., The composite decision process: A
euristic search, dynamic programming
rocedures, Proceedings of The 1983
elligence, 220-224.

E.N., Variable length encodings, *Bell*
933-968.

nd Brown, C. A., *The analysis of*
and Winston, New York, New York.

rogramming is optimal for nonserial
M J. Comput **11** (1982), 47-59.

f Database Systems, Computer Science
1982.

ORIGINAL

A GENERAL BRANCH-AND-BOUND FORMULATION FOR AND/OR GRAPH AND GAME TREE SEARCH[1]

Vipin Kumar
Artificial Intelligence Laboratory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Dana S. Nau and Laveen N. Kanal
Machine Intelligence and Pattern Analysis Laboratory
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

This paper presents a general procedure for finding an optimal solution tree of an acyclic AND/OR graph with monotone cost functions. Due to the relationship between AND/OR graphs and game trees, it can also be used as a game tree search procedure. Seemingly disparate procedures like AO*, SSS*, alpha-beta, B* are instantiations of this general procedure. This sheds new light on their interrelationships and nature, and simplifies their correctness proofs. Furthermore, the procedure is applicable to a very large class of problems, and thus provides a way of synthesizing algorithms for new applications. The procedure searches an AND/OR graph in a top-down manner (by selectively developing various potential solutions) and can be viewed as a general branch-and-bound procedure.

---

[1] This chapter is a revised version of Chapter 6 of the first author's 1982 PhD dissertation [10].

This work was supported in part by Army Research Office grant #DAAG29-84-K-0060 to the Artificial Intelligence Laboratory at the University of Texas at Austin, and in part by NSF grants to the Machine Intelligence and Pattern Analysis Laboratory at the University of Maryland.

1. INTRODUCTION

This paper presents a general procedure for finding an optimal solution tree of an acyclic AND/OR graph with monotone cost functions. Due to the relationship between AND/OR graphs and game trees, it can also be used as a game tree search procedure. This general formulation has the following advantages:

(i) Seemingly disparate procedures like AO* [23], SSS* [28], alpha-beta [8], B* [5] are instantiations of this general procedure. This sheds new light on their interrelationships and nature, and simplifies their correctness proofs.

(ii) More importantly, the procedure is applicable to a very large class of problems, and thus provides a way of synthesizing algorithms for new applications.

The procedure searches an AND/OR graph in a top-down manner (by selectively developing various potential solutions) and can be viewed as a general branch-and-bound (B&B) procedure [12]. This is noteworthy, as the relationship between B&B and AND/OR graph search procedures such as AO* has been quite controversial (see [10], [13]).

We earlier developed an abstract model of B&B which is more general than previous B&B formulations, and powerful enough to incorporate essentials of a number of AI search algorithms. We have previously shown that a number of AND/OR graph and game tree search procedures (e.g., AO*, SSS*) are essentially B&B [13], [20]. Viewing these procedures from a common perspective has given us insights into their basic nature, and has helped us synthesize the general procedure described in the current paper. This procedure is applicable to a large number of problems.

In Section 2 we briefly introduce AND/OR trees, and discuss their correspondence with game trees. In Section 3 we present an abstract B&B formulation. In Section 4 we introduce a general B&B formulation for searching acyclic AND/OR graphs. In Section 5 we develop this B&B formulation further and show that AO* and some of its variations presented in [2] are special cases of this formulation. In Section 6 we present variations of the formulation of Section 5 and show that B*, SSS*, and alpha-beta are special cases of these variations. Section 7 contains concluding remarks. A list of the definitions of the terms used in the paper appear in the appendix.

2. AND/OR GRAPHS AND THEIR RELATIONSHIP TO GAME TREES

A problem reduction representation (PRR) is a representation of how a problem might be solved recursively by transforming it into several simpler equivalents, such that the original problem may be solved by solving any one of them, or by transforming the problem into several subproblems such that the original problem may be solved by solving all of the subproblems. PRRs are modeled by AND/OR graphs, as described in detail in [22], [4]. Here, we briefly review AND/OR graphs and their correspondence with game trees.

2.1. AND/OR Graphs

Each node of an AND/OR graph represents a problem, and a special node root(G) called *root* of G represents the original problem to be solved. Nodes having children are called *nonterminal*. By convention, the children of each nonterminal node are either all of type AND or all of type OR. The hypergraphs of [17], [23] (in which nodes have both kinds of children) can be converted into AND/OR graphs by introducing extra dummy nodes. Let

$$p{:}n \rightarrow n_1,\ldots,n_k$$

be a problem transformation. If p is such that all of the problems $n_1,\ldots,n_k$ need to be solved to solve the problem n, then p is called a *reduction* and $n_1,\ldots,n_k$ are depicted as AND children of n in the AND/OR graph. If p is such that the problem n may be solved by solving any one of the problems $n_1,\ldots,n_k$, then $n_1,\ldots,n_k$ are depicted as OR children of n in the AND/OR graph. Nodes with no children are called *terminal*, and each terminal node represents a primitive problem. An AND/OR graph G is *acyclic* if no node of G is a successor of itself. An AND/OR graph G is called an AND/OR tree if G is acyclic and every node except root(G) has exactly one parent. Every acyclic AND/OR graph G can be "unfolded" (by creating duplicates of all nodes of G having multiple parents) to build an equivalent AND/OR tree called unfold(G).

Given an AND/OR graph representation of a problem, one can identify its different solutions, each one represented by a "solution tree". A *solution tree* T of an AND/OR graph G is an AND/OR tree with the following properties:

(i) root(T) = root(unfold(G)).

(ii) if a nonterminal node n of unfold(G) is in T, then all of its children are in T (as AND children of n) if they are of type AND, and exactly one of its children is in T (as an OR child of n) if they are of type OR.

To distinguish solution trees from other entities to be defined later, we will sometimes call them *total* solution trees.

A solution tree T of G represents a plausible "problem reduction scheme" for solving the problem modeled by the root node of G. Clearly, an acyclic AND/OR graph can have only a finite number of solution trees. The subgraph G'_n of G rooted at a node n is in fact a problem reduction formulation of the problem represented by n, and a solution tree of G'_n represents a solution to that problem. By a solution tree rooted at n we mean a solution tree of G'_n.

Often, a cost function f is defined on the solution trees of G, and a least-cost solution tree of G is desired. There are various ways in which this cost function can be defined, but the one defined below is applicable to a large number of problems.[2]

For a terminal node n of G, let c(n) denote the cost of n, i.e., the cost of solving the problem represented by n. With each reduction $p: n \rightarrow n_1,...,n_k$ we associate a k-ary cost function $t_p(r_1,...,r_k)$ which denotes the cost of solving n if n is solved by solving $n_1,...,n_k$ at costs $r_1,...,r_k$, respectively.

For a solution tree T, we define its cost f(T) recursively as follows:

2.1a if T consists only of a single node n = root(T), then f(T) = c(n).

2.1b If n = root(T) has AND-children $n_1,...,n_k$ such that $p: n\ n_1,...,n_k$ is a reduction, then $f(T) = t_p(f(T_1),...,f(T_k))$, where $T_1,...,T_k$ are the subtrees of T rooted at $n_1,...,n_k$.

2.1c If n = root(T) has n_i as the OR child in T, then $f(T) = f(T_i)$, where T_i is the subtree of T rooted at n_i.

[2] The definition of cost function given here is similar to the definition of recursive weight

Thus the cost of a solution tree is defined recursively as a composition of the cost of its subtrees. Fig. 1 shows an acyclic AND/OR graph, associated cost functions, and the computation of the cost of one of its solution trees. We define $c^*(n)$ for nodes n of an AND/OR graph G to be the minimum of the costs of the solution trees rooted at n. Then $c^*(root(G))$ is the cost of an optimum solution tree of G. The following theorem provides a way of computing $c^*(n)$ for nodes n of an acyclic AND/OR graph.[3]

Theorem 2.1: If the functions $t_p(.,...,.)$ are monotonically nondecreasing in each variable, then the following recursive equations hold.

(i) If n is a terminal node, then

$$c^*(n) = c(n).$$

(ii) If $p:n \rightarrow n_1,...,n_k$ is a reduction (i.e., $n_1,...,n_k$ are AND children of n), then

$$c^*(n) = t_p(c^*(n_1),...,c^*(n_k)).$$

(iii) If n has $n_1,...,n_k$ as OR children, then

$$c^*(n) = \min\{c^*(n_1),...,c^*(n_k)\}.$$

Proof: By induction on the height of n.

2.2 Maximization problems and Game Trees

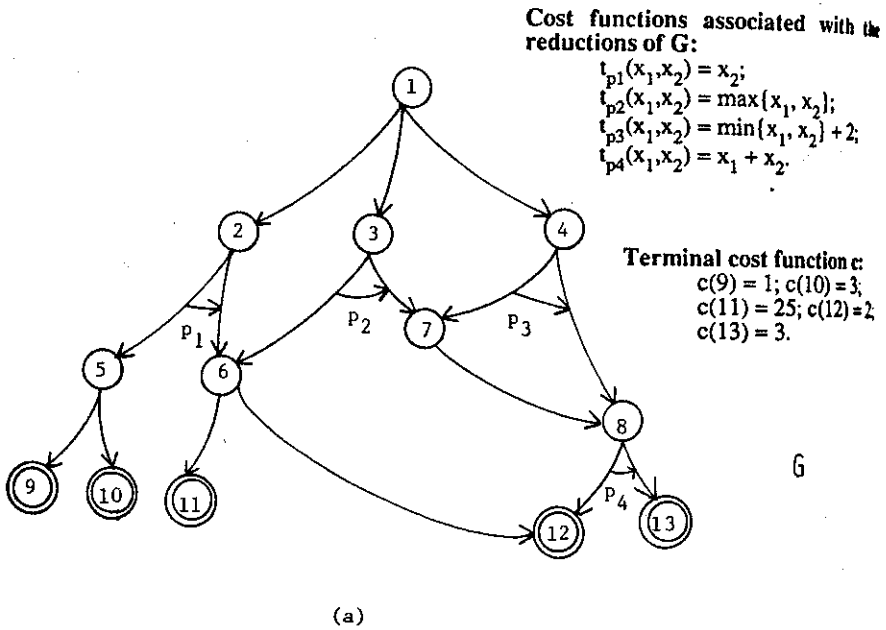
In many problem domains, f(T) denotes the merit of the solution tree T, and a solution tree of largest merit is desired. In such cases, c(n) denotes the merit of a terminal node n of G. The functions $t_p(.,...,.)$, and f are defined exactly as before, but $c^*(n)$ denotes the maximum of the merits of the solution trees rooted at n. In this case, Theorem 2.1 can be restated with its third condition replaced by

(iii') If n has $n_1,...,n_k$ as OR children then

$$c^*(n) = \max\{c^*(n_1),...,c^*(n_k)\}$$

One such case is that of two-person games.

[3] The theorem is valid for cyclic AND/OR graphs as well, but the proof, which appears in [10], is more complicated.



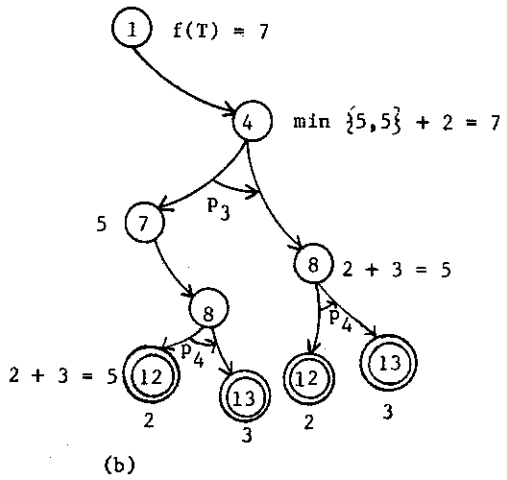


Fig. 1 (a) An AND/OR graph G and the associated cost functions. Terminal nodes of G are denoted by double circles.

(b) Computation of f(T) of a solution tree T of G

AND/OR trees can also be used as models of two-person, perfect information, zero sum board games [22], [26]. (For example, the AND/OR tree of Fig. 2 can be viewed as a game tree.) Board positions resulting from one player's moves are represented by OR nodes (circular nodes in Fig. 2), and board positions resulting from the other player's moves are represented by AND nodes (square nodes in Fig. 2). The two players are called MAX and MIN, respectively. Moves of the game proceed in strict alternation between MAX and MIN, until the game ends. After the last move, MAX receives a payoff c(n) which is a function of the final board position n, and MIN has to forfeit the same amount. Thus, MAX always seeks to maximize the payoff while MIN seeks to minimize it. 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 from any board position is given by the minimax value g(n) defined recursively as follows [8]:

(i) If n has children of type OR, then
$g(n) = \max\{g(n_i)\}$ for all children n_i of n.

(ii) If n has children of type AND, then
$g(n) = \min\{g(n_i)\}$ for all children n_i of n.

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

If for every reduction p: $n \rightarrow n_1,...,n_k$, we define $t_p(r_1,...,r_k) = \min\{r_1,...,r_k\}$, then it follows (from Theorem 2.1 using (iii')) that for every node n of G, $c^*(n) = g(n)$. More specifically, $c^*(\text{root}(G))$, the maximum of the merits of the solution trees of G, is equal to g(root(G)), the minimax value of G (also see [28], [13]). Thus, game tree search procedures such as alpha-beta can be viewed as procedures for finding a largest merit solution tree of an AND/OR tree with certain cost functions.

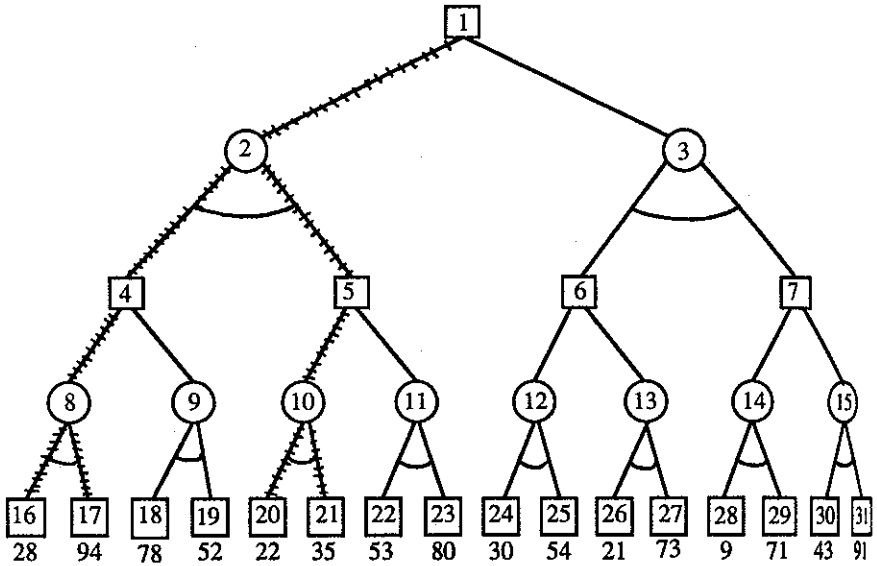


Fig. 2 An AND/OR tree G. AND nodes are represented by square nodes and OR nodes are represented by circle nodes. Hash marks show a solution tree T of G; f(T) = 22.

2.3 Versatility of Monotone Functions

The monotone functions are a wide class of functions. A number of useful cost (or merit) functions are monotone. Examples are given below.

(1) If we define $t_p(x_1,...,x_k) = x_1 + ... + x_k$, and c(n) = 1 for each terminal node n, then f(T) is the total number of terminal nodes in T.

(2) If we define $t_p(x_1,...,x_k) = 1 + \max\{x_1,...,x_k\}$, and c(n) = 0 for each terminal node n, then f(T) is the total amount of time needed to solve root(T) under the assumption that every reduction operation requires one unit of time and reductions may be performed in parallel.

(3) Let $t_p(x_1,...,x_k) = c_p + x_1 + ... + x_k$, where c_p is the cost of applying the reduction operator p. Then f(T) is the sum of the costs of solving terminal problems of T and of applying the problem reduction operators. This cost function is same as the one used by the procedure AO* in [23], [17].

(4) Let $t_p(x_1,...,x_k) = \min\{x_1,...,x_k\}$ in a maximization problem (as discussed in Section 2.2). Then the largest of the merits of solution trees of G is the minimax value of root(G) if G is viewed as a game tree (for a proof see Section 2.2; also see [28], [13]). This cost function is the one used by SSS*, alpha-beta, and B*.

3. A GENERAL BRANCH-AND-BOUND FORMULATION

The class of problems solved by branch-and-bound (B&B) procedures can be abstractly stated as follows:

> For a given arbitrary discrete set X and a real-valued cost function f: $X \rightarrow R$, find an *optimum* element of X, i.e., an $x^* \in X$ such that for all $x \in X$, $f(x^*) \leq f(x)$.[4]

B&B procedures decompose the original set into sets of decreasing size. The decomposition of each generated set S is continued until tests reveal

[4] The discussion in this section is also applicable (with appropriate modifications) to the case where f denotes the merit of the elements of X, and an element of highest merit is desired.

either that S is a singleton (in which case, we measure its cost directly)[5] or that there is an optimum element x* not in S (in which case, the set is "pruned" or eliminated from further consideration). If the decomposition process is continued (and satisfies certain properties), an optimum element will eventually be found. Often, only a small fraction of the total set X need be generated.

The basic elements of our branch-and-bound formulation are described below. This description has been greatly influenced by the earlier formulation due to Mitten [19]. The formulation presented here is very similar to the one given in [13]. As is discussed in [10], [13], the dominance relation in our formulation is used for pruning in a manner somewhat different than in [9], [6], [7].

3.1. Basic Definitions

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

A *branching function* BRANCH is any function which divides the members of the collection A into subsets which collectively include precisely the same elements of X as the original collection A. Mathematically, it is any function mapping collections into collections such that:
(i) $X_i \in \text{BRANCH}(A) \Rightarrow X_i \subseteq X_j$ for some $X_j \in A$.
(ii) $\cup(\text{BRANCH}(A)) = \cup(A)$.

[5]The situation is actually somewhat more complicated: in practice one uses a compact representation of S rather S itself. We have discussed how to handle this issue in [20]. The same concepts apply straightforwardly to the material presented in the current paper, but we do not discuss them here. In Section 4.2, the reader will see an example of the use of representations of sets.

From Property (ii) of the function BRANCH we immediately get the following result:

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

Often the function BRANCH is defined as a composition of selection and splitting functions. A *selection function* is any function SELECT mapping collections into collections such that $\text{SELECT}(A) \subseteq A$. A *splitting function* SPLIT is any function satisfying the properties of a branching function. BRANCH is then defined as

$$\text{BRANCH}(A) = (A\text{-}\{\text{SELECT}(A)\}) \cup \text{SPLIT}(\text{SELECT}(A)).$$

Although this definition of BRANCH is mathematically equivalent to the one given above, it emphasizes the characteristic that only the elements from a certain selected subset of the collection A are divided, and the rest are returned unchanged. In fact, in many implementations of BRANCH, only one selected element from the collection A is divided, and the rest are returned unchanged.

The *dominance relation* D is the binary relation between subsets X_i, X_j of X such that $X_i \, D \, X_j$ if and only if $f^*(X_i) \le f^*(X_j)$. 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 $X_i \, D \, X_j$ and $X_i, X_j \in A$, then $f^*(\cup(A)) = f^*(\cup(A\text{-}\{X_j\}))$.

This 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 changing its optimum value.

The *pruning function* PRUNE: $2^Y \rightarrow 2^Y$ prunes the dominated subsets of A. It is defined as $\text{PRUNE}(A) = A\text{-}A^D$, where A^D is a subset of A such that for all $X_i \in A^D$ there exists some $X_j \in A\text{-}A^D$ such that $X_j \, D \, X_i$.

From Lemma 3.2, it follows that all the members of A^D can be eliminated from the collection A without changing $f^*(A)$. This important result is

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

3.2. An Abstract B&B Procedure

The 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, and $|S|$ denotes the cardinality of a set S.

```
procedure P0 (* B&B procedure to search for an optimum element of a set *)

begin
  A:= {X};     (* initialize the collection A *)

  while |∪(A)| ≠ 1 do   (* loop until A contains only one element of X *)
    A:=BRANCH(A);  (* branch on the collection A  *)

    A:=PRUNE(A)    (* eliminate the dominated subsets from A *)
  end
end
```

From lemmas 3.1 and 3.3, it is easy to show that if the procedure P_0 terminates, A contains only an optimal element of X. Proofs of this are given for slightly different (but quite similar) descriptions of B&B in [13] and [20]. Note that the termination of P_0 is not guaranteed. In order to guarantee the termination of P_0, BRANCH and PRUNE must satisfy certain additional properties.

3.3. The Best-first Selection Strategy

In many problem domains it is possible to associate a lower bound $lb(X_i)$ with the subsets X_i of X such that

(i) For all $x \in X_i$, $lb(X_i) \leq f(x)$.

(ii) For all $x \in X_i$, $lb(\{x\}) = f(x)$.

Thus $lb(X_i)$ is a lower bound on the costs of the elements of X_i and the

lower bounds for singleton sets are not unnecessarily loose. This lower bound information can be fruitfully used in selecting an element for branching. If in every cycle of P_0's loop an element of A is chosen for branching which has the least lower bound of all the elements of A, then the selection rule is called *best-first*, and the branch-and-bound procedure using this strategy is called best-first branch-and-bound. An interesting feature of best-first B&B is that whenever a singleton set $\{x\}$ is selected for branching, the procedure can terminate. This is because $f^*(\{x\}) = f(x) = lb(\{x\}) \leq lb(X_i) \leq f^*(X_i)$ for all $X_i \in A$, and thus $\{x\}$ dominates all the other elements in A.[6]

If the bounds $lb(X_i)$ are good approximations of $f^*(X_i)$, then best-first B&B can be very efficient. In the extreme case, if $lb(X_i) = f^*(X_i)$ for all $X_i \subseteq X$, then the B&B procedure finds an optimal element of X by splitting only those sets which contain optimal elements.

3.4. Discussion

In this abstract formulation, a number of details have been left out. For example, we have only defined 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 for 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 could 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. In practice, we may not know which sets in A dominate which other sets in A without exhaustively enumerating the elements in the sets which are members of A. However, partial knowledge from the problem

[6] If we select more than one element of A for branching, then the selection rule is still called best-first, as long as at least one of the selected elements (let's call it X) has the least lower bound of all the elements of A.

domain is often available to reveal that certain sets in A dominate certain other sets in A. This partial knowledge of the dominance relation can be used to construct a dominated subset A^D, of A. In the next section, where we present a practical B&B procedure to find an optimum solution tree of an AND/OR graph, we show how general knowledge about AND/OR graphs is used to ascertain dominance between two sets of solution trees.

4. BRANCH-AND-BOUND SEARCH ON ACYCLIC AND/OR GRAPHS

Consider the problem of finding a least-cost solution tree of an acyclic AND/OR graph G. In this case, the discrete set X is the set of all solution trees of G. The cost 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. In this section we introduce partial solution trees to represent sets of solution trees, and present a general B&B procedure for searching acyclic AND/OR graphs for optimum solution trees.

4.1. Partial Solution Trees

A *partial solution tree* (or partial tree) T' of an AND/OR graph G is a subgraph of unfold(G) with the following properties:

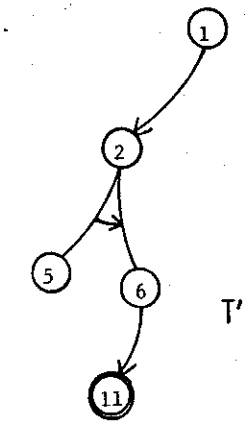
(i) root(T') = root(unfold(G)).

(ii) If any node of unfold(G) other than root(unfold(G)) is in T', then it has an ancestor in T'.

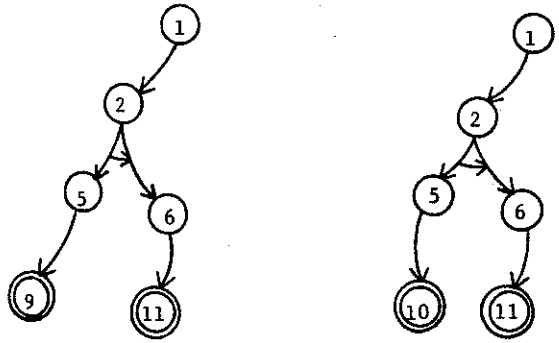
(iii) If an OR node n of unfold(G) is in T', then none of its siblings are in T'.

(iv) If an AND node of unfold(G) is in T', then all of its siblings are in T'.

A partial solution tree can be extended (possibly in several ways) to form a total solution tree. It represents the set of all solution trees which can be formed by extending it. We denote this set by S_TREES(T'). Fig. 3



(a)



(b)

Fig. 3 **(a) A partial tree T' of the AND/OR graph G of Fig. 1.**

(b) The solution trees represented by T'.

solution trees represented by T'.

A node of a partial tree T' is called a *tip* node if it has no children in T'. A tip node of T' is either a terminal node (if it has no children in G), or a nonterminal node (if it has children in G). It follows that a partial tree all of whose tip nodes are terminal nodes in G represents just one solution tree, namely itself.

We define $f^*(T')$ to be the minimum of the costs of all solution trees represented by T'; i.e., $f^*(T') = \min\{ f(T) \mid T \in S\_TREES(T') \}$. The dominance relation D between any two partial trees T_i' and T_j' is defined as follows: T_i' D T_j' if and only if $f^*(T_i') \leq f^*(T_j')$.

4.2. A B&B Procedure for a Least-Cost Solution Tree Search.

We now present a version of P_0 to do a B&B search on an acyclic AND/OR graph. Here, A denotes a collection of partial solution trees (each of which represents a set of solution trees). For any set A of partial trees, $\cup(A)$ denotes the union of the sets of solution trees represented by the partial trees in A; i.e., $\cup(A) = \cup\{S\_TREES(T_i') \mid T_i' \in A\}$.

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:

$$BRANCH(A) = (A\text{-}SELECT(A)) \cup SPLIT(SELECT(A)).$$

SELECT returns some of the partial trees in A; i.e., $SELECT(A) \subseteq A$. SPLIT returns partial trees after extending them; i.e., if $T' \in SPLIT(A)$ then there is a $T_1' \in A$ such that T' is an extension of T_1'.

The function PRUNE takes a collection A of partial trees as its input, identifies a set of partial trees A^D such that each partial tree of A^D is dominated by some partial tree in $A - A^D$, and returns $A\text{-}A^D$; i.e., $PRUNE(A) = A - A^D$. Thus PRUNE eliminates only dominated partial trees.

Let T_0' be the partial tree containing only root(unfold(G)) (note that $S\_TREES(T_0')$ is the set of all solution trees of the AND/OR tree G). The following B&B procedure searches for an optimum solution tree of G. Here A denotes the collection of partial trees upon which branching and pruning operations are performed.


```
procedure P1 (* B&B procedure to search for an optimum
              solution tree of an AND/OR graph 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 & split some set of solution trees from A *)
    A := PRUNE(A);   (* remove some dominated solution trees from A  *)
  end;
end
```

Since P_1 is an instantiation of P_0, it follows that if P_1 terminates, then the collection A will contain only an optimum solution tree of G.

4.3. Discussion

In the above rather informal description of P_1, various details have been left out. We did not specify how the collection of partial trees is maintained, how a partial tree (or a group of partial trees) is selected for branching, or how A^D is identified. Depending on the problem being modeled by the AND/OR graph and the kind of problem-specific information available, there are various ways in which these details can be specified, each leading to a different search procedure. In the next two sections, we discuss these details and show how AO*, B*, SSS*, and alpha-beta can be considered as special cases of these procedures.

5. A BEST-FIRST B&B SEARCH FOR A LEAST-COST SOLUTION TREE

In this section we present an instantiation of P_1 which uses a specific data structure for representing a collection of partial trees, and a best-first selection function for branching.

5.1. Partial Graphs: A Representation for a Collection of Partial Trees

A partial graph G' is a subgraph of G with the following properties:

(i) root(G') = root(G).

(ii) Any node n of G' other than root(G) has an ancestor in G'.

The nodes having no children in G' are called *tip* nodes. A partial tree T' of G' is defined exactly as it was defined for an AND/OR graph in Section 4. A partial graph G' *represents* all partial trees T' of G' such that all tip nodes of T' are also tip nodes of unfold(G'). The set of partial trees represented by G' is denoted by P_TREES(G'). For example, Fig. 4 shows a subgraph G' of the AND/OR graph G of Fig. 1, and the set of partial trees represented by G'.

At the beginning of P_1, G' contains only root(G), and thus represents the partial tree T_0'. The branching operation on G' consists of the following actions:

(i) Select a partial tree T' from P_TREES(G').

(ii) Select a tip node n of T'.

(iii) Let $n_1,...,n_k$ be the children of n in G. Expand n by augmenting G' to include $n_1,...,n_k$ as children of n.

This is equivalent to selecting all the partial trees in P_TREES(G') which contain the node n, and performing splitting operations upon them as follows. If n has $n_1,...,n_k$ as AND-children in G, then each partial tree T' in P_TREES(G') containing n (as a tip node) is in effect replaced by the partial tree T'<n-n_1..n_k>. This is an extension of T' which includes $n_1,...,n_k$ as AND children of n. If n has $n_1,...,n_k$ as OR children in G, then each partial tree T' in P_TREES(G') containing n is in effect replaced by a set of partial trees $\{T'<n\text{-}n_i> \mid 1\le i\le k\}$. A partial tree T'<n-$n_i$> is an extension of T which includes n_i as the OR child of n in T'. Clearly,

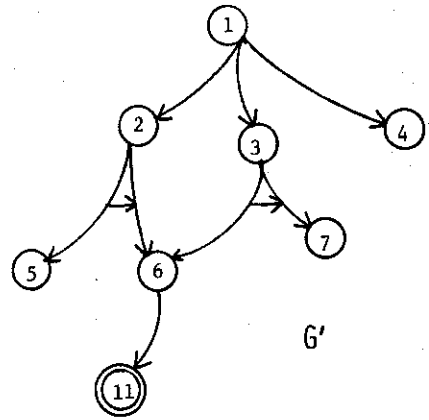
$$S\_TREES(T') = \cup\{S\_TREES(T'<n\text{-}n_i>) \mid 1\le i\le k\},$$

and

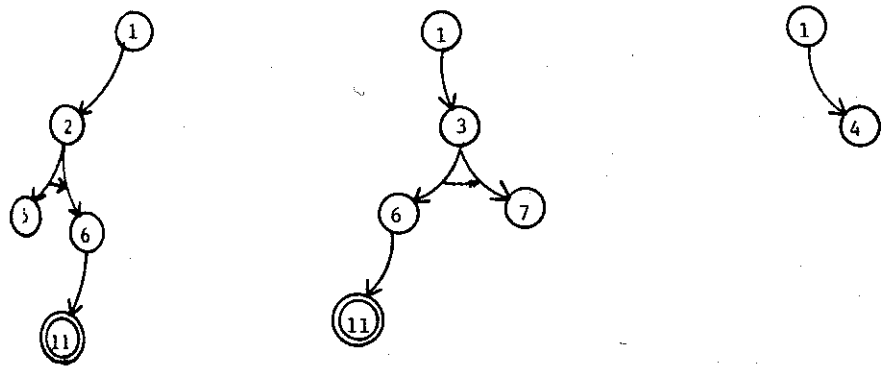
$$S\_TREES(T') = S\_TREES(T'<n\text{-}n_1...n_k>).$$

It follows that the branching operation on G' has both the properties required for it in Section 3.[7]

[7] Note that a successor n_i of n may be in G' even before $n_1,...,n_k$ are included as successors of n in G'. If such n_i has some successors present in G' (because n_i was expanded in a previous branching operation) then effectively some more branching operations have been performed on



(a)



(b)

Fig. 4 (a) A partial graph G' of the AND/OR graph G of Fig. 1.

(b) The partial trees represented by G'.

Pruning on G' is problem dependent, and is performed by deleting certain portions of G' in such a way that no nontip node of G' becomes a tip node as a result of deletion. It follows that P_TREES(PRUNE(G')) $\subseteq$ P_TREES(G'). The pruning operation should be performed in such a way that only dominated partial trees are eliminated.

5.2. Lower Bounds on Partial Trees

Suppose there is a real-valued function b defined over the nodes of G, which has the following properties:

(i) If n is a terminal node of G, then $b(n) = c(n)$.

(ii) If n is a nonterminal node of G, then $b(n) \leq c^*(n)$.

Then b(n) is a lower bound on the value of $c^*(n)$; i.e., the cost of a solution tree rooted at n is always at least b(n). For a partial tree T', we define $f_b(T')$ recursively as follows:

5.1a If T' consists only of a single node m = root(T'), then $f_b(T') = b(m)$.

5.1b If m = root(T') has AND-children $m_1,...,m_k$ such that p: m $m_1,...,m_k$ is a reduction, then $f_b(T') = t_p(f_b(T'_1),...,f_b(T'_k))$, where $T'_1,...,T'_k$ are the subtrees of T' rooted at $m_1,...,m_k$.

5.1c If m = root(T') has m_i as OR child in T', then $f_b(T') = f_b(T'_i)$, where T'_i is the subtree of T' rooted at m_i.

Due to the monotonicity of t_p, it can be proved by induction on the height of root(T') that $f_b(T') \leq f^*(T')$. It follows that f_b satisfies the properties of a lower bound as mentioned in Section 3.

5.3. Selecting a Partial Tree Having a Least Lower Bound

We define $b^*_{G'}(n)$ (G' is omitted if unambiguously determined) to be the smallest value $f_b(T')$ such that T' is a partial tree of G' rooted at n whose tip nodes are also tip nodes of G'. The following theorem provides a way of computing $b^*(n)$ for the nodes n of a partial graph G'.

Theorem 5.1. If the functions $t_p(.,...,.)$ are monotonically nondecreasing in each variable, then the following recursive equations hold.

(i) If n is a tip node, then
$$b^*(n) = b(n)$$

(ii) If p:n $\rightarrow n_1,...,n_k$ is a reduction and $n_1,...,n_k$ are AND children of n in G', then
$$b^*(n) = t_p(b^*(n_1),...,b^*(n_k))$$

(iii) If n has $n_1,...,n_k$ as OR children in G', then
$$b^*(n) = \min\{b^*(n_1),...,b^*(n_k)\}$$

Proof: Similar to the proof of Theorem 3.1.

[]

From the monotonicity of t_p, it follows that $b^*(n)$ is also a lower bound on $c^*(n)$. Note that the functions f_b and b^* are defined for a partial graph G' in nearly the same way as the functions f and c^* have been defined for G in Section 2.

The following procedure selects a partial tree having a least lower bound from P_TREES(G').

Procedure S_0

(1) Calculate $b^*(n)$ for all nodes n of G'. Since G (and therefore G') are acyclic, this can be done using the equations of Theorem 5.1 in a bottom-up manner.

(2) Direct arrows[8] from each nontip node n of G' as follows: if n has $n_1,...,n_k$ as AND children, then direct arrows from n to all the children; if n has $n_1,...,n_k$ as OR children, then direct arrows from n to the child n_i which has the least value $b^*(n_i)$ of all the children.

(3) Choose the partial tree T' of G' by following arrows from the root node to the tip nodes of G'.

[8] The procedure for directing arrows here is similar to the one used in [21] for selecting a par-

It follows from Theorem 5.1 and the monotonicity of t_p that the chosen partial tree T' has the least lower bound.

5.4. A Best-First B&B Procedure

The following procedure finds a least-cost solution tree of an acyclic AND/OR graph G. A partial graph G' represents a collection of partial trees.

Procedure P_2

(1) (initialize): G' := root(G).

(2) (branch): In G', select a partial tree T' of least lower bound by traversing the arrows (if there are any) from root(G'). If the selected partial tree is a total solution tree, then go to step 4, else for some tip node[9] n of T', expand n (i.e., include all children of n in G as the children of n in G').

(3) (rearrange arrows): Calculate $b^*_{G'}(n)$ for the nodes n of G' and set arrows between the nodes of G' using the procedure S_0. (More specifically, set $b^*(n_i) = b(n)$ for each newly generated successor n_i of n, and update $b^*_{G'}(m)$ and arrows for each ancestor m of n). Go to step 2.

(4) (prune): Remove all the nodes from G' except those belonging to T since T' dominates every other partial tree in P_TREES(G'). Stop (T' is a least-cost solution tree).

Since P_2 is an instantiation of P_1, it follows that at the termination of P_2, G' will have only a least-cost solution tree. The termination of P_2 is guaranteed because in each iteration of P_2 one node of G is expanded and G has only a finite number of nodes.

[9] The node n may be selected in whatever way one believes is most likely to increase the

5.5. AO* as a B&B Procedure

Let us define monotone functions $t_p(.,...,.)$ associated with reductions $p:n \rightarrow n_1,...,n_k$ (where $n_1,...,n_k$ are AND children of n) as $t_p(x_1,...,x_k) = c_p + x_1 +...+ x_k$, where c_p is the cost associated with the reduction $p: n \rightarrow n_1,...,n_k$. In this case, P_2 becomes equivalent to the AO* procedure as described by Nilsson [23]. The heuristic estimates h for nodes of G used in [23] are equivalent to the lower bounds b associated with the nodes of G. The values $h^*(n)$ and q(n) defined for a node n in [23] are equivalent, respectively, to $c^*(n)$ and $b^*(n)$ in our treatment.

Note that the monotonicity restriction (also called the consistency property in [17]) on h as presented in p. 103 of [23] has no connection with the monotonicity of the cost functions t_p associated with reductions of G. If h satisfies monotonicity restriction then it merely makes the procedure somewhat simpler (see [23]). The same simplification can be made in P_2 (i.e., in Step 3, we only need to update b^*-value of those ancestors of n from which n can be reached by following arrows) if b satisfies the following *consistency* condition:

$$b(n) \leq t_p(b(n_1),...,b(n_k)).$$

5.6. Variations of P_2

In Section 5.2, we defined a lower-bound function f_b on partial trees, which was used by P_2 to select a most promising partial tree of G'. It follows from the discussion in Section 3.3 that if P_2 uses a lower-bound function which is a better approximation of f^* (as compared to f_b), then we can expect P_2 to be more efficient. In this section, we define two lower-bound functions f^1_b and f^2_b which are at least as good (and some times better) approximations of f^* than f_b.

For a partial tree T', both b(root(T')) and the computed lower bound of T' based upon the lower bounds on the subtrees of T' (as in equations 5.1a and 5.1b) provide lower bounds on $f^*(T')$. $f^1_b(T')$ is the best bound on

$f^*(T')$ based upon the available information, and is defined as follows:

5.2a if T' consists only of a single node m = root(T'), then $f^1_b(T') = b(m)$.

5.2b If m = root(T') has AND-children $m_1,...,m_k$ such that p: m $m_1,...,m_k$ is a reduction, then $f^1_b(T') = \max\{b(m), t_p(f^1_b(T'_1),...,f^1_b(T'_k))\}$, where $T'_1,...,T'_k$ are the subtrees of T rooted at $m_1,...,m_k$.

5.2c If m = root(T') has m_i as OR child in T', then $f^1_b(T') = \max\{b(m), f^1_b(T'_i)\}$, where T'_i is the subtree of T' rooted at m_i.

Due to the monotonicity of t_p, it can be proved by induction on the height of root(T') that $f^1_b(T') \le f^*(T')$. It follows that f^1_b satisfies the properties of a lower bound as mentioned in Section 3. From equations 5.1 and 5.2 it follows that for any partial tree T', $f^1_b(T') \ge f_b(T')$. If b is consistent than it can be proved by induction that f_b and f^1_b are identical.

We redefine $b^*_{G'}(n)$ (G' is omitted if unambiguously determined) to be the smallest value $f^1_b(T')$ such that T' is a partial tree of G' rooted at s whose tip nodes are also tip nodes of G'. Theorem 5.1 is redefined as follows:

Theorem 5.1a. If the functions $t_p(.,...,.)$ are monotonically nondecreasing in each variable, then the following recursive equations hold.

(i) If n is a tip node, then
$b^*(n) = b(n)$.

(ii) If p:n $\rightarrow n_1,...,n_k$ is a reduction and $n_1,...,n_k$ are AND children of n in G', then
$b^*(n) = \max\{b(n), t_p(b^*(n_1),...,b^*(n_k))\}$.

(iii) If n has $n_1,...,n_k$ as OR children in G', then
$b^*(n) = \max\{b(n), \min\{b^*(n_1),...,b^*(n_k)\}\}$.

[]

From the monotonicity of t_p, it follows that $b^*(n)$ is also a lower bound on $c^*(n)$. If P_2 uses f^1_b to select a least-lower-bound partial tree, then in step 3, it only needs to update b^* and arrows for those ancestors of n (the expanded node) from which it is possible to get to n by traversing arrows. (Note that because of Theorem 5.1a, b^*-value of an ancestor of n can only increase via the updating process.) For sumcost functions (i.e., $t_p(x_1,...,x_k) = c_p + x_1 +...+ x_k$), P_2 using f^1_b is identical to the algorithm B in [2]. For inconsistent b (since f^1_b can be a tighter bound than f_b) P_2 using f^1_b is expected to be more efficient than P_2 using f_b.[10]

We define $f^2_b(T')$ for a partial tree T' of G' as follows[11]:

5.3a If T' consists only of a single node m = root(T'), then $f^2_b(T') = b(m)$.

5.3b If m = root(T') has AND-children $m_1,...,m_k$ such that p: m $m_1,...,m_k$ is a reduction, then $f^2_b(T') = t_p(f^2_b(T'_1),...,f^2_b(T'_k))$, where $T'_1,...,T'_k$ are the subtrees of T' rooted at $m_1,...,m_k$.

5.3c If m = root(T') has m_i as OR child in T', then
if it is possible to go from m to the most recently expanded node n of G' by following arrows then $f^2_b(T') = f^2_b(T'_i)$, where T'_i is the subtree of T' rooted at m_i.
Otherwise, $f^2_b(T') = f^2_b(T'')$, where T'' is the previous version of T' (i.e., T' as it looked before n was expanded).

---

[10] Similar improvements to the state space search algorithm A* have been proposed in [15], [16], [1], [3], when the heuristic function is inconsistent.

[11] Note that $f^2_b(T')$ is defined only for the partial trees T' represented by partial graphs G' which are constructed during the execution of P_2.

Due to the monotonicity of t_p, it can be proved by induction on the height of root(T') that $f^2_b(T') \leq f^*(T')$. It follows that f^2_b satisfies the properties of a lower bound as mentioned in Section 3. From equations 5.1, 5.2 and 5.3 it follows that for any partial tree T', $f^1_b(T') \geq f^2_b(T') \geq f_b(T')$. If b is consistent than it can be proved by induction that f_b, f^1_b and f^2_b are identical.

We redefine $b^*_{G'}(n)$ (G' is omitted if unambiguously determined) to be the smallest value $f^2_b(T')$ such that T' is a partial tree of G' rooted at n whose tip nodes are also tip nodes of G'. Theorem 5.1 is redefined as follows:

Theorem 5.1b. If the functions $t_p(.,...,.)$ are monotonically nondecreasing in each variable, then the following recursive equations hold.

(i) If n is a tip node, then
$b^*(n) = b(n)$.

(ii) If $p:n \rightarrow n_1,...,n_k$ is a reduction and $n_1,...,n_k$ are AND children of n in G', then
$b^*(n) = t_p(b^*(n_1),...,b^*(n_k))$.

(iii) If n has $n_1,...,n_k$ as OR children in G', then
if the most recently expanded node of G' can be reached from n by following arrows, then
$b^*(n) = \min\{b^*(n_1),...,b^*(n_k)\}$.
Otherwise, $b^*(n)$ is unchanged.

Proof: Similar to the proof of Theorem 3.1.
[]

From the monotonicity of t_p, it follows that $b^*(n)$ is also a lower bound on $c^*(n)$. If P_2 uses f^2_b to select a least-lower-bound partial tree, then in step 3, it only needs to update b^* and arrows for those ancestors (of the expanded node n) from which it is possible to get to n by traversing arrows. For sumcost functions (i.e., $t_p(x_1,...,x_k) = c_p + x_1 +...+ x_k$), P_2 using f^2_b is identical to the algorithm A in [2]. For inconsistent b (since f^2_b can be a tighter bound than f_b) P_2 using f^2_b is expected to be more efficient than P_2 using f_b, but less efficient than P_2 using f^1_b.

Note that f^2_b is presented here only to explain the working of the algorithm A from [2] in terms of B&B. Viewing A as a B&B procedure simplifies its correctness and makes it easy to see that it works for monotone cost functions.

5.7. Discussion

When AO* is viewed as a B&B procedure, its correctness proof becomes very simple (as compared, for instance, to the one given in [16]). The correctness of AO* and its variations (algorithms A and B given in [2]) directly follows from the correctness of the general B&B formulation for AND/OR graph search. Note that the only requirement for the correctness of our "AO*-type" B&B formulation is that the functions $t_p(.,...,.)$ be monotone. Thus the heuristics developed for acyclic AND/OR graphs with additive cost functions are also applicable to acyclic AND/OR graphs with arbitrary monotone cost functions. Such functions, as discussed in Section 3, can model a much larger class of problems.

The algorithm P_2 first appeared in the 1982 PhD dissertation of the first author. In [24], Pearl independently presented a generalized version of AO* which is very similar to P_2. He assumed that the procedure works for all cost functions (including ones which are not monotone) as long as a lower-bound function b can be found. From the discussion in this section, it is clear that Pearl's assumption was incorrect; i.e., the monotonicity of t_p is crucial to the correctness of P_2. For example, theorems 2.1 and 5.1 do not hold in general if the cost functions are not monotone). Fig. 5 shows an AND/OR graph with (nonmonotone) cost functions and the lower bounds, for which the general version of AO* given in [24] (and P_2) would not find an optimal solution tree. The solution found by these algorithms has cost 10, whereas the optimal solution tree has cost 5.

Cost functions associated with the reductions of G:

$t_{p1}(x_1,x_2) = x_1 - x_2$;

Terminal cost function c:

c(6) = 20; c(7) = 15;
c(3) = 10.

Lower bound function b:

b(1) = 0; b(2) = 3;
b(3) = 10; b(4) = 15;
b(5) = 0; b(6) = 20;
b(7) = 15.

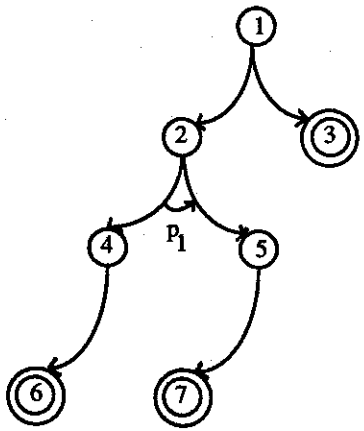


Fig. 5 An AND/OR Graph with associated cost functions and lower bounds.

6. B&B SEARCH FOR A SOLUTION TREE OF LARGEST MERIT

In the discussion of AND/OR graph search in Sections 4 and 5, we considered c and f as the cost functions for terminal nodes and solution trees of G, and we constructed a B&B procedure for finding a least-cost solution tree. The procedure can easily be modified to find a solution tree of largest merit (see Section 2.2) if the functions c and f denote the merits of terminal nodes and solution trees of G respectively. In the modified version, $f^*(T') = \max\{f(T) \mid T \in S\_TREES(T')\}$, and $T_i' \; D \; T_j'$ if and only if $f^*(T_i') \geq f^*(T_j')$. To perform a best-first B&B search, we define an upper bound function u (similar to the lower bound function b) on the nodes of G such that u(n) is an upper bound on the value of $c^*(n)$. The functions f_u and u^* are defined similarly to the functions f_b and b^*, respectively. It is easily seen that the procedure P_2 can be appropriately modified to perform a best-first search for a largest-merit solution tree of an acyclic AND/OR graph.

6.1. B* as a B&B Procedure

If the functions t_p are defined as $t_p(x_1,...,x_k) = \min\{x_1,...,x_k\}$, and c and f are taken as the merits of the terminal nodes and solution trees of G, then the B&B procedure for AND/OR tree search with best-first branching strategy (i.e., P_2) is nearly identical to the B* [5] algorithm for minimax search of game trees. The only difference is that with AND/OR trees (or graphs), the aim is to find a solution tree of largest merit (or least cost, depending on the problem), whereas with game trees the objective is merely to find the immediate successor of the root in a largest-merit solution tree. This difference in requirements has been the motivation for introducing the "prove-best" and the "disprove-rest" strategies in the B* algorithm.

Let us assume that both the upper bound function u and lower bound function b are available for the nodes of a game tree G (viewed as an AND/OR tree). Let the B&B procedure for searching G with some strategy (not necessarily best-first) for partial tree selection also keep track of both $b^*(n)$ and $u^*(n)$ for each node n of the partial graph G'. Let $n_1,...,n_k$ be the OR successors of root(G). If it happens during the search that for some $1 \leq j \leq k$,

(6.1) $b^*(n_j) \geq \max\{ u^*(i) \mid 1 \leq i \leq k \text{ and } i \neq j\}$,

then the largest-merit solution tree rooted at n_j is no worse than the largest-merit solution tree rooted at n_i ($1 \le i \le k$ and $i \ne j$). A B&B procedure searching a game tree can terminate at this point since the identity of the OR-successors of root(G) in a largest-merit solution tree is known.

For nodes n of a partial graph G', $b^*(n)$ and $u^*(n)$ are lower and upper bounds on $c^*(n)$, respectively. As G' grows by the expansion of nodes in the subgraph rooted at n, $b^*(n)$ and $u^*(n)$ will possibly more closely approximate $c^*(n)$. Since, in the B&B search of game trees, the aim is to satisfy Equation 6.1 as soon as possible, one of the following two policies can be adopted:

(i) Select a node n_j (from the successors $n_1,\ldots,n_k$ of root(G)) such that

$$u^*(n_j) = \max\{ u^*(n_i) \mid 1 \le i \le k \}.$$

Explore the subgraph of G' rooted at n_j with the expectation that $b^*(n_j)$ will ultimately be raised beyond $\max\{ u^*(n_i) \mid 1 \le i \le k \text{ and } i \ne j \}$. This is called a *prove-best* strategy by Berliner in his description of B* [5].

(ii) Select a node n_i such that $n_i \ne n_j$ and $u^*(n_i) > b^*(n_j)$. (It must be possible to find such a node if Equation 6.1 is not satisfied yet.) Explore the subgraph of G' rooted at n_i with the expectation that $u^*(n_i)$ will ultimately come below $b^*(n_i)$. This is called a *disprove-rest* strategy by Berliner.

Once a node n (a child of root(G)) is selected by either of the policies, a partial tree of largest upper bound containing the node n is selected for branching using a procedure similar to S_0. In fact, if B* uses only the prove-best strategy, then it is virtually the same as the procedure P_0 modified to search a game tree for a largest solution tree.

The point to be made is that B* search can be viewed essentially as a B&B search of an AND/OR tree using a somewhat "nonstandard" rule for termination. Furthermore, the special strategies (prove-best & disprove-rest) of B* are applicable as long as the merit functions are monotone (i.e., not necessarily minimax), lower and upper bounds for the nodes of G are available, and the aim is to find just the successor of the root of the best solution tree. For example, a B* type procedure could be used for game tree search even if the evaluation function is the one used in [25] instead of minimax.

6.2. SSS* as B&B

Even if the upper bounds u(n) on $c^*(n)$ for the nodes n of G are not available, we can associate "uninformed" upper bounds with the nodes n of G as follows:

$u(n) = c(n)$ if n is a terminal node;
$u(n) = +\infty$ if n is a nonterminal node.
(the absolute upper bound)

A best-first B&B search procedure (similar to P_2) can find a largest-merit solution tree of G by using these bounds for selection. In any iteration of this procedure there may exist more than one partial tree with the same upper bound. In such cases, various tie-breaking rules can be adopted, each leading to a different variation of the best-first B&B procedure.

A careful observation of the SSS* algorithm reveals that it is a best-first B&B procedure (with uninformed bounds and particular tie breaking rule for selecting from partial trees of the same largest upper bound) searching for a largest-merit solution tree of an AND/OR tree. A detailed treatment of SSS* as a B&B procedure can be found in Kumar and Kanal [13]. We discuss the basic ideas here. Note that in SSS*, as presented in [28], the search graph G' is not maintained explicitly (as it is done in AO* and B*); but it exists conceptually as a subgraph of G, consisting of nodes which have been visited but have not been pruned. Instead of G', in [28] a list of states of traversal called OPEN is maintained. Each element of the OPEN list represents a partial tree T' and the current upper bound f(T') associated with it. State selection and expansion in SSS* directly correspond to selecting a partial tree of largest upper bound from G' and expanding one of its tip nodes. Purging of states from OPEN corresponds to pruning certain parts of G' to eliminate dominated partial trees. Due to a special property of minimax functions, the process of selecting a partial tree of largest upper bound in SSS* is considerably more simple than the procedure S_0 of Section 5.

6.3. Alpha-Beta as a B&B Procedure

In the procedures discussed so far, upper or lower bounds on the f^* values of the partial trees in the collection A were used for selecting a partial tree (or a group of partial trees) for branching. It is also possible to select partial trees regardless of their associated bounds. A possible scheme is to expand the nodes of G in the preorder sequence regardless of which partial tree they belong to. Since the selection of a node for expansion (hence of a set of partial trees for branching) is done in a predefined manner, all the overhead associated with selecting partial trees for branching in best-first B&B is eliminated.

Due to the monotonicity of t_p, partial trees can be pruned (even in the absence of any other problem-specific information) in the following situation. Suppose T_1 and T_2 are two solution trees in G' rooted at a node n such that $f(T_2) \leq f(T_1)$. It follows from the monotonicity of the merit functions t_p that any partial tree T_i' of G' having T_2 as a subtree is dominated by another partial tree T_j' (of G') which is identical to T_i' except that T_2 is replaced by T_1. Hence we can prune all those partial trees from G' which contain T_2 as a subtree by pruning all the nodes from G' belonging only to T_2.

A good example of such types of procedures is the classical alpha-beta algorithm. The alpha-beta procedure as usually presented in the literature (e.g., in [8], [23]) looks at first glance to be very different than conventional B&B procedures. Hence, to make the discussion easier, we first present Nilsson's version [23] of alpha-beta in a somewhat different terminology, and then we show that it can actually be considered as a B&B procedure. An informal description of alpha-beta as B&B can be found in [13].

Nilsson's Version of Alpha-beta

Nilsson's version of alpha-beta can be described as follows:

(1) (initialize) G' := root(G).

(2) Choose the first tip node n of G' in the postorder sequence such that n has not been evaluated yet. If there is no such node, then terminate.

(3) Evaluate n by computing alpha(n) and beta(n) as specified below, and for each ancestor m of n recompute alpha(m) and beta(m). Also generate successors of n in G' if n is a nonterminal.

(4) If there is ever an ancestor m of the node n such that m ≠ parent(n) and (beta(m) ≤ alpha(n) or beta(n) ≤ alpha(m)) then prune all the unevaluated successors of parent(n) from G'.

alpha(m) and beta(m) are defined as follows[12]:

alpha(m) = c(m) if m is terminal and m has been evaluated
= -∞ if m is a nonterminal node whose children have not been generated, or if m is an unevaluated terminal node
= max {alpha(n)| n is a child of m} if m is a max node whose children have been generated
= min {alpha(n)| n is a child of m} if m is a min node whose children have been generated

beta(m) = c(m) if m is terminal and m has been evaluated
= ∞ if m is a nonterminal node whose children have not been generated, or if m is an unevaluated terminal node
= max {alpha(n)| n is a child of m} if m is a max node whose children have been generated
= min {alpha(n)| n is a child of m} if m is a min node whose children have been generated

Some Definitions and properties

Let trees(n) be the set of all solution trees containing n. If V is a solution tree and n is a node of V, then let V/n be the subtree of V rooted at n. The following properties hold:

(6.1) If T is a solution tree, then f(T) = min {c(n)| n is a terminal node of T}. This is because the cost function associated with each AND branch takes the minimum of the values of the child nodes (since it corresponds to a move by Min).

(6.2) Let T' be any subtree of T. Then from (1), $f(T) \leq f(T')$. As a special case, if n is a node of T, then $f(T') \leq f(T'/n) \leq c^*(n)$.

---

[12]Note that Nilsson's definition of alpha and beta is rather different from Knuth's definition [8].

(6.3) As defined above, alpha(n) and beta(n) have ready interpretations as instances of $b^*(n)$ and $u^*(n)$. Hence, alpha(n) $\leq c^*(n) \leq$ beta(n).

(6.4) Suppose alpha(n) > $-\infty$. Then from the definition of alpha, there must be at least one solution tree rooted at n which has already been explored, for else we would have alpha(n) = $-\infty$. From the definition of alpha, it follows that alpha(n) = max{f(V) | V is a solution tree rooted at n and has been completely explored}.

(6.5) Let T and T' be solution trees such that T-(T/m) = T'-(T'/m) and f(T') $\leq$ f(T/m). Then from (1), f(T') $\leq$ f(T).

Pruning by Beta Cutoff

Suppose m is an ancestor of n, and suppose beta(n) $\leq$ alpha(m). Then alpha(m) > $-\infty$, so from (6.4) we know that there is at least one tree rooted at m which has already been completely explored. Let V be the best such tree. Let T' be any member of trees(n), and let T'' = (T'-T/m) $\cup$ V. Then

f(T') $\leq$ f(T'/n)	(from (6.2))
$\leq c^*(n)$	(from (6.2))
$\leq$ beta(n)	(from (6.3))
$\leq$ alpha(m)	(from the assumption)
= f(V)	(from (6.4))
= f(T''/m)	(by definition of T'').

Thus from (6.5), f(T') $\leq$ f(T''). Since this reasoning holds for every T' in trees(n), trees(n) is a dominated set and thus may be pruned.

Pruning by Alpha Cutoff

Suppose m is an ancestor of n, and suppose beta(m) $\leq$ alpha(n). Then alpha(n) > $-\infty$, so from (6.4) we know that there is at least one tree rooted at n which has already been completely explored. Let V be the best such tree. Let T' be any solution tree containing an unexplored child of n, and let T'' = (T' - T'/n) $\cup$ V. Then T' contains m, so

f(T') $\leq c^*(m)$	(from (6.2))
$\leq$ beta(m)	(from (6.3))
$\leq$ alpha(n)	(from the assumption)
= f(V)	(from (6.4))
= f(T''/n)	(by definition of T'').

Thus from (6.5), f(T') $\leq$ f(T''). Since this reasoning holds for every T' containing an unexplored child of n, every such child represents a dominated set which may be pruned.

It is clear that the process of generating successors of a tip node of G' in alpha-beta corresponds to a valid branching function. As discussed above, the pruning that is done by alpha-beta can be explained as the pruning of dominated sets of solution trees. If we supplement the pruning by alpha and beta cutoffs with the pruning criterion suggested at the beginning of Section 6.3, then at the termination of alpha-beta G' will contain only a largest-merit solution tree of G.

7. CONCLUDING REMARKS

This paper has presented a general B&B procedure for searching AND/OR graphs, and described how several procedures for searching AND/OR graphs and game trees may be viewed as instances of the general B&B formulation. This study reveals that a number of seemingly disparate search procedures are in fact quite similar. The essential relationships among the procedures are summarized below.

As shown in the preceding pages, AO* and B* are very similar procedures. Both do best-first searches of AND/OR graphs. The only significant[13] differences are:

(1) AO* searches graphs having additive cost functions and B* searches graphs having minimizing cost functions. But these are both straightforward special cases of the monotone cost functions used for B&B.

---

[13] Another difference between AO* and game tree search procedures is that AO* searches for a solution of lowest cost whereas the other procedures search for solutions of highest merit.

(2) Since the goal of B* is merely to find the immediate successor of the root in a largest-merit solution tree (rather than the entire solution tree), B* has a "nonstandard" termination criterion and the "prove-best" and the "disprove-rest" strategies for selecting nodes for expansion.

Furthermore, as pointed out in Sections 5 and 6, both of these procedures (as well as variations of AO* given in [2]) will work as long as the cost functions are monotone. For example, B* could be used for game tree search even if the cost function is the one used in [25].

AO* and SSS* are both best-first procedures. The only significant difference between them is that AO* assumes more problem-specific knowledge in terms of informed bounds. This knowledge is encoded into the heuristic function h (in our formulation, this is same as the lower bound function b) which the user supplies to AO*. If h(n) is set identically equal to 0 for all n, then the operations of AO* and SSS* are almost exactly the same, except for the different tie-breaking rules used when there are more than one best-bound partial trees available.

SSS* and alpha-beta both use the same amount of problem-specific knowledge, but they use different node selection strategies for branching. SSS* uses a best-first selection strategy, but alpha-beta uses a preorder expansion of the game tree nodes (which amounts to a kind of depth-first strategy). Thus the B&B formulation of alpha-beta differs from the B&B formulation of SSS* primarily in the SELECT function. This shows that these two seemingly very different algorithms are in fact very close cousins. Considering that alpha-beta has been known for over two decades, it is noteworthy that SSS* was discovered only recently in the context not of game playing but of a waveform parsing system [27], [29].

If AO* and its variations given in [2] are viewed as B&B, their correctness proofs become simpler, and the criteria governing their correctness become clearer (see Section 5). For example, in [24] it was incorrectly thought that AO* would work for general cost functions (i.e., including non-monotone cost functions). Our development of the general B&B formulation for AND/OR graph search makes it clear that the monotonicity of the cost function is crucial to the correctness of the general procedure. Variations of AO* given in [2] can be viewed as B&B procedures using lower bound functions which are slightly different than the one used by AO*. This makes it easier to see that they are correct and that they work for monotone cost functions.

The development of the general top-down procedure presented in this paper was inspired by a unified approach to search procedure developed in [10], [14], where it was shown that a large number of procedures for searching AND/OR graphs and state space graphs can be viewed either as top-down or bottom-up. In addition to the general top-down procedure presented here, we have developed a general bottom-up procedure for searching AND/OR graphs [11] which subsumes most of the bottom-up procedures for searching AND/OR graphs and the dynamic programming procedures for solving discrete deterministic optimization problems.

APPENDIX: DEFINITIONS

$c(n)$ = cost of (solving the problem denoted by) node n.

$f(T)$ = cost of solution tree T.

$c^*(n) = \min\{f(T) \mid T \text{ is rooted at } n\}$.

$b(n)$ = lower bound on $c^*(n)$.

$f_b(T')$ = lower bound on the costs of solution trees represented by the partial tree T'.

$b^*_{G'}(n) = \min\{f(T') \mid T' \text{ is rooted at } n \text{ and represented by the partial graph } G'\}$. G' is omitted if unambiguously determined from the context.

S_TREES(T') = the set of solution trees represented by the partial tree T'.

P_TREES(G') = the set of partial trees represented by the partial graph G'.

trees(n) = the set of solution trees containing the node n.

REFERENCES

[1] A. Bagchi and A. Mahanti, Search Algorithms Under Different Kinds of Heuristics - A Comparative Study, *JACM*, pp. 1-21, January 1983.

[2] A. Bagchi and A. Mahanti, Admissible Heuristic Search in AND/OR Graphs, *Theoretical Computer Science 24*, pp. 207-219, 1983.

[3] A. Bagchi and A. Mahanti, Three Approaches to Heuristic Search in Networks, *JACM*, pp. 1-27, January 1985.

[4] A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence. Vol I*, William Kaufman, Inc., Los Altos, CA, 1981.

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

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

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

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

[9] 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.

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

[11] V. Kumar, A General Bottom-up Procedure for Searching And/Or Graphs., *Proc. National Conference on Artificial Intelligence (AAAI-84)*, Austin, Texas, pp. 182-187, 1984.

[12] V. Kumar, Branch-and-Bound Search, pp. 1000-1004 in *Encyclopedia of Artificial Intelligence*, ed. S.C. Shapiro, Wiley-Interscience, 1987.

[13] 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.

[14] V. Kumar and L. N. Kanal, The CDP: A Unifying Formulation for Heuristic Search, Dynamic Programming and Branch & Bound, in *Search in Artificial Intelligence*, ed. Kanal and Kumar, Springer-Verlag, 1988.

[15] A. Martelli, On the Complexity of Admissible Search Algorithms, *Artificial Intelligence 8*, pp. 1-13, 1977.

[16] A. Martelli and U. Montanari, Additive AND/OR Graphs, *Proc. Third Internat. Joint Conf. on Artif. Intell.*, pp. 1-11, 1973.

[17] A. Martelli and U. Montanari, Optimizing Decision Trees Through Heuristically Guided Search, *Comm. ACM 21*, pp. 1025-1039, 1978.

[18] L. Mero, Some Remarks on Heuristic Search Algorithms, *IJCAI-81*, Vancouver, Canada, pp. 572-574, 1981.

[19] 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.

[20] D. S. Nau, V. Kumar, and L. N. Kanal, General Branch-and-Bound and its Relation to A^* and AO*, *Artificial Intelligence 23*, pp. 29-58, 1984.

[21] N. Nilsson, *Searching Problem Solving and Game Playing Trees for Minimum Cost Solutions,* in A.J.H. Morrel(ed.), Information processing-68, 1968.

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

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

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

[25] A. L. Reibman and B. W. Ballard, Non-Minimax Search Strategies for Use Against Fallible Opponents, *Proc. of AAAI,* pp. 338-342, 1983.

[26] 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.

[27] 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.

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

[29] 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.