to be very efficient for implementing SSS* in parallel, but it could be quite effective in B&B implementations in many other applications. This approach is somewhat similar to the one proposed by Baudet [3] which provided an efficient parallel implementation of the alpha-beta procedure. This type of parallel search (like Baudet's parallel implementation of alpha-beta) is suited only for multiprocessors with a few processors because the speedup achieved is usually limited even if many processors are used. The reason is that the amount of search done by a B&B process started with a correct bound could be a major fraction (as it is in the case of minimax search) of the search required by a process started with an uninformed (i.e., a pessimistic) bound.

VI.B. Parallel Implementation of SSS - II*

An alternate approach to conducting Branch & Bound search in parallel is to divide the total search space into several disjoint parts and let each part be searched in depth first fashion, concurrently, by a different process. Whenever, during the search of its local space, a process comes up with a better solution, it communicates it to all the other processes in the system. Whenever any process encounters a partial solution (or a group of solutions) which is guaranteed to be worse than the globally known solution, it discards that partial solution. Thus the independent processes cooperate (by keeping a current global best solution) and perform the search in parallel. Processes work asynchronously, as at no time does a process have to wait to receive any input from another process. Therefore, this approach should be ideal for implementation on a loosely coupled distributed network (see Kung [17]). A technique for deriving parallel B&B similar in spirit to the one we describe here was proposed in [6]. The speedup provided by any such parallel implementation would depend upon how neatly a search space can be divided into similar disjoint subspaces, and the extent to which search in one space affects the search in the other space.

In contrast, note that if individual (concurrent) processes perform a best first search on their local space, they will not be able to benefit from each other's search by sharing information on bounds. The reason is that in best first search, a process does not obtain a complete solution until it actually terminates. Information about the best recent bound (e.g. lower bound when searching for a least cost solution) of one process cannot help in pruning the local search space of another process.

An AND/OR tree with the root node having OR successors, can be easily divided into disjoint subspaces (of AND solution trees) each a subtree rooted at an OR-successor of the root node of G. On the other hand a division of the total space of solution trees in terms of OR solution trees is going to be rather complicated since all the OR solution trees will have all the top OR branches in common.

Having divided the AND solution tree space into disjoint parts, we need to find an efficient method for doing depth first search in the AND solution tree space. In its present form, SSS* does best first search in the AND solution tree space and depth first search in the OR solution tree space. What we wish is the converse, i.e. depth first search in the AND solution tree space, or equivalently best first search in the OR solution tree space.

A Branch & Bound strategy for the search of the best (i.e., least valued) OR-solution tree of an And/Or tree can be formulated (in a way similar to the search for the best And-solution tree of an And/Or tree) by expanding and evaluating partial OR solution trees in the best first (i.e., least lower bound first) order. We call the resulting procedure dual-SS*. Using an argument similar to the one used in Section V it can be shown that dual-SS* is also a depth first search in the AND solution tree space.

Viewed as a depth first search in And-solution tree space, dual-SS* maintains a complete AND-solution tree found to be best so far, eliminates any partially explored AND-solution tree of lower merit than the current solution, and replaces the current solution only when a solution tree with better merit is found. The merit of the current best AND-solution tree characterizes the search at any stage of the process. It follows that we can use concurrent dual-SS* processes to perform depth first search in the disjoint subspaces of the original AND/OR solution tree.

Speed up factor

Let R(n) be the branching factor (as defined in [23]) of SSS* for a RUG tree of degree n and depth d (denoted as G(n,d)). The average number of terminal nodes evaluated by SSS* (or by dual-SS*) *** in searching G(n,d) is $[R(n)]^d$. If the tree is statically divided at the top level then each resulting subtree is G(n,d-1). The average number of terminal nodes evaluated by SSS* (or by dual-SS*) in searching G(n,d-1) is $[R(n)]^{d-1}$. Let us assume that the time taken by a search process is proportional to the number of terminal nodes evaluated. It follows that n processes,

***Even though, the SSS* and dual-SS* procedures search the nodes of a game tree in very different orders, there is no reason to believe that one procedure would have a lower branching factor than the other. This intuitive observation is supported by our simulations.

running on n independent processors, searching G(n,d-1) trees, will finish search faster, and the speedup (over the case when the whole G(n,d) tree is searched by a single process) should be $[R(n)]^d/[R(n)]^{d-1} = R(n)$. This assumes that all independent processes finish search of their G(n,d-1) trees at the same time. But, the number of terminal nodes evaluated and, therefore, the search time taken by these processes is a random variable as it depends upon the terminal values of the game tree being searched. Hence, the actual speedup is also a random variable. Let $N_{G(n,d)}$ be the number of terminal nodes evaluated by a SSS* process in the search of a G(n,d) tree, and let $N^i_{G(n,d-1)}$ be the number of terminal nodes evaluated by the dual-SS* process searching the game tree G(n,d-1) rooted at the ith successor of the root of G(n,d). Then, the speed-up = $N_{G(n,d)}/\max\{N^i_{G(n,d-1)} \mid 1 \leq i \leq n\}$. Thus the speedup should be less than R(n). But in our implementation, these parallel processes searching G(n,d-1) are allowed to communicate which should increase the value of speedup.

Simulation Results

We implemented dual-SS* by making modifications in the existing implementation of SSS*, seeking a maximum wherever a minimum was taken, replacing OR's by AND's and vice versa. We ran simulations for the parallel implementation on RUG trees G(n,d) for various branching factors n and depths d. For each of these (n,d) pairs, 50 RUG trees were generated, and their tip values were independently drawn from a uniform distribution between 0 and 10000. For a RUG tree G(n,d), n dual-SS* processes were used to search the game trees rooted at the n immediate successors of the root of G(n,d). In the absence of parallel hardware, a controller process was used to run the "parallel" dual-SS* processes in an interleaved fashion and to provide the communication of global bounds between them. For each G(n,d) tree the largest of the number of terminal nodes evaluated by concurrent dual-SS* processes searching the subtrees of G(n,d), was measured. The largest number is denoted by $M_{G(n,d)}$. The speed-up, in searching G(n,d), of the parallel implementation over the sequential procedure was calculated as $N_{G(n,d)}/M_{G(n,d)}$.

For various values of (n,d), Table 1 shows average, maximum and minimum speed-ups of the parallel implementation (on n processes) over the sequential case, in searching 50 G(n,d) game trees. It also shows the average values and the standard deviations in $M_{G(n,d)}$ and $N_{G(n,d)}$. Note that the standard deviation of $M_{G(n,d)}$ is consistently much smaller than the standard deviation of $N_{G(n,d)}$. This suggests that the maximum of the number of terminal nodes evaluated by the parallel processes is not very much larger in a case where sequential SSS* evaluates lots of nodes. To verify this, for each (n,d) pair, out of the 50 G(n,d) trees we selected the 10 G(n,d) trees in which sequential SSS* evaluates the largest number of terminal nodes. For these 10 G(n,d) trees we computed the average speed-up, denoted by S_{10}. Interestingly, this speed-up figure was always found to be considerably better than the average speed-up over the 50 cases (see the last column of Table 1). Thus the parallel implementation is most effective in the situations where it is needed most.

VII. Conclusion

The formulation of Section VI.B allows parallel non-directional search of a description of a pattern in alternate structural models (context free grammars). Each candidate model for an observed pattern can be considered as an AND/OR subtree of a larger AND/OR tree with the root of each subtree being one of the OR successors of the root of the larger AND/OR tree. The procedure can be implemented via a set of processes working independently and asynchronously on the different models with the merit of the recent best parse tree being communicated among the processes. As noted earlier such a mode of loosely coupled parallel computation is natural for a distributed computer environment. In the context of game playing, each process analyzes the goodness of an alternative move for MAX. The processes together maintain a globally best strategy found so far for MAX, and discard any partial strategy (or a set of strategies) which are guaranteed to be worse than the global solution. This kind of parallel search on independent different subtrees of a game tree does not suffer from the lack of information flow between various processes (mentioned by Baudet [3] in reference to alpha-beta) because, at any point during the search, the decision of pruning by any process is based upon all the information previously acquired during the search by all the processes.

Due to the complex nature of the interaction between communicating processes, it is difficult, even under simplifying assumptions, to come up with a model to calculate the actual speedup due to parallelism. It would be interesting to run extensive simulations of the parallel implementations discussed in Section VI and the others proposed in the literature for the alpha-beta and B&B procedures and compare the results.

REFERENCES

[1] S. G. Akl, D. T. Bernard, and R. J. Doran, Design and Implementation of a Parallel Tree Search Algorithm, *IEEE Transactions on PAMI 4,* 2, pp. 192-203, 1982.

[2] E. Balas, A Note on the Branch-and-Bound Principle, *Operations Research 16,* pp. 442-444, 886, 1968.

[3] G. Baudet, The Design and Analysis of Algorithms for Asynchronous Multiprocessors, Ph.D. Dissertation, Carnegie-Mellon Univ. , 1978.

[4] H. Berliner, The B* Tree Search Algorithm: A Best-First Proof Procedure, *Artificial Intelligence 12,* pp. 23-40, 1979.

[5] M. Campbell, Algorithms for the Parallel Search of Game Trees, Tech. Rep. 81-8 Dept. of Computer Science, University of Alberta, 1981.

[6] O. I. El-Dessouki and W. H. Huen, Distributed Enumeration on Network Computers, *IEEE Transactions on Computers C-29,* pp. 818-825, Sept. 1980.

[7] R. A. Finkel and J. P. Fishburn, Parallelism in Alpha-beta search, *Artificial Intelligence 19,* pp. 89-106, 1982.

[8] P. A. V. Hall, Equivalence between AND/OR Graphs and Context-Free Grammars, *Comm. ACM 16,* pp. 444-445, 1973.

[9] T. Ibaraki, The Power of Dominance Relations in Branch and Bound Algorithms, *J. ACM 24,* pp. 264-279, 1977.

[10] T. Ibaraki, Branch-and-Bound Procedure and State-Space Representation of Combinatorial Optimization Problems, *Inform and Control 36,* pp. 1-27, 1978.

[11] M. Imai, Y. Yoshida, and T. Fukumura, A Parallel Searching Scheme for Multiprocessor Systems and its Application to Combinatorial Problems, *Proc. Sixth Internat. Joint Conf. on Artif. Intell.,* pp. 416-418, 1979.

[12] L. N. Kanal, Problem-Solving Models and Search Strategies for Pattern Recognition, *IEEE Trans. Pattern Analysis and Machine Intell. 1,* pp. 193-201, 1979.

[13] D. E. Knuth and R. W. Moore, An Analysis of Alpha-Beta Pruning, *Artificial Intelligence 6,* pp. 293-326, 1975.

[14] W. H. Kohler and K. Steiglitz, Characterization and Theoretical Comparision of Branch and Bound Algorithms for permutation problems, *J-ACM 21,* pp. 140-156, 1974.

[15] V. Kumar, A Unified Approach to Problem Solving Search Procedures, Ph.D. thesis, Dept. of Computer Science, University of Maryland, College Park, December, 1982.

[16] V. Kumar and L. Kanal, A General Branch and Bound Formulation for Understanding and Synthesizing And/Or Tree Search Procedures, *Artificial Intelligence 21,* 1, pp. 179-198, 1983.

[17] H. T. Kung, Synchronized and Asynchronous Algorithms In J. F. Traub (ed.),, *Algorithms and Complexity -- New Directions and Recent Results,* Academic Press, New York, pp. 153-200, 1976.

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

[19] Eva Ma and B. W. Wah, MANIP -- A Parallel Computer System for Solving NP-Complete Problems, *Proc. N. C. C. AFIPS Conf.,* 1981.

[20] T. A. Marsland and M. Campbell, Parallel Search of Strongly Ordered Game Trees, *Computing Surveys 14,* 4, pp. 533-551, 1982.

[21] L. G. Mitten, Branch and Bound Methods: General Formulations and Properties, *Operations Research 18,* pp. 23-34, 1970 Errata in Operations Research, 19 pp 550, 1971.

[22] D. S. Nau, V. Kumar, and L. N. Kanal, General Branch-and-Bound and its Relation to A* and AO*, submitted for publication. 1982.

[23] N. Nilsson, *Problem-Solving Methods in Artificial Intelligence,* McGraw-Hill Book Company, New York, 1971.

[24] N. Nilsson, *Principles of Artificial Intelligence,* Tioga Publ. Co., Palo Alto. CA, 1980.

[25] J. Pearl, Asymptotic Properties of Minimax Trees and Game-Tree Searching Procedures, *Artificial Intelligence 14,* pp. 113-138, 1980.

[26] I. Pohl, Is Heuristic Search Really Branch and Bound?, *Proc. Sixth Annual Princeton Cong. Inform. Sci. and Systems,* pp. 370-373, 1972.

[27] C. J. Rieger, ZMOB: A Mob of 256 Cooperative Z80A based Microcomputers, T.R.-825, Computer Science Dept., Univ. of Maryland, College Park, 1979.

[28] I. Roizen and J. Pearl, A Minimax Algorithm Better Than Alpha-Beta, Yes and No, *Artificial Intelligence 21,* pp. 199-220, 1983.

[29] J. R. Slagle, Heuristic Search Program, in R. Banerji and M. Mesarovic (eds.) , *Theoretical Approaches to Non-Numerical Problem Solving.,* New York, Springer Verlag , 1970.

[30] G. C. Stockman, A Problem-Reduction Approach to the Linguistic Analysis of Waveforms, Ph.D. Dissertation, TR-538, Comp. Sci. Dept., Univ. of MD, College Park, MD, May 1977.

[31] G. C. Stockman, A Minimax Algorithm Better than Alpha-Beta?, *Artificial Intelligence 12,* pp. 179-196, 1979.

[32] G. C. Stockman and L. N. Kanal, Problem Reduction Representation for the Linguistic Analysis of Waveforms., *IEEE Trans. on PAMI 5,* 3, pp. 287-298, 1983.

[33] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, Cm*- A Modular Multi Microprocessor., *AFIPS Conf. Proc. 46,* p. 637, 1977.

Table I: Summary of simulation results of the parallel implementation of section VI.B

S is the speedup of the parallel implementation (on n processors) over the sequential case for a RUG tree G(n,d) of branching degree n and depth d.

$N_{G(n,d)}$ is the number of terminal nodes evaluated by a (sequential) SSS* process in the minimax search of G(n,d).

$M_{G(n,d)}$ is the largest of the number of terminal nodes evaluated by concurrent dual-SS* processes searching the subtrees of G(n,d).

$\bar{S}_{10}$ is the average speedup over those 10 G(n,d) trees (out of 50 G(n,d) trees) for which sequential SSS* evaluates the largest number of nodes.

n,d	Speedup S			$N_{G(n,d)}$		$M_{G(n,d)}$		$\bar{S}_{10}$
	Average	Max	Min	Average	σ	Average	σ	
3,7	2.29	4.22	1.27	456	91.8	203	31.1	2.738
3,8	1.71	2.66	.96	873	204.78	520	66.8	2.178
3,9	2.11	3.65	1.23	2148	454.65	1043	153.9	2.810
5,5	3.14	5.58	1.84	626	162.35	203	28.82	4.235
5,6	2.25	4.10	1.20	1662	333.74	755	88.70	2.936
5,7	2.80	4.64	1.20	6037	1367.86	2193	249.92	3.856
8,3	4.95	6.74	2.74	187	31.53	38	3.32	5.960
8,4	2.92	4.39	1.83	693	142.5	239	22.17	4.059
8,5	4.09	6.10	2.10	3631	713.97	897	83.57	5.284

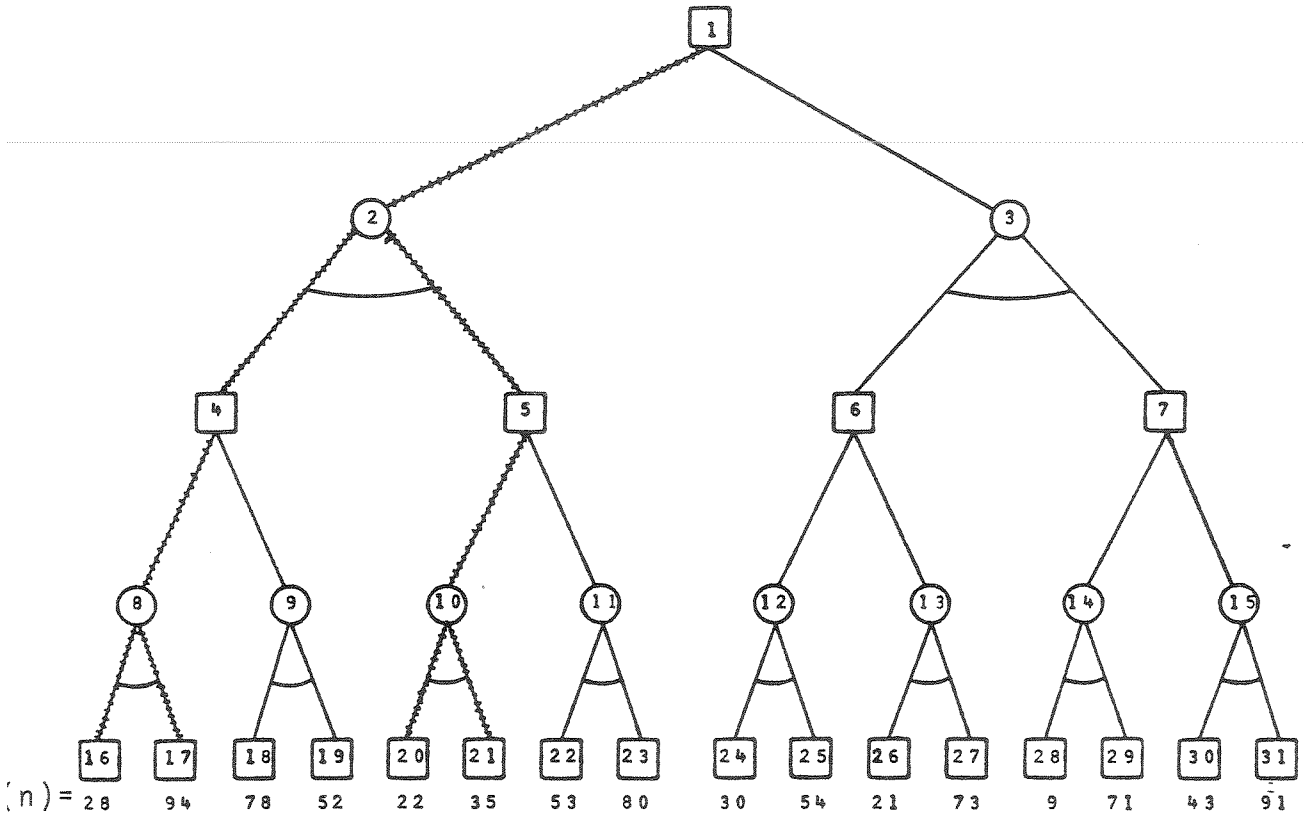

Fig. 1: An AND/OR tree G with hatch marks showing an AND-solution tree of G.

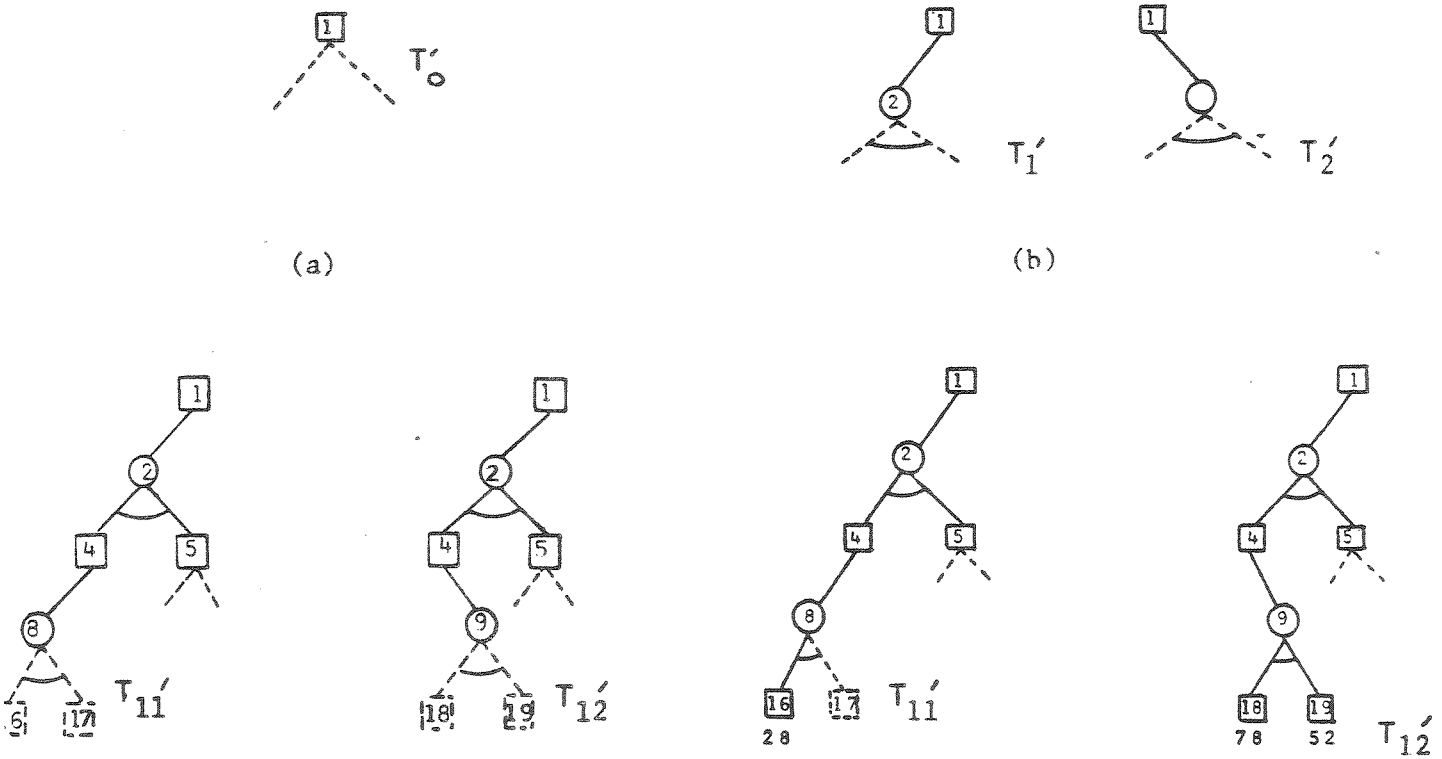

Fig. 2: Some steps of SSS* viewed as B&B search for a largest merit solution tree of the And/Or tree G of Fig. 1.

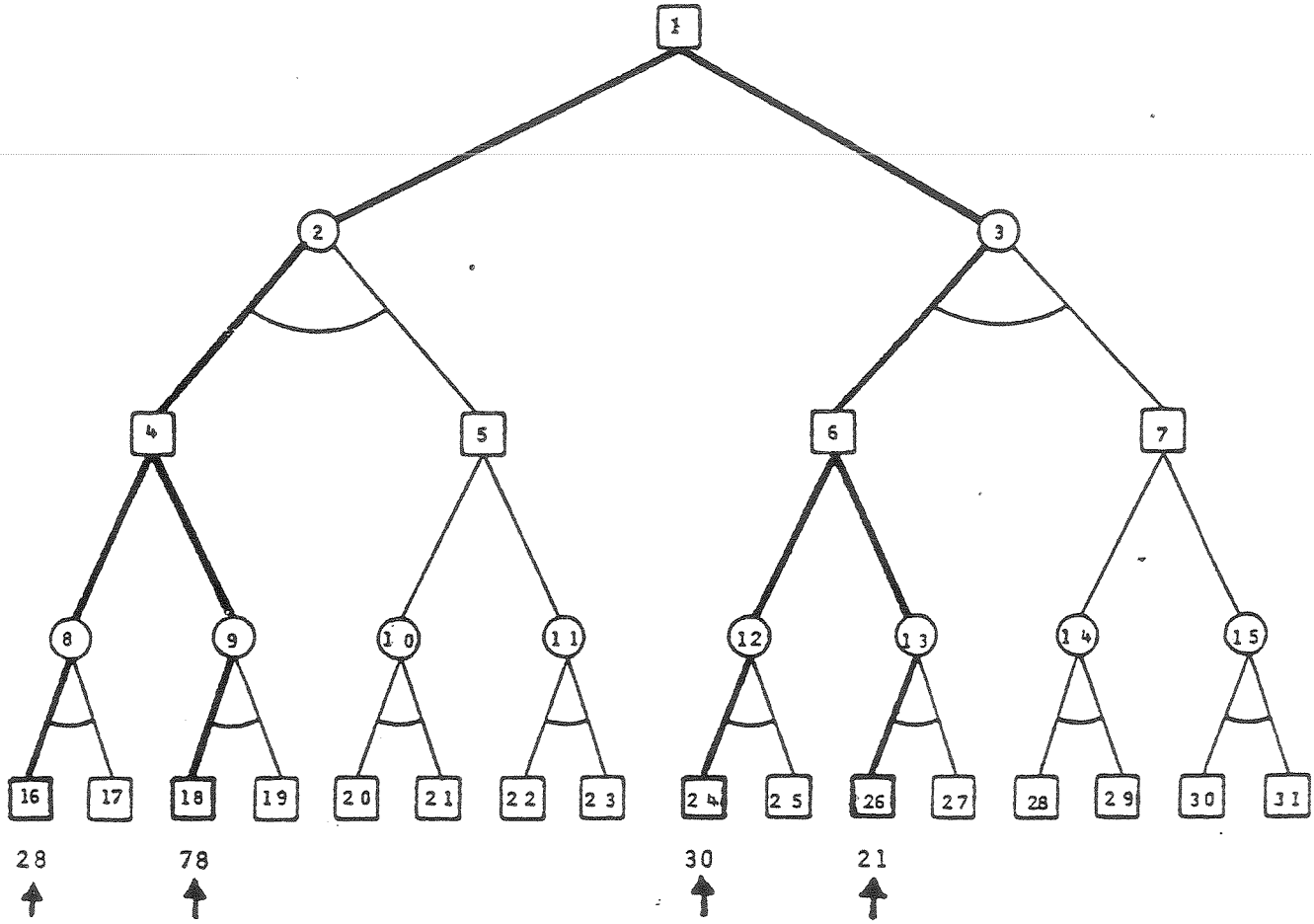

Fig. 3.a

Fig. 3 Interpretation of SSS* as a depth first B&B search in the space of OR solution trees of the And/Or tree G of Fig. 1.

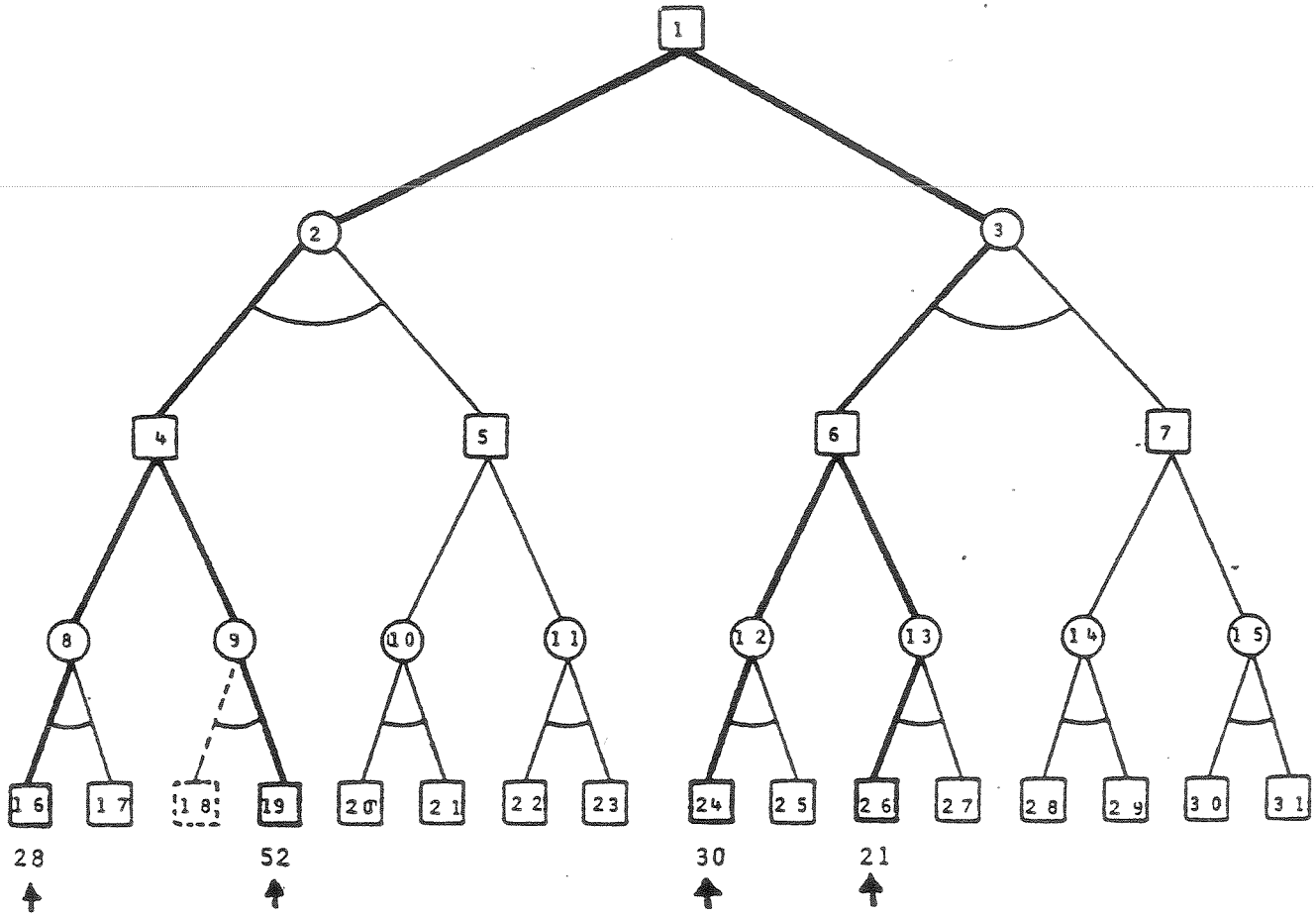
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
28
52
30
21

Fig. 3.b

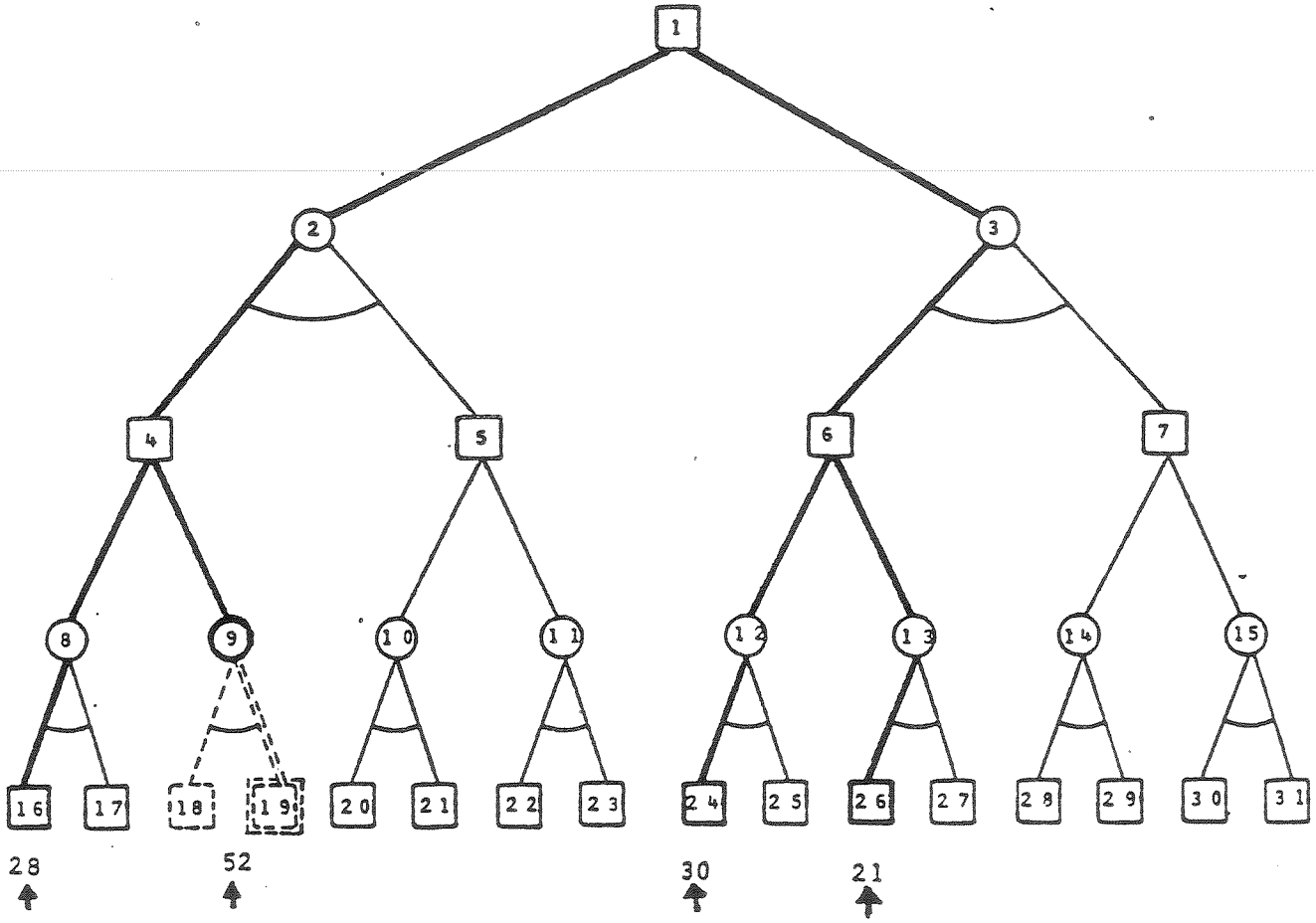
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
28
52
30
21

Fig. 3.c

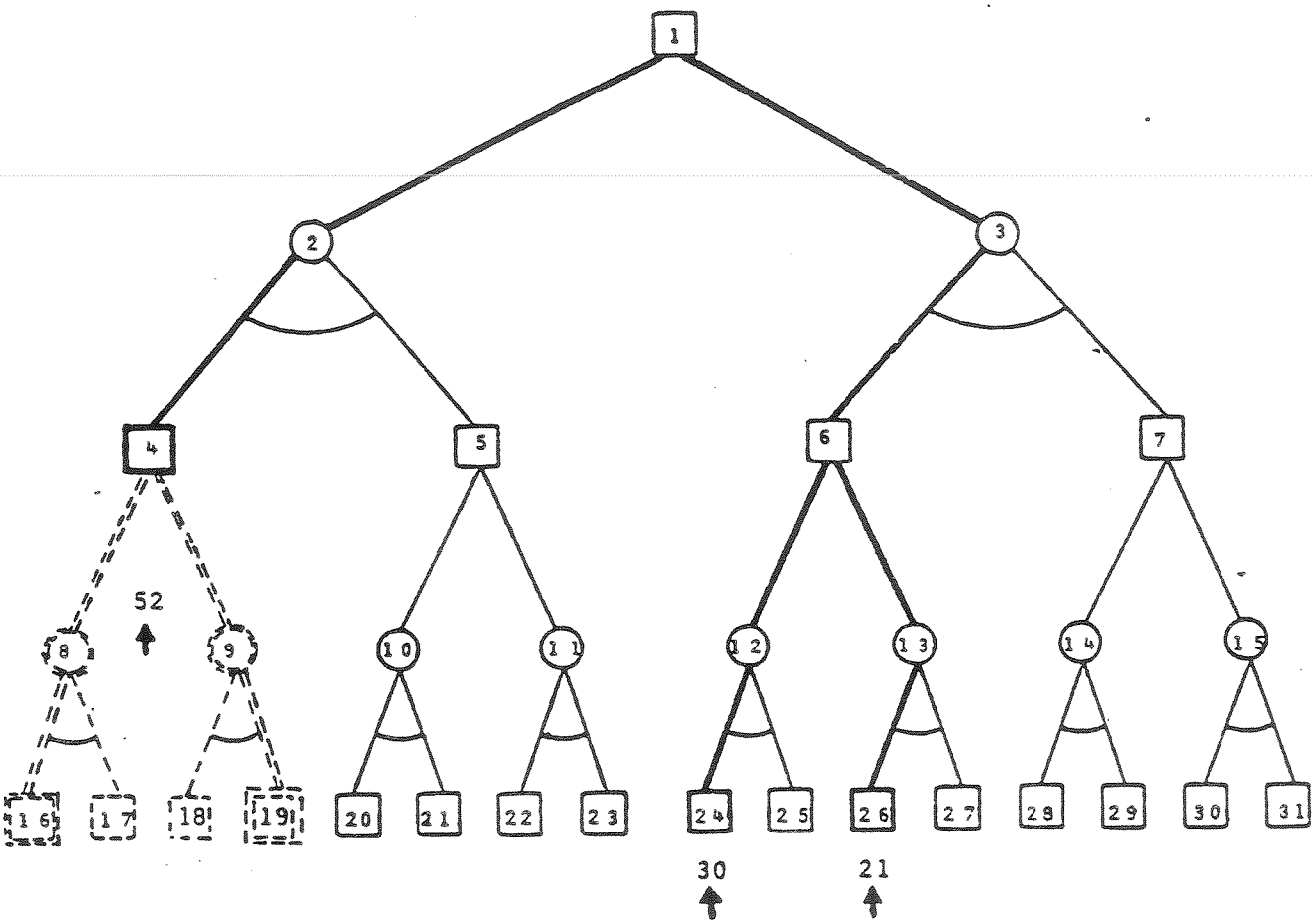
1
2
3
4
5
6
7
52
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
30
21

Fig. 3.d

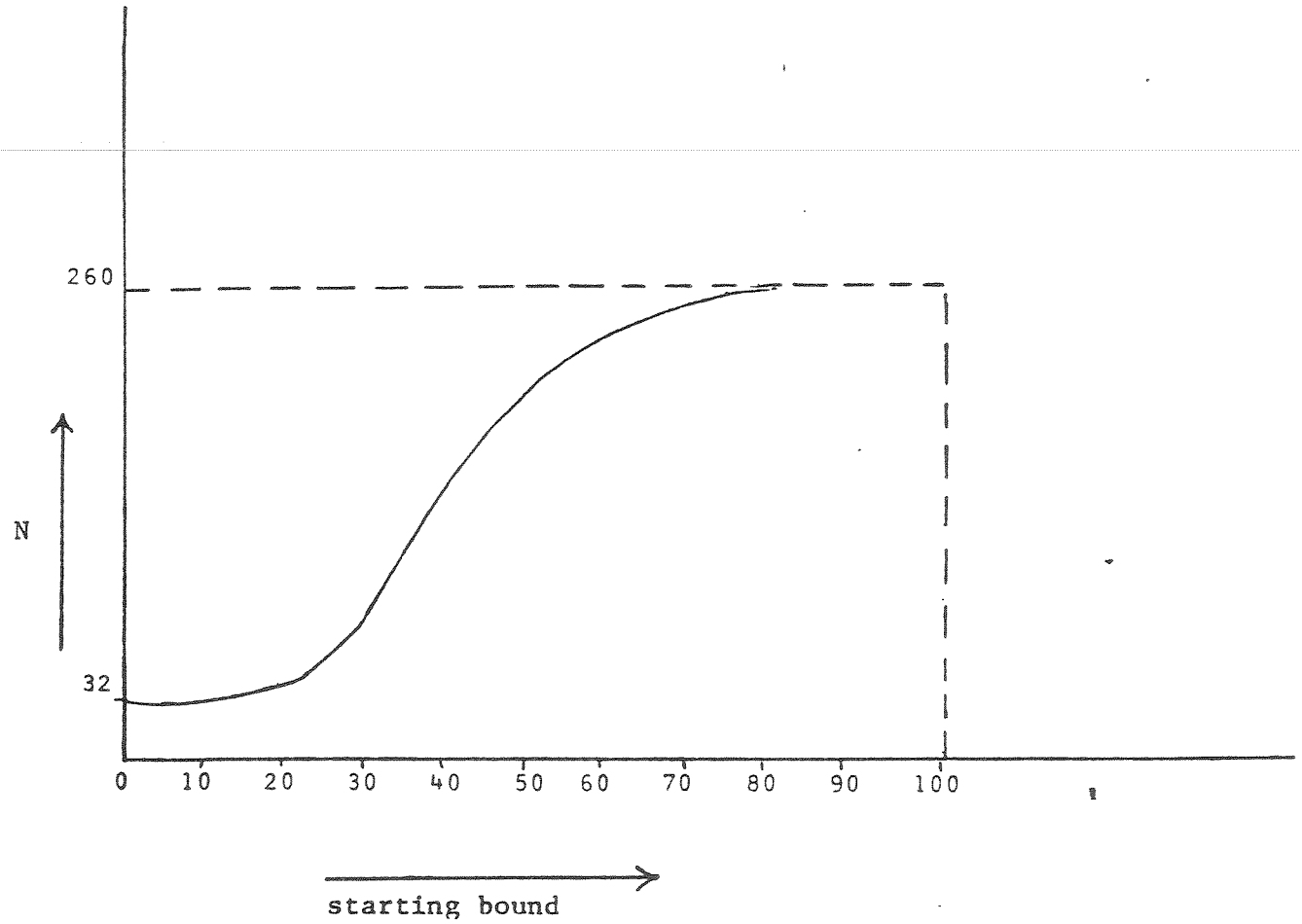

Fig. 4 Starting bound vs. number of tip nodes N evaluated by SSS*.

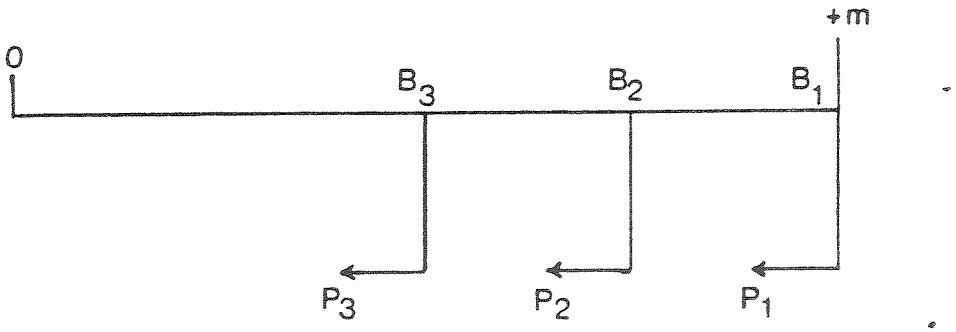

Figure 5 Depth first B&B processes searching for an optimum solution with different starting bounds. The process P_1 is started with bound B_i.