A General Branch and Bound Formulation for Understanding and Synthesizing And/Or Tree Search Procedures*

Vipin Kumar and Laveen N. Kanal
Laboratory for Pattern Analysis, Computer Science Department, University of Maryland, College Park, MD 20742, U.S.A.

ABSTRACT
Citing the confusing statements in the AI literature concerning the relationship between branch and bound (B&B) and heuristic search procedures we present a simple and general formulation of B&B which should help dispel much of the confusion. We illustrate the utility of the formulation by showing that through it some apparently very different algorithms for searching And/Or trees reveal the specific nature of their similarities and differences. In addition to giving new insights into the relationships among some AI search algorithms, the general formulation also provides suggestions on how existing search procedures may be varied to obtain new algorithms.

1. Introduction

Various heuristic procedures for searching And/Or graphs, game trees, and state space representations have appeared in the AI literature over the last few decades, and at least some of them have been thought to be related to the branch and bound (B&B) procedures of operations research. But the relationships between these two classes of procedures have been rather controversial. For example, Pohl argues in [23] that heuristic search procedures are very different from B&B procedures, whereas Hall [4] and Ibaraki [5, 7] claim that many heuristic procedures for searching state space representations are essentially B&B procedures. Knuth does not consider the alpha-beta game tree search algorithm to be a B&B procedure; he considers its less efficient version (called $F1$ in his classical treatment of alpha-beta [11]) to be branch and bound. But, Reingold et al. [24] consider alpha-beta to be a type of B&B. In the

*Research supported by NSF grants #ECS-7822159 and #MCS81-17391 to the Laboratory for Pattern Analysis

Artificial Intelligence 21 (1983) 179–198
0004–3702/83/0000–0000/$03.00

formal description of the game tree search algorithm B*, its author Berliner specifically states that "B* is not a B&B algorithm" [2]. While describing the algorithm HS (same as the AO* And/Or graph search algorithm [22]) in [16] Martelli and Montanari state that their algorithm is different from B&B because "(B&B) technique does not recognize the existence of identical subproblems". But Ibaraki's B&B procedure [7] for combinatorial optimization problems does recognize the existence of identical subproblems. It would seem that different people have differing notions of the words 'branch and bound'.

Part of the confusion and controversy can be explained on historical grounds. B&B techniques appear to have been conceptualized in the early 1960s to tackle integer programming and nonlinear assignment problems. Later similar techniques with some modifications were found to be applicable in many other problem domains. As more and more applications were discovered the B&B methodology evolved. Various formal models of B&B were presented and later superseded by more general models [1, 17, 12, 6, 7]. A lack of awareness of these later developments would be one explanation for some of the confusion noted above; the early survey by Lawler and Wood [14] is very often the only reference on B&B cited in papers on search in the AI literature.

It is easy to see that the central idea of B&B—the technique of branching and bounding to discover the optimum element of a set—is at the heart of many heuristic search procedures of AI. Confusion arises when one examines the specifics of B&B formulations presented in the literature. Even the later formulations, having been developed for specific problem domains, do not adequately model And/Or graph and game tree search procedures such as alpha-beta, SSS* [28], AO* and B*. For example, the characterization of B&B presented in [12] is developed in the context of permutation problems; while the one in [7] attempts to serve as a general model for state space search procedures.

Abstracting the essentials of B&B and dropping problem specific restrictions we have developed a formulation which is more general and also much simpler than existing formulations; the reader need only look at the previously cited references on B&B to be convinced of the latter claim. In this paper, we illustrate the utility of our B&B formulation by presenting SSS* as a B&B procedure for the search of an optimum solution tree of an And/Or tree, and showing how this formulation makes it possible to assess the precise relationship between SSS* and alpha-beta, two algorithms which hitherto were considered to be quite different.

SSS* was originally developed by Stockman [27] to find the largest merit solution of an And/Or graph when the merit is defined in a specific manner. Stockman noticed that, because of the correspondence between And/Or trees and game trees, SSS* can also be used to do minimax search of a game tree [28]. What came as a greater surprise was that SSS* outperforms alpha-beta in terms of the number of nodes expanded. The relative performance of SSS* with respect to

alpha-beta, according to various performance criteria, is a subject of continuing investigations [28, 25, 3].

One of the consequences of examining SSS* in our B&B framework is that if a minor modification is made in the B&B formulation of SSS*, the resulting procedure is equivalent to alpha-beta. This is most interesting, for alpha-beta as conventionally presented [11] appears very different from SSS* as described by Stockman [28]. Considering that alpha-beta has been known for over twenty years, it is noted worthy that SSS* was discovered only recently in the context not of game playing, but of a waveform parsing system [27]. Perhaps if an adequate B&B formulation for alpha-beta had been available earlier, SSS* would have been developed as a natural variation of alpha-beta.

This insight into the relationship between SSS* and alpha-beta is but one instance of the utility of our B&B formulation of SSS*. The formulation makes it easy to visualize other variations and parallel implementations of SSS*, some of which have been presented in Kanal and Kumar [8, 9].

We note that many other state space, And/Or graph, and game tree search procedures can also be formulated as B&B procedures in a manner similar to the formulation of SSS* described here. A brief outline of A* and AO* as branch and bound procedures can be found in [18].

Section 2 briefly introduces And/Or trees and their correspondence with game trees. Section 3 presents our abstract B&B formulation, and Section 4 presents SSS* as a B&B procedure. In Section 4 we also show how alpha-beta is related to SSS*, and briefly discuss what the B&B formulation reveals about the relationship of SSS* to AO* and B*. Section 5 contains concluding remarks.

2. And/Or Trees and their Correspondence with Game Trees

And/Or graphs provide graphical representations for problem reduction formulations. A detailed treatment of And/Or graphs can be found in [21, 22]. To keep the discussion simple in this paper we limit our presentation to And/Or trees. Many of the concepts and techniques presented are directly applicable to And/Or graphs. In this section we briefly review And/Or trees and their correspondence with game trees.

2.1. 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 solution to a problem whose (nonterminal) node has immediate successors of type OR is obtained by solving any one of the successors; while a solution to a problem whose node has immediate successors of type AND is obtained by solving all of these successors. Nodes with no successors are called *terminal*.

Given an And/Or tree representation of a problem we can identify its different solutions, each one 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 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 shows an And/Or tree and one of its solution trees. In many problems a merit is associated with solution trees.

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 of a terminal node n of G.

Merit function f. Let T be a solution tree of an And/Or tree G, then $f(T)$ is defined as the minimum merit of all terminal nodes in T, i.e., $f(T) = \min\{c(t) \mid t$ is a terminal node of $T\}$.

To facilitate evaluation of f-values of solution trees we define an evaluation function c_T for the nodes of a solution tree T.

Evaluation function c_T. If n is a node of a solution tree T, then $c_T(n)$ is defined as follows.

(i) If n has $n_1, \ldots, n_k$ as immediate successors of type AND,

$$c_T(n) = \min\{c_T(n_i) \mid 1 \leq i \leq k\}$$

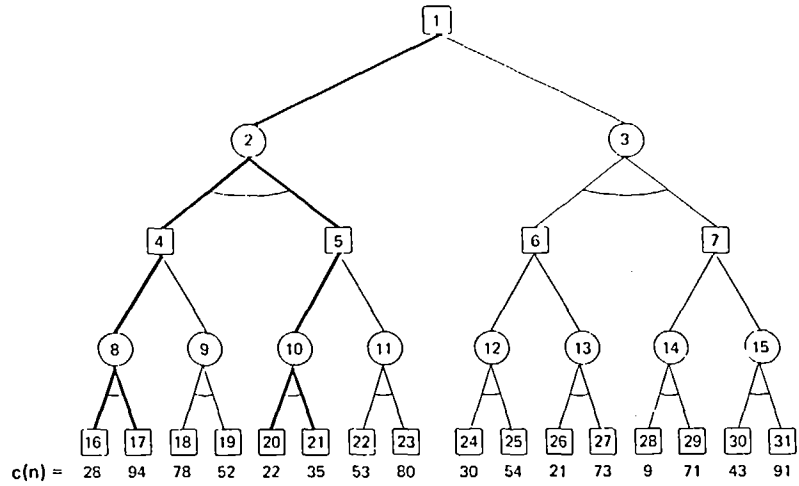

FIG. 1. An And/Or tree G. AND nodes are represented by square nodes, and OR nodes are represented by circle nodes. Boldface lines show a solution tree T of G; $f(T) = 28$.

(ii) If n has n_i as immediate successor of type OR,

$$c_T(n) = c_T(n_i).$$

(iii) If n is a terminal node,

$$c_T(n) = c(n).$$

Lemma 2.1. *If n is the root node of a solution tree T, then $c_T(n) = f(T)$.*

Proof. By induction on the height of T.

Hence, to calculate the f-value of a solution tree T with root node n we successively evaluate the c_T-value of its nodes until $c_T(n)$ is known. Among the different solution trees for a problem the solution tree with highest merit would then be chosen.

2.2. Correspondence with game trees

And/Or trees can also be used as models of two person, perfect information board games [21, 26]; 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 with 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,

$$g(n) = \max\{g(n_i)\} \quad \text{for all immediate successors } n_i \text{ of } n.$$

(ii) If n has immediate successors of type AND,

$$g(n) = \min\{g(n_i)\} \quad \text{for all immediate successors } n_i \text{ of } n.$$

(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 as $g(n)$, where n is the root node of G.

In [28] 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 2.1) of all solution 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.

3. A General and Bound Formulation

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 $f : X \to \mathbb{R}$, find some $x^* \in X$ such that for all $x \in X$, $f(x^*) \geq f(x)$.[1]

Branch and bound procedures decompose the original set into sets of decreasing size. The decomposition of each generated set 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.[2] 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 to generated.

In the earlier formulations of B&B [17, 1] 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

[1](a) 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. (b) In some problems all largest merit elements of X are desired. In this paper we restrict ourselves to the case when only one optimal element is needed.

[2]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.

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 [12, 6, 7]). The generalization involves using information about the sets other than just bounds (see, e.g., [18, 19]). 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 of $X_1 D X_2$, then X_2 can be pruned.

Here we briefly describe the basic elements of our branch and bound formulation. Our development of B&B has been greatly influenced by the earlier formulation due to Mitten [17]. As is discussed later, in our formulation, the dominance relation is used for pruning in a manner somewhat different than in [12, 6, 7].

Let Y be the set of all subsets of X, i.e., $Y = 2^X$. Symbol X_i denotes a subset of X, and symbol A denotes a collection of subsets of X, i.e., $A \subseteq Y$. For brevity, a collection A of the subsets of X will sometimes be referred to simply as a 'collection'. For notational convenience the union of all subsets in any collection A is denoted by $\bigcup(A)$, i.e., $\bigcup(A) = \{X_i \mid X_i \in A\}$. We define $f^*(X_i)$ to be the maximum of the merits of the elements in X_i. Any element $x^* \in X_i$ for which $f(x^*) = f^*(X_i)$ is called an *optimum element* of the set X_i.

We next define branching and pruning functions, and the dominance relation, and introduce the B&B procedure as an iterative procedure operating on sets.

Branching function BRANCH : $2^Y \to 2^Y$ is defined over collections A of subsets of X such that,

(i) $X_i \in \text{BRANCH}(A) \Rightarrow$ for some $X_j \in A$, $X_i \subseteq X_j$.

(ii) $\bigcup(\text{BRANCH}(A)) = \bigcup(A)$.

These conditions state that the branching rule divides the members of the collection A into subsets which collectively include precisely the same elements of X as the original collection A. From property (ii) of the function BRANCH we get the following result.

Lemma 3.1. $f^*(\bigcup(\text{BRANCH}(A))) = f^*(\bigcup(A))$.

Often, the function BRANCH is implemented such that only a selected member of the collection A is divided into subsets, and the rest are returned unchanged, i.e., $\text{BRANCH}(A) = (A - \text{SELECT}(A)) \cup \text{SPLIT}(A)$, where SELECT and SPLIT are defined as follows.

Selection function SELECT : $2^Y \to Y$ is a function which returns a subset X_i of X from the collection A, i.e., $\text{SELECT}(A) \in A$.

Splitting function SPLIT : $Y \to 2^Y$ splits a subset X_i of X into a collection of subsets of X_i, i.e., $X_i = \bigcup(\text{SPLIT}(X_i))$.

Dominance relation D is a binary relation defined between any subsets X_i, X_j

of X such that X_iDX_j if and only if $f^*(X_i) \geq f^*(X_j)$. From the definitions of f^* and D we obtain the following lemma.

Lemma 3.2. *Let A be a collection of subsets of X. If X_iDX_j and $X_i, X_j \in A$, then* $f^*(\bigcup(A)) = f^*(\bigcup(A - \{X_j\}))$.

The lemma says that if X_i and X_j are present in a collection A and X_i dominates X_j, then X_j can be eliminated from the collection A without lowering its optimum value.

Pruning function PRUNE : $2^Y \to 2^Y$ prunes the dominated subsets of A. It is defined as follows. PRUNE$(A) = A - A^D$ where A^D is a subset of A with the following property: for all $X_i \in A^D$ there exists some $X_j \in A - A^D$ such that X_jDX_i.

Hence, from Lemma 3.2, all the members of A^D can be eliminated from the collection A without lowering its f^*-value. This important result is summarized in the following lemma:

Lemma 3.3. $f^*(\bigcup(\text{PRUNE}(A))) = f^*(\bigcup(A))$.

We note that the use of dominance in our B&B formulation differs from that in earlier formulations. In the formulations presented in [12, 6, 7] sets previously generated and not currently in the collection are also allowed to participate in the test for dominance. When dominance is used in that way the cited formulations have to impose more restrictions (in addition to the definition) on D to ensure that Lemma 3.3 still remains valid (otherwise there exists, for example, the possibility of all members of the current set being pruned away because of dominance by different elements of the ancestor set). These restrictions tend to be dependent upon the problem domain being dealt with, and thus complicate the idea behind pruning by dominance. In [13] it is shown that the simpler pruning method used in the formulation presented here is no less general than pruning methods in the previous formulations.

Let $|S|$ denote the cardinality of a set S. Procedure P_0 given below represents the essence of many B&B procedures. Here, A denotes the collection of subsets of X upon which the branching and pruning operations are performed in each iteration of P_0.

```
procedure P0   (*B&B procedure to search for
                  an optimum element of a set X*)
begin
  A := {X};   (*initialize active collection A*)
  while |⋃(A)| ≠ 1 do (*loop until collection A has
                        only one element of set X*)
    A := BRANCH(A);   (*branch on the collection A*)
    A := PRUNE(A);  (*eliminate the dominated
                      subsets from A*)
  end;
end
```

Theorem 3.4. *When the procedure P_0 terminates for some x_0 in X, $\{x_0\} = \bigcup(A)$, and $f(x_0) = f^*(X)$ (i.e., at the termination the collection A contains only an optimum element of the set X).*

Proof. The predicate I_0, $\{f^*(\bigcup(A)) = f^*(X)\}$, is clearly established by the initialization statement ($A = \{X\}$) and (from Lemmas 3.1 and 3.3) is maintained by the operators BRANCH and PRUNE. Hence, I_0 is also true at the termination of P_0.

Also the predicate I_1, $\{|\bigcup(A)| = 1\}$, is true at the termination of the 'while-loop' of P_0. But, $\{|\bigcup(A)| = 1\}$ implies there exists an x_0 such that $\{x_0\} = \bigcup(A)$.

Hence, from I_0 and I_1, we conclude $f^*(X) = f(x_0)$.

Theorem 3.4 guarantees that, at termination, the collection A will contain only an optimum element of X; but the termination itself is not guaranteed. In general, we need to impose extra structure on the functions BRANCH and PRUNE to be able to guarantee the termination of P_0.

3.1. Discussion of the formulation and its relationship to previous work

In this abstract formulation a number of details have been left out. For example, we only define the basic properties of a branching function. In a practical implementation of a B&B procedure a branching function is chosen which is natural for the problem domain in question and satisfies the properties given here.

For pruning, in each cycle of P_0, a dominated subset A^D of the collection A needs to be constructed. Note that given any two subsets X_1, X_2 of X, at least one of them dominates the other (either $f^*(X_1) \geq f^*(X_2)$, or $f^*(X_2 \geq f^*(X_1))$. Hence, in theory, A^D can be constructed to have all but one set of the collection A. This would make the procedure P_0 terminate in a very few cycles, since in every cycle of P_0, all but one of the generated sets will be eliminated.

But we may not know the dominance relation for every pair of members of the collection A without exhaustively enumerating the elements in the sets which are members of A. However, in a particular problem, partial knowledge from the problem domain is often available to reveal the dominance relation between some of the members of A. This partial knowledge of the dominance relation can be used to construct A^D, a dominated subset of A. In the next section, where we present a practical B&B procedure to find an optimum solution tree of an And/Or tree, we show how knowledge of the problem domain is used to ascertain dominance between two sets of solution trees.

4. A B&B Procedure for Optimum Solution Tree Search

Here, the domain of elements X is the set of solution trees of an And/Or tree G. The merit function f for solution trees is defined in Section 2.1. In practical

implementations of B&B procedures the set X and its subsets are not represented explicitly. Instead, some problem specific data structure is used which implicitly represents the set X and its subsets. We shall use 'partial solutions trees' (also called partial trees) defined below, to represent sets of solution trees of G.

First, in Section 4.1 we introduce partial trees and define the function f^* and the dominance relation D on partial solution trees. Then, in Section 4.2 we present implementations of the functions SPLIT, SELECT, BRANCH and PRUNE to operate on partial trees and the collections of partial trees so as to have the same properties as in Section 3; we then present SSS* as a B&B procedure for the search of an optimum solution tree of an And/Or tree.

4.1. Partial solution trees—a representation for sets of solution trees

A *partial solution tree* (or partial tree) T' of an And/Or tree G is a subtree of G with the following properties.

(i) The root node of the And/Or graph G is the root node of the partial tree T'.

(ii) If any node (other than the root) of G is in T', then its ancestor is also in T'.

(iii) If a nonterminal node n of G is in T' and immediate successors of n in G are of type OR, then at most one of the immediate successors of n is in T'.

A partial solution tree is an 'incomplete' solution tree which can be extended (in possibly various ways) to form a complete solution tree. It represents all those solution trees which can be formed by extending it. We use S-TREES(T') to denote the set of solution trees represented by a partial tree T'. Fig. 2 shows a partial tree T' of the And/Or tree G of Fig. 1 and the set of solution trees represented by T'. A node of a partial tree is called a *tip* node, if it has no successors in T'. A tip node of T' is either a terminal node (if it has no successor in G), or a nonterminal (if it has successors in G). It follows that a partial tree all of whose tip nodes are terminal nodes in G represents just one solution tree, viz., itself.

For a node n of partial tree T', $c_T(n)$ (T is any solution tree in S-TREES(T')) can be evaluated by using the definition of c_T in Section 2.1, provided all successors of n (i.e., immediate successors, successors of immediate successors, ...) in G are also in T'. A node n of T' is called *solved* if $c_T(n)$ is known.

Define $f^*(T')$ to be the maximum of the merits of all solution trees represented by T', i.e., $f^*(T') = \max\{f(T) \mid T \in \text{S-TREES}(T')\}$. Let SOLVEDTIPS($T'$) and UNSOLVEDTIPS($T'$) represent the sets of solved and unsolved tip nodes of T' respectively. The following lemma can be proved by induction on the height of the tip nodes of T'.

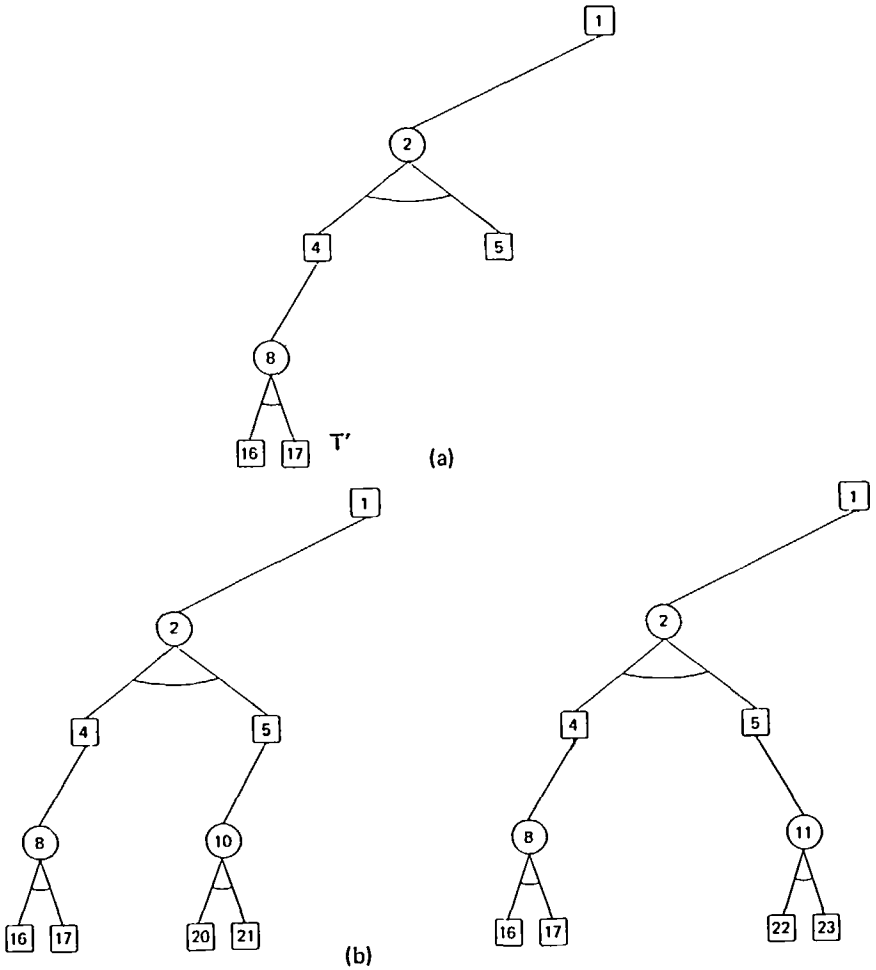

FIG. 2. (a) A partial tree T'. (b) Solution trees represented by T'.

Lemma 4.1.

$$f^*(T') = \min\{\min\{c(n) \mid n \in \text{SOLVEDTIPS}(T)\},\ \min\{g(n) \mid n \in \text{UNSOLVEDTIPS}(T')\}\}$$

where $g(n)$ *is the minimax value of node* n *as defined in Section* 2.2.

The dominance relation D between any partial trees T'_i and T'_j is defined as follows. $T'_i D T'_j$ if and only if $f^*(T'_i) \geq f^*(T'_j)$. Let us define the function *ubound* for any partial tree T' to be an upper bound on the merits of the

solution trees represented by T' as follows

$$\text{ubound}(T') = \min\{c(n) \mid n \in \text{SOLVEDTIPS}(T')\}.$$

Lemma 4.2. *For partial trees T'_i and T'_j, if* $\text{UNSOLVEDTIPS}(T'_i) \subseteq \text{UNSOLVEDTIPS}(T'_j)$ *and* $\text{ubound}(T'_i) \geq \text{ubound}(T'_j)$, *then $T'_i DT'_j$.*

Proof. From Lemma 4.1

$$\begin{aligned} f^*(T'_i) &= \min\{\min\{c(n) \mid n \in \text{SOLVEDTIPS}(T'_i)\}, \\ &\qquad \min\{g(n) \mid n \in \text{UNSOLVEDTIPS}(T'_i)\}\} \\ &= \min\{\text{ubound}(T'_i), \min\{g(n) \mid n \in \text{UNSOLVEDTIPS}(T'_i)\}\} \\ &\geq \min\{\text{ubound}(T'_j), \min\{g(n) \mid n \in \text{UNSOLVEDTIPS}(T'_j)\}\} \\ &= f^*(T'_j). \end{aligned}$$

Now, from the definition of 'dominance', we conclude $T'_i DT'_j$.

Note that given any two partial trees T'_i and T'_j of G, it is always possible, by exhaustively generating and evaluating all solution trees represented by T'_i and T'_j, to determine whether or not T'_i dominates T'_j. But Lemma 4.2 permits us to conclude that T'_i dominates T'_j without such exhaustive generation provided certain information about T'_i and T'_j is known. For example, using Lemma 4.2, we can conclude that in Fig. 3(d) T'_{11} dominates T'_{12}.

4.2. SSS* as a B&B procedure for And/Or tree search

Here, A will be used to represent a collection of partial solution trees (instead of a collection of sets of solution trees directly). For any collection A of partial trees, $\bigcup(A)$ denotes the union of sets of solution trees represented by the partial trees in A, i.e., $\bigcup(A) = \bigcup\{\text{S-TREES}(T'_i) \mid T'_i \in A\}$.

The function BRANCH takes a set of partial trees as input and returns another set of partial trees as output. It is implemented as a composition of two functions SELECT and SPLIT: $\text{BRANCH}(A) = (A - \text{SELECT}(A)) \cup \text{SPLIT}(\text{SELECT}(A))$.

The function SELECT is implemented to return a partial tree in A which has the largest upperbound. Hence, it has the following properties: (i) $\text{SELECT}(A) \in A$, and (ii) $\text{ubound}(\text{SELECT}(A)) \geq \text{ubound}(T'_i)$ for all $T'_i \in A$.

The function SPLIT takes a partial tree T' as an argument, and after expansion or evaluation of a node of T', returns the resulting (set of) partial tree(*s*). Since several nodes in T' may be available for expansion and evaluation, a certain order is followed. The first unevaluated node n of T' in post order[3] sequence (denoted as $\text{CURRENTNODE}(T')$) is selected; if n is a nonterminal tip node of T', then T' is extended by generating successors of the node n, otherwise n is evaluated. The reason for choosing this strategy for node selection is explained

[3]Post order is recursively defined as follows. First, visit sub-trees in left to right order, then visit the root node (see [10]).

later. Thus, SPLIT is implemented using functions EXPAND and EVALUATE as follows.

If n is a nonterminal tip node of T',
SPLIT(T') = EXPAND(T').
Otherwise (n is either a terminal node or has successors in T'),
SPLIT(T') = EVALUATE(T').

The function EXPAND returns the set of partial trees which results from expanding the selected node n of T'. If $n_1, \ldots, n_k$ are successors of n in G, then EXPAND(T') is defined as follows.

If n has OR successors,
EXPAND(T') = $\{T' * \langle n - n_i \rangle \mid 1 \leq i \leq k\}$
and CURRENTNODE($T' * \langle n - n_i \rangle$) = n_i.
Otherwise (n has AND successors),
EXPAND(T') = $\{T' * \langle n - n_1 \ldots n_k \rangle\}$ and
CURRENTNODE($T' * \langle n - n_1 \ldots n_k \rangle$) = n_1

Here $T' * \langle n - n_i \rangle$ denotes the extension of T' by including n_i as an immediate OR successor of n, and $T' * \langle n - n_1 \ldots n_k \rangle$ denotes the extension of T' by including $n_1, \ldots, n_k$ as immediate AND successors of n. Clearly,

$$\text{S-TREES}(T') = \bigcup \{\text{S-TREES}(T' * \langle n - n_i \rangle) \mid 1 \leq i \leq k\}$$

and

$$\text{S-TREES}(T') = \text{S-TREES}(T' * \langle n - n_1 \ldots n_k \rangle).$$

The function EVALUATE is implemented to return T' after evaluating the selected node n. As defined, the post order successor of n becomes the new CURRENTNODE(T'). Clearly, S-TREES(T') = S-TREES(EVALUATE(T')). If n is a terminal node of T', then the upper bound of T' is updated

$$\text{ubound}(\text{EVALUATE}(T')) = \min\{\text{ubound}(T'), c(n)\}.$$

Thus, viewed in entirety, the function BRANCH takes a set of partial trees as input, selects a largest ubound[4] partial tree T' from the set, chooses a nonterminal tip node n (=CURRENTNODE(T')) of T', and either evaluates n and return T', or expands n and returns the resulting extensions of T'. It follows that the function BRANCH, as defined here, has both the properties of its definition in Section 3.

The function PRUNE takes a collection A of partial trees as the argument, identifies the set of partial trees A^D (using lemma 4.2) such that each partial tree of A^D is dominated by some partial tree in $A - A^D$, and returns $A - A^D$, i.e., PRUNE(A) = $A - A^D$. Thus the function PRUNE eliminates only the dominated partial trees.

[4]In case there are more than one partial trees with the same largest ubound the one whose CURRENTNODE is in the left most position is selected.

Let T_0' be the partial tree containing only the root node of G (i.e., S-TREES(T_0') is the set of all solution trees of the And/Or tree G), then the following B&B procedure P_1 searches for the optimum solution tree of G. Here A denotes the collection of partial trees upon which branching and pruning operations are performed in each iteration of P_1.

```
procedure P1   (*B&B procedure for the search of optimum
                  solution tree of an And/Or tree G*)
begin
  A := {T0'};  (*initialize A with the complete
                  set of solution trees of G*)
  while |∪(A)| ≠ 1 do  (*repeat until A has just
                          one solution tree*)
    A := BRANCH(A);  (*select and split some set
                       of solution trees from A*)
    A := PRUNE(A);  (*remove the dominated
                      solution trees from A*)
  end;
end
```

The functions BRANCH and PRUNE implemented for collections of partial trees have the same properties as in Section 3; hence, from Theorem 3.4 it follows that, at the termination of the procedure P_1, the collection A will contain only an optimum solution tree of G. If G is finite it has only a finite number of solution trees each having only a finite number of nodes. In each iteration of procedure P_1, a node of some solution tree is either evaluated or expanded. Hence, the procedure will eventually terminate as all the solution trees will either be completely explored (and all except one will be pruned) or will be dominated by some other partial tree before complete evaluation.

Careful observation reveals that SSS* [28] is equivalent to the B&B procedure presented above. SSS* as presented in [28] maintains a list of states of traversal called OPEN; each of these states represents a partial tree (i.e., a set of solution trees) and the current upperbound associated with it. State selection and expansion directly correspond to the branching operation, and purging of states from OPEN corresponds to pruning dominated partial trees. The states eliminated from the OPEN list correspond precisely to the dominated partial trees found by the use of Lemma 4.2. Thus SSS* can be considered a practical implementation of the above B&B procedure. Fig. 3 provides an illustration of the similarity between the two.

Fig. 3(a) shows partial tree T_0' of the And/Or tree G of Fig. 1. T_0' consists of only the root node, and thus represents the total set of solution trees of G. Fig. 3(b) shows partial trees resulting from the branching operation on T_0', or equivalently from the expansion of node 1 in SSS*. Branching splits the total set of solution trees into two disjoint subsets now represented by partial trees T_1' and T_2'. Fig. 3(c) shows partial 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

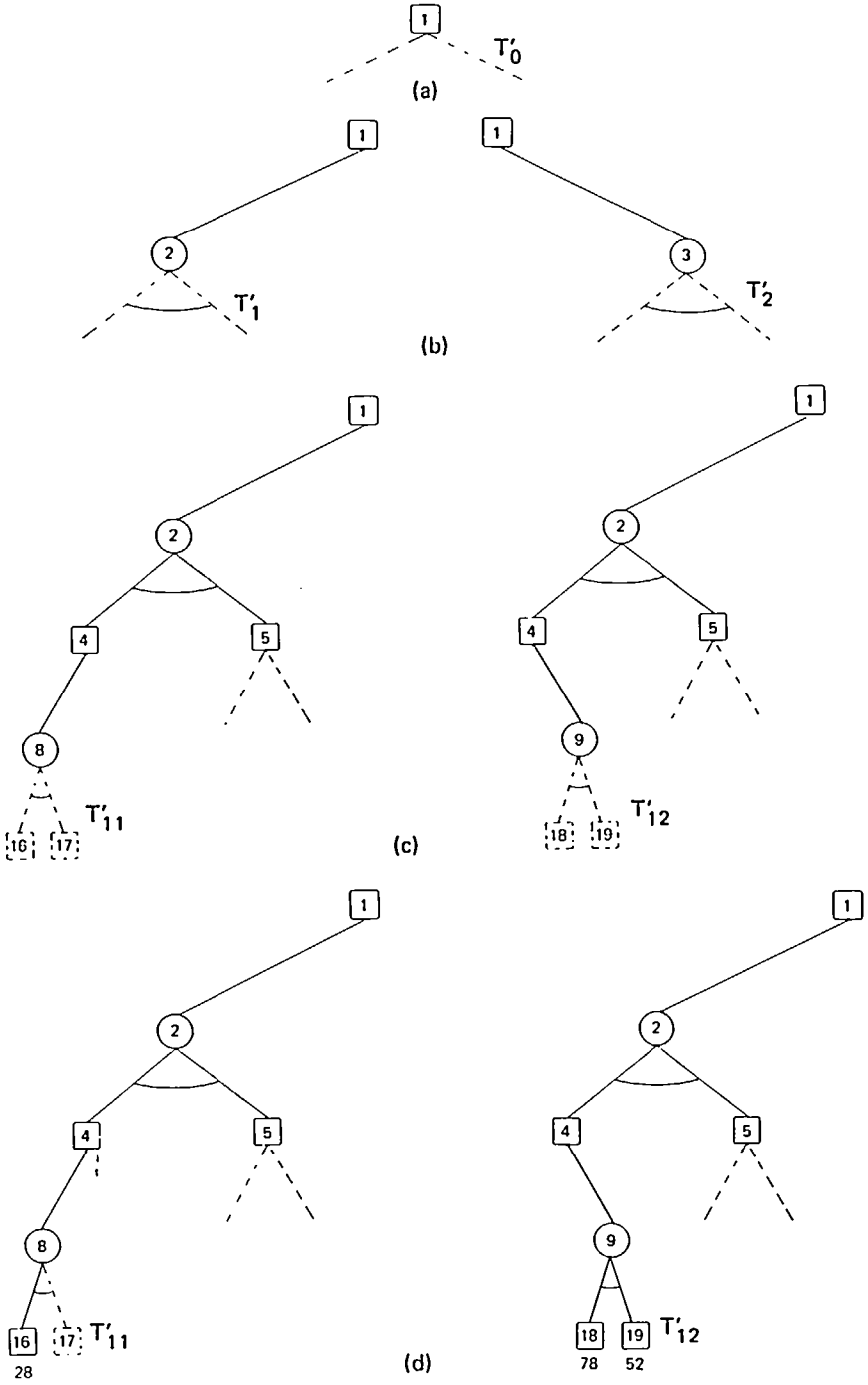

FIG. 3. Some steps of SSS* viewed as B&B search for an optimum solution tree of the And/Or tree G of Fig. 1.

SSS*. Fig. 3(d) shows partial trees T'_{11} and T'_{12} after evaluation of node 16 of T'_{11} and nodes 18, 19 of T'_{12}. From Lemma 4.2 T'_{12} dominates T'_{11}, hence T'_{11} is eliminated; equivalently, in SSS* node 17 is eliminated.

4.3. Relationship of SSS* to alpha-beta, B* and AO* in the light of the B&B formulation

Interestingly, if in the B&B formulation of SSS* the function SELECT(A) is modified to choose a partial tree whose CURRENTNODE is in the 'left-most' position in the explicit part of G, then the resulting procedure is equivalent to the well known alpha-beta procedure. This observation implies that alpha-beta (unlike SSS*) is not a best-first B&B procedure; hence, intuitively, it should expand more nodes than SSS*. Fig. 4 shows some steps of alpha-beta search reinterpreted as B&B search for an optimum solution tree of the And/Or tree G of Fig. 1.

The alpha-beta search would proceed in a manner similar to the SSS* search, starting from the partial tree T'_0 (as in Fig. 3(a)). The expansion of node 1 would split T'_0 into T'_1 and T'_2 (as shown in Fig. 3(b)). Further branching on T'_1 would result in T'_{11} and T'_{12} of Fig. 3(c). Now, due to the 'directional' nature of alpha-beta, the nodes 16, 17 and 8 of T'_{11} would be evaluated before the nodes 18, 19, 9 of T'_{12}. For this reason, alpha-beta, in this example, can not avoid evaluating node 17. Note that, by selecting a largest ubound partial tree for branching in each cycle, SSS* avoided evaluating node 17. After evaluating nodes 18 and 19 of T'_{12}, from Lemma 4.2 T'_{12} dominates T'_{11}. Fig. 4(a) shows partial tree T'_{121} and T'_{122} resulting from the branching operation on T'_{12} (the result of generating successors of node 5). Fig. 4(b) shows partial trees T'_{121} and T'_{122} after evaluation of nodes 20, 21 of T'_{121} and node 22 of T'_{122}. From Lemma 4.2 T'_{121} dominates T'_{122}; hence T'_{122} is eliminated; equivalently, a cutoff occurs at node 11, and node 23 is never evaluated by alpha-beta.

The algorithms AO* and B* can also be viewed as B&B procedures for searching for an optimal solution tree by appropriately defining the BRANCH and PRUNE functions (see [19, 13]). In AO* and B*, heuristic values are associated with unexplored nodes, which are used to associate bounds on the f^*-values of the partial trees on which branching and pruning is performed. In contrast in SSS* using the definition of ubound in Section 4.1, the upper bound on the f^*-values of a partial tree is computed solely on the basis of the merit of the evaluated nodes; it is assumed that an uninformed bound of $+\infty$ is associated with unevaluated nodes. In both SSS* and AO*, always a best bound (largest merit upper bound or least cost lower bound) partial tree is selected for further branching. The branching rule and termination criterion of the B&B formulation of B* are somewhat unconventional. This is because B*, being used for game playing, needs to find only the immediate OR successor of the root of an optimal solution tree of G, rather than a 'complete' optimum solution tree.

In general, in a partial tree selected for branching, several nodes may be

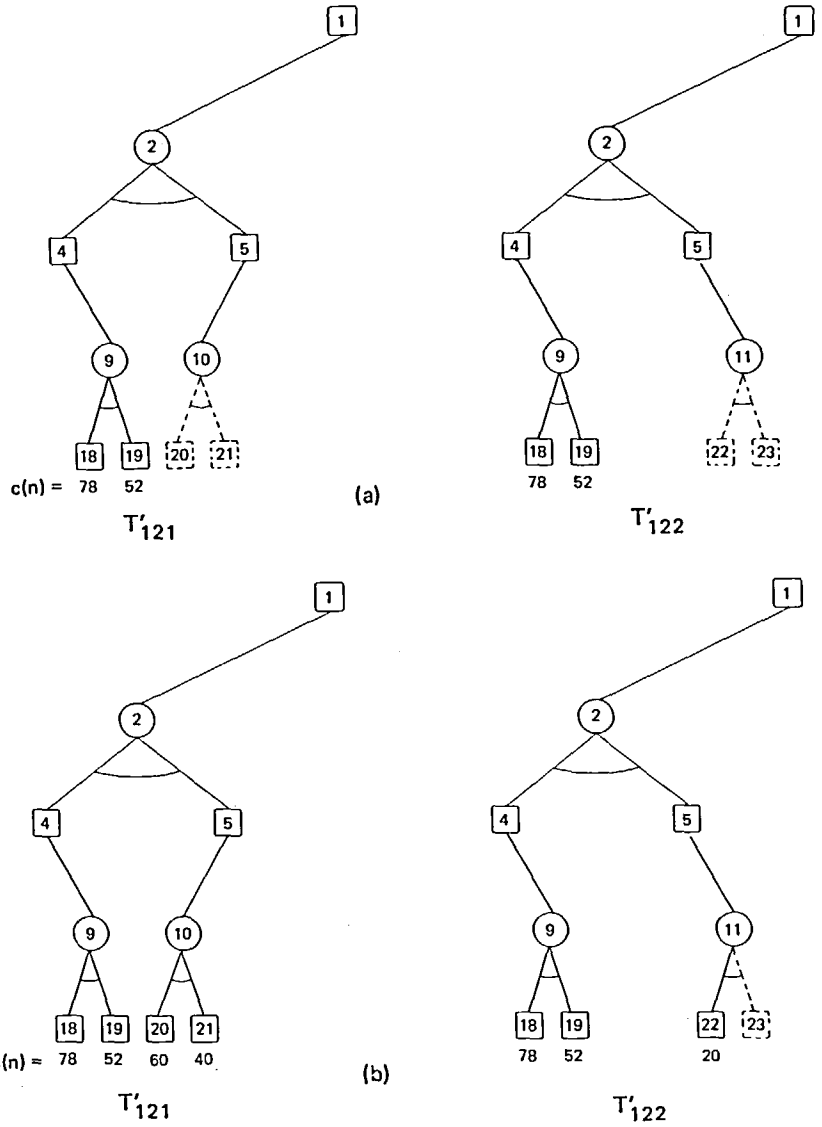

FIG. 4. Some steps of alpha-beta reinterpreted as B&B search for an optimum solution tree of the And/Or tree G of Fig. 1.

available for expansion or evaluation. The choice of which node to expand or evaluate next can significantly affect the time and space requirements of the algorithm, and is thus very important. Depending upon the available knowledge of the problem domain, different procedures use different kinds of heuristics for node selection. For example, in the B* algorithm for minimax evaluation of game trees [2] heuristic upper and lower bounds on the node

values are used to decide which node to expand next. Similarly in AO*, heuristic bounds on the nodes are used in the selection criterion.

SSS* (and alpha-beta) does not assume availability of such heuristic information about the tip values except in ordering successors after a node expansion. But, SSS* follows a certain order in expanding and evaluating nodes of the selected partial trees. Due to the dominance relation[5] given by Lemma 4.2 this ordering allows representation of a partial tree by a 3-tuple for the purpose of node selection. The 3-tuple $\{n, s, h\}$ (introduced in the treatment of SSS* by Stockman [28]) summarizes the state of a partial tree T'; n denotes the CURRENTNODE(T') in the present or previous iteration of SSS* depending upon whether the status s is LIVE or SOLVED, and h denotes the upper bound on the merits of the solution trees represented by the partial tree T'. These 3-tuples are kept in an ordered (on h) list, and always the partial tree corresponding to the 3-tuple on the top of the list is chosen for branching. Thus, the 3-tuple representation makes the process of selection of a node for expansion in SSS* extremely efficient compared to, for instance, AO* which needs an elaborate mechanism of arrows (see [16, 20]) to select a node for expansion. Note that even though a 3-tuple $(\cdot, \cdot, \cdot)$ completely characterizes the state of a partial tree, the (undominated) partial trees of the And/Or graph G still need to be explicitly stored so that (i) in every cycle the determination of next CURRENTNODE can be done, (ii) at the termination of P_1 an optimum solution tree can be found.

5. Concluding Remarks

We have shown that by abstracting the core of B&B and divorcing it from problem specifics, a general B&B formulation is obtained within which seemingly very different algorithms reveal their essential similarities and differences. Because many of the existing search procedures were developed for problems in different fields by authors with differing backgrounds and perspectives, it is not surprising that the essential nature of these procedures is masked by problem specific details. Looking at the descriptions of SSS* and alpha-beta in [28] and [11] who would have thought they are close cousins.

The purpose of a general formulation has to be to provide a greater insight than available from previous formulations. As should be evident from this paper, our general formulation has led to new insights into the relationship between some search algorithms and has also provided suggestions on how the existing search methods may be varied to provide new algorithms. We believe that through this new formulation, we have dispelled some of the confusion appearing in the literature concerning the nature of B&B procedures and their relationship to AI search techniques.

[5]Besides minimum function there are other merit functions (e.g., additive cost function used in [15]) for which a dominance relation can be obtained.

This work has encouraged us to seek a more general formulation which would help clarify the relationships between B&B, dynamic programming, and a wider variety of AI heuristic search procedures. Such a formulation is currently under investigation [13].

REFERENCES

1. Balas, E., A note on the branch-and-bound principle, *Oper. Res.* **16** (1968) 442–444, 886.
2. Berliner, H., The B* tree search algorithm: a best-first proof procedure, *Artificial Intelligence* **12** (1979) 23–40.
3. Campbell, M., Algorithms for the parallel search of game trees, Tech. Rept. 81-8 Dept. Computer Science, University of Alberta, Edmonton, 1981.
4. Hall, P.A.V., Branch-and-bound and beyond, *Proc. Second Internat. Joint Conf. Artificial Intelligence*, London (1971) 641–658.
5. Ibaraki, T., On the optimality of algorithms for finite state sequential decision processes, *J. Math. Anal. Appl.* **53** (1976) 618–643.
6. Ibaraki, T., The power of dominance relations in branch and bound algorithms, *J. ACM* **24** (1977) 264–279.
7. Ibaraki, T., Branch-and-bound procedure and state-space representation of combinatorial optimization problems, *Inform. Control* **36** (1978) 1–27.
8. Kanal L.N. and Kumar, V., A branch and bound formulation for sequential and parallel game tree searching, *Proc. Seventh Internal. Joint Conf. Artificial Intelligence*, Vancouver (1981) 569–571.
9. Kanal L.N. and Kumar, V., Parallel implementations of a structural analysis algorithm, *Proc. IEEE Computer Society Conf. Pattern Recognition and Image Processing*, Dallas (1981) 452–458.
10. Knuth, D.E., *The Art of Computer Programming*—Vol. 1 (Addison-Wesley, Reading, MA, 1968).
11. Knuth D.E. and Moore, R.W., An analysis of alpha-beta pruning, *Artificial Intelligence* **6** (1975) 293–326.
12. Kohler W.H. and Steiglitz, K., Characterization and theoretical comparison of branch and bound algorithms for permutation problems, *J. ACM* **21** (1974) 140–156.
13. Kumar, V., A unified approach to problem solving search procedures, Ph.D. Thesis, Dept. Computer Science, University of Maryland, College Park, MD, 1983.
14. Lawler E.L. and Wood, D.E., Branch-and-bound methods: a survey, *Oper. Res.* **14** (1966) 699–719.
15. Martelli A. and Montanari, U., Additive AND/OR graphs, *Proc. Third Internat. Joint Conf. Artificial Intelligence*, Stanford, CA (1973) 1–11.
16. Martelli A. and Montanari, U., Optimizing decision trees through heuristically guided search, *Comm. ACM* **21** (1978) 1025–1039.
17. Mitten, L.G., Branch and bound methods: general formulations and properties, *Oper. Res.* **18** (1970) 23–34; Errata in *Oper. Res.* **19** (1971) 550.
18. Nau, D.S., Kumar, V. and Kanal, L.N., A general paradigm for AI search algorithms, *Proc. of AAAI-82*, Pittsburgh, PA, 1982.
19. Nau, D.S., Kumar, V. and Kanal, L.N., General branch-and-bound and its relation to A* and AO* *Artificial Intelligence* (1983) submitted.
20. Nilsson, N., Searching problem solving and game playing trees for minimum cost solutions, in: A.J.H. Morrel (Ed.), *Information Processing-68* (North-Holland, Amsterdam, 1968).
21. Nilsson, N., *Problem-Solving Methods in Artificial Intelligence* (McGraw-Hill, New York, 1971).
22. Nilsson, N., *Principles of Artificial Intelligence* (Tioga, Palo Alto, CA, 1980).
23. Pohl, I., Is heuristic search really branch and bound?, *Proc. Sixth Annual Princeton Conf. Information Science and Systems*, Princeton, NJ (1972) 370–373.

24. Reingold, E., Nievergelt, J. and Deo, N., *Combinatorial Optimization* (Prentice-Hall, Englewood Cliffs, NJ, 1977).
25. Roizen I. and Pearl, J., A minimax algorithm better than alpha-beta? yes and no, *Artificial Intelligence* **21** (1, 2) (1983) 199–220.
26. Slagle, J.R., Heuristic search program, in: R. Banerji and M. Mesarovic (Eds.), *Theoretical Approaches to Non-Numerical Problem Solving* (Springer Verlag, Berlin, 1970).
27. Stockman, G.C., A problem-reduction approach to the linguistic analysis of waveforms, Ph.D. Dissertation, TR-538, Computer Science Dept., University of Maryland, College Park, MD, 1977.
28. Stockman, G.C., A minimax algorithm better than alpha-beta?, *Artificial Intelligence* **12** (1979) 179–196.

Received August 1982; revised version received October 1982