BRANCH-AND-BOUND AND BEYOND

Patrick A. V. Hall
Department of Mathematics,
(Division of Computing Science),
City University,
London, EC1.

Abstract

The branch-and-bound algorithm is stated in generality, and illustrated by two applications, unidirectional graph search, and reducing a sparse matrix to its minimal band form. The algorithm is then generalised to multiple partitions, applied to bidirectional graph searching for both heuristic and non-heuristic searches, and further extended to graph searches and problem solving with subgoals.

1.0 Introduction and Summary

The branch-and-bound algorithm is a simple technique for the optimisation search for the minimum (or maximum) of a function. It is very basic to artificial intelligence and operations research, and has been repeatedly rediscovered and used in the solution of hard problems. (1,2,6,9) In this paper I shall give a general statement of the algorithm, and then consider the application to reducing a sparse matrix to the band matrix of minimal band-width. After this I shall show that the well known graph searching methods (4,5,9) are somewhat disguised examples of branch-and-bound.

This simple branch-and-bound idea will then bp extended to a more complex optimisation search with multiple starting points, in which form it will be used to solve the bidirectional graph-search problem, giving a new unified view of existing solutions, both heuristic and non-heuristic. Finally the application of multiple branch-and-bound is further extended to cover the use of subgoals in graph search and problem solving.

Graph searching as a problem area in artificial intelligence is important because many problems can be represented as a graph which requires a path to be found between two given nodes (3,5,9). Any path between the start and termination nodes would constitute a solution, but a path of minimal length *(*= number of steps) can be thought of as the most 'elegant' solution, and is often aimed for. Hence icneral Problem Solving becomes finding the shortest path in a graph, which in turn is, as we shall see, a special application of the branch-and-bound al Torithm.

However, not. all problems are most naturally formulated as graph searches; for example the travelling salesman problem is expressible as a shortest path problem, but occurs more naturally as a branch-and-bound application (6). Another example is a matching problem in chromosome analysis considered by Montonari (7); the natural way to tackle this problem is directly using branch-and-bound, but the author laboriously converts the problem to a graph search and then applies the algorithm of Hart, Nilson and Raphael (5).

2.0 The Branch-and-Bound Algorithm

Consider the problem of minimising a function f in some (discrete) set X: i.e. $f:X \rightarrow R$ and we wish to find x^* in X such that $f(x^*) \leq f(x)$ for all x in X. Suppose that we have available a function $g:2^X \rightarrow R$ which computes a lower bound for f in a set A:

$$g(A) \leq f(x) \text{ for all } x \in A \text{ and } A \subset X$$

We shall also require that when A is a single point a, then

$$g(A) = g(\{a\}) = f(a)$$

though this can easily be waived since the algorithm assumes the ability to test whether a set comprises a single point.

Informally, described elsewhere (2,6), the strategy is to divide up (partition) the set X into successively smaller portions, each time only further subdividing the most promising subset of X, as measured by g, until eventually a subset containing a single point is selected for subdivision, which terminates the algorithm with the single point as the minimum.

Definition: a partition π of a set X is a set of subsets of X which are mutually exclusive and exhaustive:

$$\pi = \{A_1, \ldots, A_m : A_i \subseteq X, A_i \cap A_j = \emptyset \text{ if } i \neq j, \text{ and } \bigcup_{i=1}^{m} A_i = X\} \quad \square$$

Formally, the algorithm is:

START

STEP 1. Set partition $\pi = \{X\}$.

ITERATION

STEP 2. Find the Y_m in π such that $g(Y_m)$ is minimal. If more than one, choose any of them.

STEP 3. If Y_m is a set containing a single point, go to TERMINATION, STEP 5.

STEP 4. Partition Y_m into $Y_{m1}, \ldots, Y_{mn}$, and remove Y_m from π, replacing it by $Y_{m1}, Y_{m2}, \ldots, Y_{mn}$. Return to Step 2.

TERMINATION

STEP 5. The minimum x^* is the single point contained in Y_m, with $f(x^*) = g(Y_m)$. $\square$

Clearly the above algorithm does locate the global minimum for any lower bounding function g (providing only that g and f coincide for sets of single points) and any partitioning policy. When g computes the greatest lower bound g^*, the algorithm is most efficient and makes no unnecessary partitions; while if g were constant for all sets of more than one element, the search would have to ultimately evaluate f at every point, making an exhaustive search. These are the two extreme cases: it is very readily seen that for intermediate cases, if $g_1 \leq g_2 \leq g^*$ then using g_1 must result in at least as many partitioning iterations as using g_2.

Two common properties of lower bounding functions are worth remarking on: they will appear later in graph searching. These are

(i) $g(A) \geq g(B)$ if $A \subseteq B$

(ii) $g(A \cap B) \geq \max \{g(A), g(B)\}$.

The actual lower bounding function g depends upon the particular problem and In that sense is a 'heuristic funcztion'. The more information about the problem that is used, the closer g approaches the greatest lower bound g^*, the more efficient is the algorithm as outlined above. One could even contemplate using any estimate of the greatest lower bound, recognising that if the bounding condition is violated, a non-optimal solution may be returned, but the error could be no more than the error in the estimator.

In the form given, the algorithm may only find a candidate solution point after considerable searching and branching: in practice this is undesirable since it may be necessary to terminate search early with a suboptimal solution, because of exceeding either storage or time limits. This requires a modification to the basic algorithm to force the search to refine partitions until a possible solution point has been found - and then to continue searching until the minimal point is found; one way to do this would be to subtract a depth factor from the bounds to bias the search towards depth. An additional advantage to locating a potential solution point early in the search is that pruning of the search tree is possible, while we acquire the disadvantage of (possibly) more iterations. Note however that we are still guaranteed finding the optimum providing that early termination is not forced upon us.

2.1 Application to Band Matrix Reduction

A sparse matrix is one in which the

majority of the elements are zero: these arise in many problems such as engineering structural analysis.(10) Storing sparse matrices in the usual way is wasteful; one solution is to store only the non-zero elements, for example by using hashing methods.(6) This leads to special numerical techniques.(10) A preferred solution would be to reduce the matrix by row and column permutations to a band matrix (i.e. a matrix in which all non-zero elements lie in some band close to the diagonal), for which only the band elements need be stored: the advantage is then that standard and efficient numerical techniques are applicable. The reduction to a band matrix is normally carried out by ad hoc methods, by skilled humans prior to computer runs: what is desired is a guaranteeable method for performing this reduction. The branch-and-bound algorithm proved ideal.

For an n by n matrix, there are $(n!)^2$ possible configurations, and exhaustive searches are untenable. The technique used was to partition by row or column selection. Assuming for the moment that the lower bound function g is available (this will be outlined later), then the first partition of the totality of possible configurations is to consider the n possible selections of particular columns of the original matrix for the first position in the reduced matrix. Among these n subsets, the one with smallest lower bound is selected, and the n possible selections of rows for the first position of the reduced matrix considered. Among the resulting 2n-l subsets that with minimal lower bound is selected for row or column partition as appropriate, and so on, for a minimum of 2n partitions.

The lower bound function computes a best possible bandwidth for a partially determined matrix (with the first m columns fixed and the first m or m-1 rows fixed) as the maximum of the following-
(i) for the fixed portion at top left; count from the diagonal of the fixed portion along the rows and columns to locate the last non-zero.element; within the fixed portion (m by m or m by m-1) the elements are fixed, while beyond the fixed portion it is assumed that the non-zero elements could be positioned immediately following the fixed columns or rows. Repeat this for each row and column within the fixed portion, taking the maximum count found.
(ii) for the unfixed portion at bottom right; count the number of elements in each row and column and assume that these could be placed symmetrically about the diagonal - hence bound to bandwidth here is maximum of [(count + 1)/2] where [I] means the integer part of I.

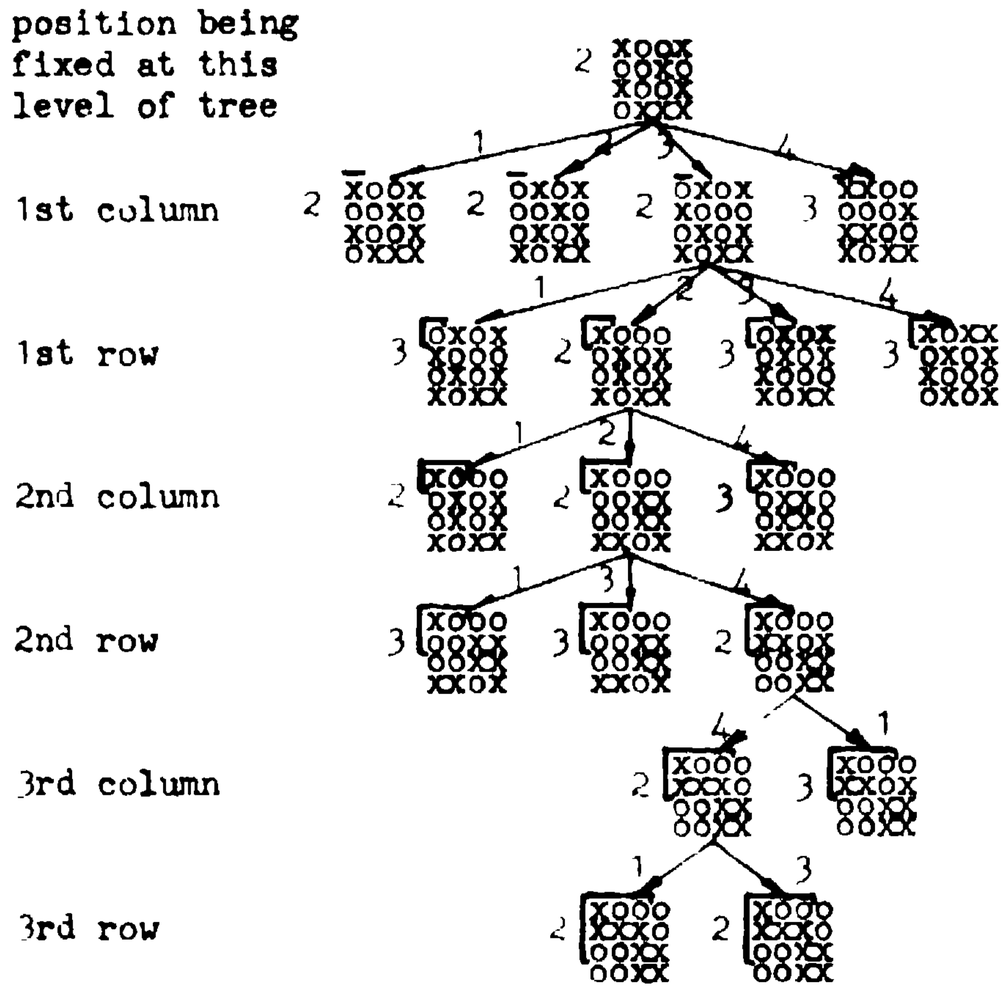
Figure 1 illustrates the method for a 4 by 4 matrix of bandwidth 2. Figure 1(a) shows the original matrix and Figure 1(b) illustrates in tree form the complete search. Each node is labelled with the value of the lower bound function g, and the matrix to data; the elements whose positions are fixed appear in the upper left-hand submatrix marked with thick lines. The search for the next node to develop selects, when more than one node has the minimal value, that node nearest the bottom of the tree. Each level of the tree is labelled to show what development is being made, which row or column is being fixed: each branch of the tree is labelled to indicate which row or column in the original matrix is the one that is being fixed in the position indicate for the whole level.

FIGURE 1. The application of Branch-and-bound to reducing a sparse matrix to its minimal band matrix form.
(a) the original matrix
(b) the search tree: branches are labelled with the row or column of the original being fixed in the position shown at the left; nodes are labelled with the bound, and the fixed portion in the upper left marked off.

1a)

X 0 0 X
0 0 X 0
X 0 0 X
0 X X X

1b)



The algorithm as actually implemented (in Algol) uses a depth forcing initial search (as discussed previously) so that suboptimal results could be returned on exceeding storage bounds. Search length has been found to be very dependent upon the particular matrix, varying from searches direct to goal (as in the example) to searches where only suboptimal (but acceptable) solutions could be found.

3.0 Graph Searching as Branch-and-bound

We shall now examine graph searching and apply the preceding algorithm to its solution. We wish to find the shortest route between two nodes s and t in a graph, such as that illustrated in Figure 1.

We shall define a weighted directed graph as an ordered set (N,F,w) where N is the set of nodes of the graph; E is a set of ordered pairs of nodes which we call the edges of the graph, saying that an edge (a,b) goes from node a to node b; and w is the weight or length function, mapping E to the non-negative real numbers, saying that w(a,b) is the length of edge (a,b).

We define a path or route between two nodes s and t as a sequence of nodes

$$\mu(s,t) = \langle s=n_0, n_1, n_2, \ldots, n_{k-1}, n_k=t \rangle$$

and define the length of this path as

$$w(\mu(s,t)) = \sum_{j=1}^{k} w(n_{j-1},n_j).$$

The set X to be searched is the set of all possible loop-free routes from s to t, and the subsets of routes which will be used in partitions of X will be those which start at s and have an initial part in common. The function g will give the length of the path in common - all routes in the set must obviously be at least that long. Partitioning will be by extending the initial path by one edge for all possible edges eliminating those which introduce loops: each of these extended initial paths defines a subset of the refined partition. A subset of the partition comprises a single point when the defining initial path extends from s all the way to t.

With these definitions, the graph search becomes a branch-and-bound problem, and the search of a simple graph is illustrated in Figure 2. In part (a) the graph is shown, a directed graph with edge weights as labelled. The search process is shown in part (b)t subsets of the partition are named by the path that is common (thus <sac> is the set of all paths starting at s going through a and then c and finally ending at t). The search starts with the partition consisting of the single set <s> of all possible routes from s to t. This is then partitioned into two subsets, <sa> all those routes from s through a to t and <sb> all those routes from s through b to t. These have bounds 1 and *2* respectively. The minimum of these is selected for further refinement, and thus <sa> is refined to <sac> and <sad> with bounds 6 and 7 respectively. And so on until stage 8 where the search for partition-subset of minimum bound g finds <sbct> which contains a single point which is therefore the minimum. Part (c) of Figure 2 shows the search process displayed as a tree with nodes labelled inside by subset and outside by bound. The sequence in which the tree nodes are developed (i.e. subsets partitioned) is in ascending order of bound, with development from left to right where nodes have equal bound.

This search procedure is almost the Dijkstra (4) graph search method, excepting that some redundant routes are eliminated in the Dijkstra method - for example, having two sets of routes with initial parts ending in c (i.e. <sac> and <sbc>) is wasteful and only that of small bound <sbc> need be retained after stage 4: effectively one is coarsening the partition by forming the union of the two sets, and computing the union bound from g(A U B) = min(g(A),g(B)). Note that this pruning is also retroactive in the sense that some unions would occur between a new subset and one that was previously partitioned (for example between <sacd> and <shcd>), but in this case cannot change the earlier bound (of <sbca>) since g has the monotonic property g(A) > g(B) if A C B.

The branch-and-bound algorithm can further be extended to heuristic graph search in a way comparable to the extension of the Dijkstra Algorithm.(6,9) Suppose that we possess a heuristic function h(x,y) which estimates the distance through the graph from node x to node y. If h(x,y) < d(x,y) where d(x,y) is the 'true' minimal distance through the graph from x to y, then using the bounding function

$$\bullet g(\langle s \ldots x \rangle) = w(\langle s \ldots x \rangle) + h(x,y)$$

(formed from the length of the part in common plus the estimate of the distance left to go) gives a lower bound for the set of all paths initially along <s...x> and thence to t.

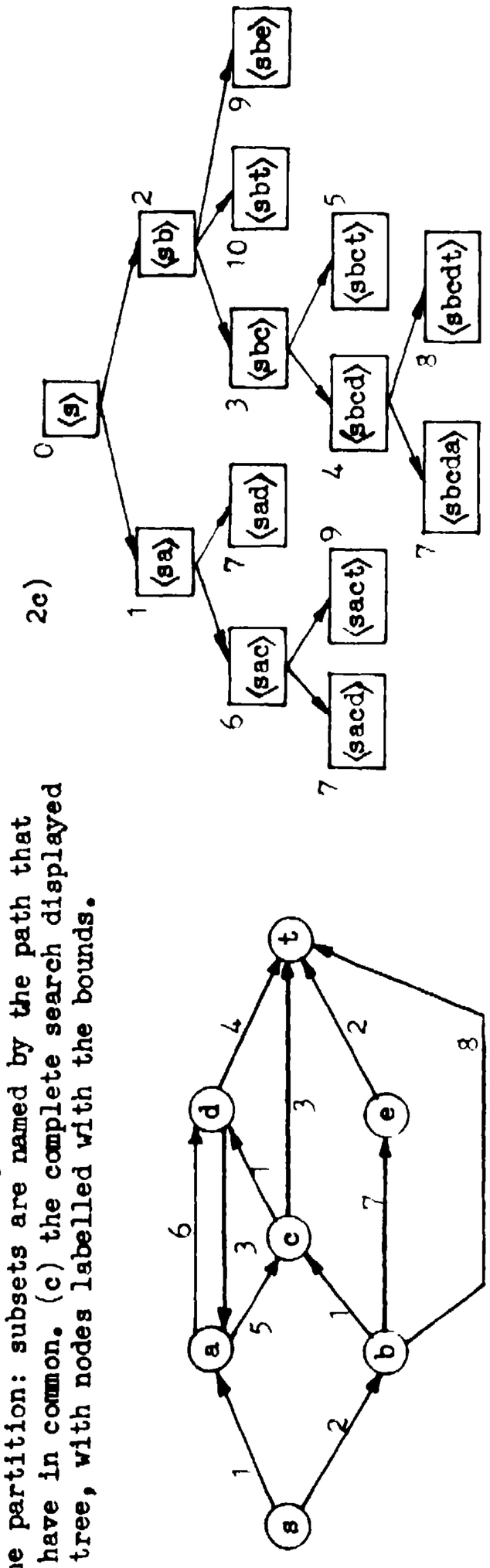


FIGURE 2. The application of the branch-and-bound algorithm to find the shortest path through a weighted graph from node s to node t. (a) the graph. (b) the search process showing successive refinements of the partition: subsets are named by the path that they have in common. (c) the complete search displayed as a tree, with nodes labelled with the bounds.

2b)

STAGE	SUBSET REFINED	Subsets comprising partition, labelled with bound g.
1	Start	⟨s⟩ ,0
2	⟨s⟩	⟨sa⟩,1 ⟨sb⟩,2
3	⟨sa⟩	⟨sad⟩,7 ⟨sb⟩,2
4	⟨sb⟩	⟨sad⟩,7 ⟨sbt⟩,10 ⟨sbc⟩,3 ⟨sbe⟩,9
5	⟨sbc⟩	⟨sad⟩,7 ⟨sbt⟩,10 ⟨sbcd⟩,4 ⟨sbct⟩,6 ⟨sbe⟩,9
6	⟨sbcd⟩	⟨sad⟩,7 ⟨sbt⟩,10 ⟨sbcds⟩,7 ⟨sbcdt⟩,8 ⟨sbct⟩,6 ⟨sbe⟩,9
7	⟨sac⟩	⟨sacd⟩,7 ⟨sact⟩,9 ⟨sad⟩,7 ⟨sbt⟩,10 ⟨sbcds⟩,7 ⟨sbcdt⟩,8 ⟨sbct⟩,6 ⟨sbe⟩,9
8	⟨sbct⟩	terminates with shortest path ⟨sbct⟩, length 6.

Using this g in the branch-and-bound algorithm applied to graph search leads to a guided search, guaranteed to return the minimum. Again irrelevant repetitions can be eliminated - if two partial paths have a common end point, one can coarsen the partition as in the non-heuristic case, but there is a note of caution: retroactive coarsening could prune a branch of the tree and not the new leaf, unless the monotonacity property holds for g. This is the "consistency condition" of Hart et al,(6) and allows a 'tidy' algorithm, but in no way affects the correctness of the solution obtained.

A note on difference of emphasis between the approach here, and the more standard approach (6,9) to graph search, must be made to avoid confusion. We have seen above that in partitions, sets defined by partial paths with common end points should

in the interests of efficiency be united: the important parameter of the subset is the terminal node of the common initial path. In the standard approach the emphasis is upon the nodes, with the common paths defined implicitly by some form of chaining. The search for the minimum for next branching is over the node set, restricting the search to a subset of nodes which are "open" or "visited" with the consistency condition ensuring that previously "closed" or "developed" nodes do not need to be re-opened. Relating this terminology to our viewpoint, the open nodes are those at the end of a common path for some set in the partition, and these become closed when the corresponding subset is selected for subdivision.

The approach to graph search given here, and the standard approach, lead to the same algorithms: the approach via branch-and-bound serves to add fresh insight into the workings of unidirectional graph searches. Later, in more complex searches, the branch-and-bound approach will fully justify itself with the generation of powerful new algorithms.

4.0 Multiple Branch-and-Bound

The natural extension of the branch-and-bound algorithms discussed in section 2 is to use more than one partition, refining each partition in turn (or in some arbitrary order). At each stage of the search we would have an effective partition on the set of the product of the separate partitions:

Definition: given a set of partitions $\pi_1,\pi_2,\ldots,\pi_n$ we shall define a product partition as

$$\Pi = \pi_1 \times \pi_2 \times \ldots \times \pi_n = \{A=\bigcap_{i=1}^{n} A_i : A_i \in \pi_i\} \quad \square$$

The algorithm follows, again minimising a function f on a set X, using as an aid a lower bound function g and a new function q. The minimum will be located in a subset of the product partition, and since this grows much more rapidly than the individual component partitions, the actual search would be expected to be considerably more effi cient.

As previously, $g : 2^X \to R$ is defined by

$g(A) \leq f(x)$ for all x in A, and $A \subseteq X$

and $g(\{a\}) = f(a)$ for all a in X.

In addition we are given a "combination" function q such that

$$g^*(\bigcap_{i=1}^{n} A_i) \geq q(g(A_1),\ldots,g(A_n)) \geq \max(g(A_1),\ldots,g(A_n))$$

for any $A_1 \in \pi_1, A_2 \in \pi_2, \ldots, A_n \in \pi_n$,

where $g^*(A)$ is the greatest lower bound of f in A and where $\pi_1,\ldots,\pi_n$ are partitions that we will progressively refine. We shall denote by m_1, the minimum g(A) for A in π_1:

$$m_1 = \min_{A \in \pi_i} (g(A)).$$

Two parameters x* and MIN will be used to record the best point found so far.

START

STEP 1. Set MIN=∞, and $\pi_j=\{X\}$ and $m_j=g(X)$ for all j.

ITERATION

STEP 2. Choose any of the π_j for further refinement.

STEP 3. Suppose that π_s selected:

3.1. Select $Y_m \in \pi_s$ such that $g(Y_m)$ is minimal. If more than one, choose any of them.

3.2. Partition Y_m into $Y_{m1},Y_{m2},\ldots,Y_{mk}$.

3.3. In π_s, replace Y_m by $Y_{m1},\ldots,Y_{mk}$.

3.4. Check whether $Y_{mi} \cap Z$ is a single point x for all i=1,...,k and $Z \in \Pi_s$ (where Π_s is the product partition of all the separate partitions other than π_s)

If it is, then evaluate f(x) and if f(x) < MIN then change MIN to f(x) and x* to x.

3.5. Update m_s to the minimum of the g(Y) for $Y \in \pi_s$.

STEP 4. If MIN > $q(m_1,m_2,\ldots,m_n)$ then

repeat from STEP 2: otherwise, terminate with STEP 5.

TERMINATION

STEP 5. x* is a minimum with value MIN. □

The choice of partition to refine at Step 2 is entirely arbitrary, and could be random, or each in turn, or always choosing the one with fewest members. We could even always choose the same partition, in which case we obtain a variation of the simpler algorithm of section 2.

That this algorithm does indeed return a minimum point follows from the observation that the algorithm terminates when first

$$MIN \leq q(m_1, m_2, \ldots, m_n)$$

from which we deduce

$$MIN \leq q(g(A_1), g(A_2) \ldots, g(A_n))$$

for all $A_1 \in \pi_1$, $A_2 \in \pi_2$, etc.

from the definition of the m_i's. Hence from the definition of q

$$MIN \leq g^*(A_1 \cap A_2 \cap \cdots \cap A_n)$$

$$= g^*(A) \qquad \text{all } A \in \Pi$$

and hence

$$MIN \leq \min_{A \in \Pi} g^*(A) .$$

Note that while partitioning and associated searches occur for the individual partions π_i, the minimum is identified by reference to the product partition Π. The terminating condition is encountered by virtue of the properties of g and q.

Note that Step 3.5 and Step 4 could be replaced by directly testing at a modified STEP 4 whether MIN > g(Z) for all Z in the product partition Π. The usefulness of the function q is that it obviates this necessity and introduces a radical improvement in efficiency over this more naive approach. The simplest form for q, always possible, is $q(x_1, \ldots, x_n) = \max(x_1, \ldots, x_n)$; but as we shall see, other functions are also possible.

4.1 Application to Bidirectional Graph Searching

The bidirectional graph searching problem (9) can be very naturally solved using the multiple branch-and-bound algorithm. Two partitions would be used, generated from partial paths anchored at either end: thus one partition is determined by subsets defined by common initial paths, (as in the Dijkstra search) while the other partition is determined by subsets defined by common final paths. Two subsets (one from each partition) intersect in a point when they both have the same extreme point on their defining partial paths. Heuristic and non-heuristic searches differ both in the lower bound function used (the function g) and in the way these combine on intersection (the function q). This double distinction is important. The general bidirectional graph-search will be detailed below, before proceeding further: refinements of efficiency will be overlooked in the interests of clarity, and as in the unidirectional form, unnecessary partial paths will be retained.

The problem is to find the shortest path through a graph from node s to node t. The set to be searched is all possible loop-free paths starting at s and terminating at t. We shall call the two partitions π_s and π_t and label the elemental subsets of the partitions by the partial paths defining them (as we did previously in section 2.1). X will have two representations as <s- and <t>.

START

STEP 1. Set $\pi_s = \{<s>\}$ and $\pi_t = \{<t>\}$ and MIN=∞. $m_s = m_t = 0$.

ITERATION

STEP 2. Choose to refine either π_s or π_t.

STEP 3. Suppose that π_s were chosen: a symmetrical set of steps would be done if π_t were chosen.

3.1. Find the set <s...x> in π_s such that g(<s...x>) is minimal. If there are more than one, choose any of them.

3.2. Partition <s...x> by finding all the successors $y_1, \ldots, y_k$ of node x (i.e. all nodes connected by a single edge from x) but rejecting all successors y on the path s...x , and form sets $<s...xy_1>, \ldots, <s...xy_k>$.

3.3. Replace $\langle s...x \rangle$ by $\langle s...xy_1 \rangle$, $\langle s...xy_2 \rangle, .., \langle s...xy_k \rangle$.

3.4. The previous step will have produced a lot of new single point sets in the product partition $\pi_s \times \pi_t$. Search all additions to $\pi_s \times \pi_t$ (i.e. search the set $\langle s...xy_i \rangle \cap Z$ for all $i=1,...,k$ and all $Z \in \pi_t$) to see whether these are single points (the matching of the last symbol in the label of the set of π_s with the first symbol of the label for the set of π_t determines this) and if the intersection is a single point $\langle s...xy_i...t \rangle$, if $f(\langle s...xy_i...t \rangle) < MIN$ update MIN and x* respectively to $f(\langle s...xy_i...t \rangle)$ and $\langle s...xy_i...t \rangle$.

3.5. Update m_s to the minimum of $g(Y)$ for $Y \in \pi_s$.

STEP 4. If $MIN > q(m_s, m_t)$ then repeat from STEP 2.

TERMINATION

STEP 5. x* is path of minimal length M1N. □

Heuristic and non-heuristic search have the following particular bounding functions g and q:

(1) Non-heuristic

The bounding function g is the length of the common path

$$g(\langle x_o...x_m \rangle) = w(\langle x_o...x_m \rangle).$$

The combination function q is given by

$$q(g(A), g(B)) = g(A) + g(B).$$

The intersection of two sets implies having two partial paths (one from each set) which must be common to all paths in the intersection set (note that our sets arc such that one path cannot be included in the other): hence one can compute a bound for the intersection simply as the total length of path common to all paths in the set.

(2) Heuristic

The bounding function g is given by

$$g(\langle x_o...x_m \rangle) = h(s, x_o) + w(\langle x_o...x_m \rangle) + h(x_m, t)$$

where h(a,b) is the heuristic estimator of the minimal path length from a to b, and must underestimate. For the one partition $x_o = s$ and for the other partition $x_m = t$. The combination function is given by

$$q(g(A), g(B)) = Max(g(A), g(B)).$$

The usual form of these algorithms includes an extra coarsening of the partition at Step 3.3, removing redundant sets as in the unidirectional case. Two different sets with labels matching at beginning and end need not both be retained, only that with shorter defining path being needed. The resulting algorithms for bidirectional graph searching are similar to existing ones. (9), The heuristic form is almost identical to the general algorithm VGHA of Ira Pohl (9), but our non-heuristic form is neither a simple specialisation of VGHA with h(x,y) = 0 nor a restatement of Pohl's VGA, but is something of a marriage of the two. The heuristic and non-heuristic cases have in the past been tackled separately: here they are united in a single general algorithm.

That our algorithms work follows from the fact that they are particular cases of the multiple branch-and-bound algorithm: there is no need to worry about the subtle termination conditions that have proved troublesome in the more conventional approaches.(9)

4.2 Graph searches with subgoals

In graph searching, why should we stop at a mere two partitions? We can introduce further partitions by choosing a point as a possible intermediate node, and then in addition conduct further explorations from there. This in problem solving is the setting up of subgoals. No matter what the number of subgoals, whether these are added to during search, whether they are in some sense parallel or sequential (i.e. whether they are alternatives, or whether the solution path is thought likely to pass through them both), the algorithm will be the same. Further, the search can be heuristically guided or not, using the functions g and q as previously given.

FIGURE 3. Graph search with subgoals. Graph as in Figure 2a, working forwards from s, backwards from t, and backwards from c.

STAGE	PARTITIONS (partial)												q	MIN	x^*
	Π_s			m_s	Π_t				m_t	Π_c		m_c			
Start	⟨s⟩,0			0	⟨t⟩,0				0	⟨c⟩,0		0			
1	⟨sa⟩,1		⟨sb⟩,2	1									1		
2					⟨dt⟩,4	⟨ct⟩,3	⟨et⟩,2	⟨bt⟩,8	2				3	10	⟨sbt⟩
3										⟨ac⟩,5	⟨bc⟩,1	1	4	6	⟨sbct⟩
4	⟨sac⟩,6	⟨sac⟩,7		2									5		
5							⟨bet⟩,9		3				6		
Termination; ⟨sbct⟩ shortest path, length 6															

FIGURE 4. Two examples of binary trees.

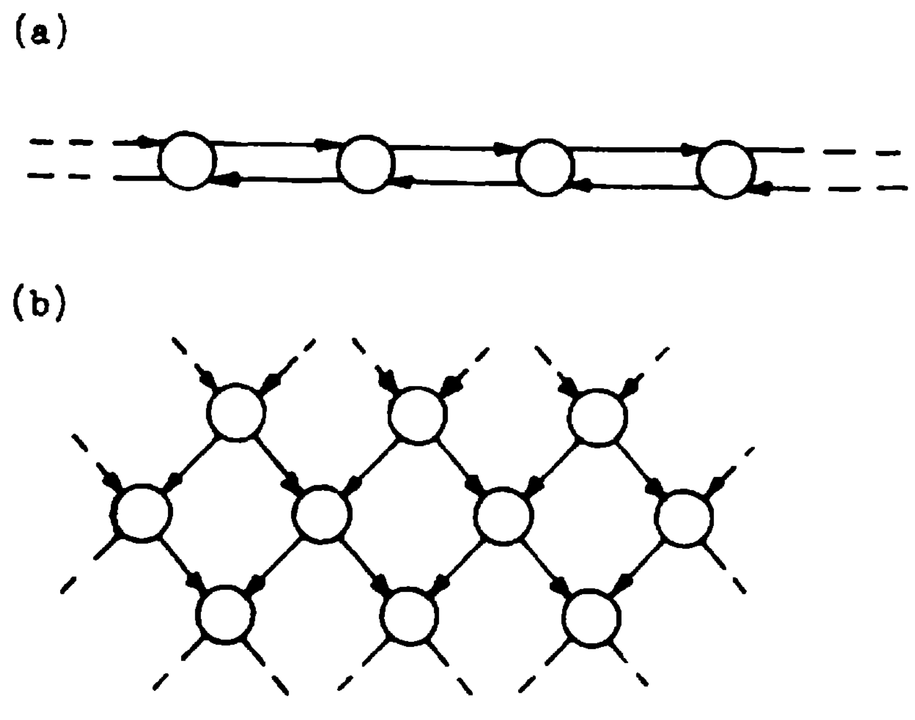
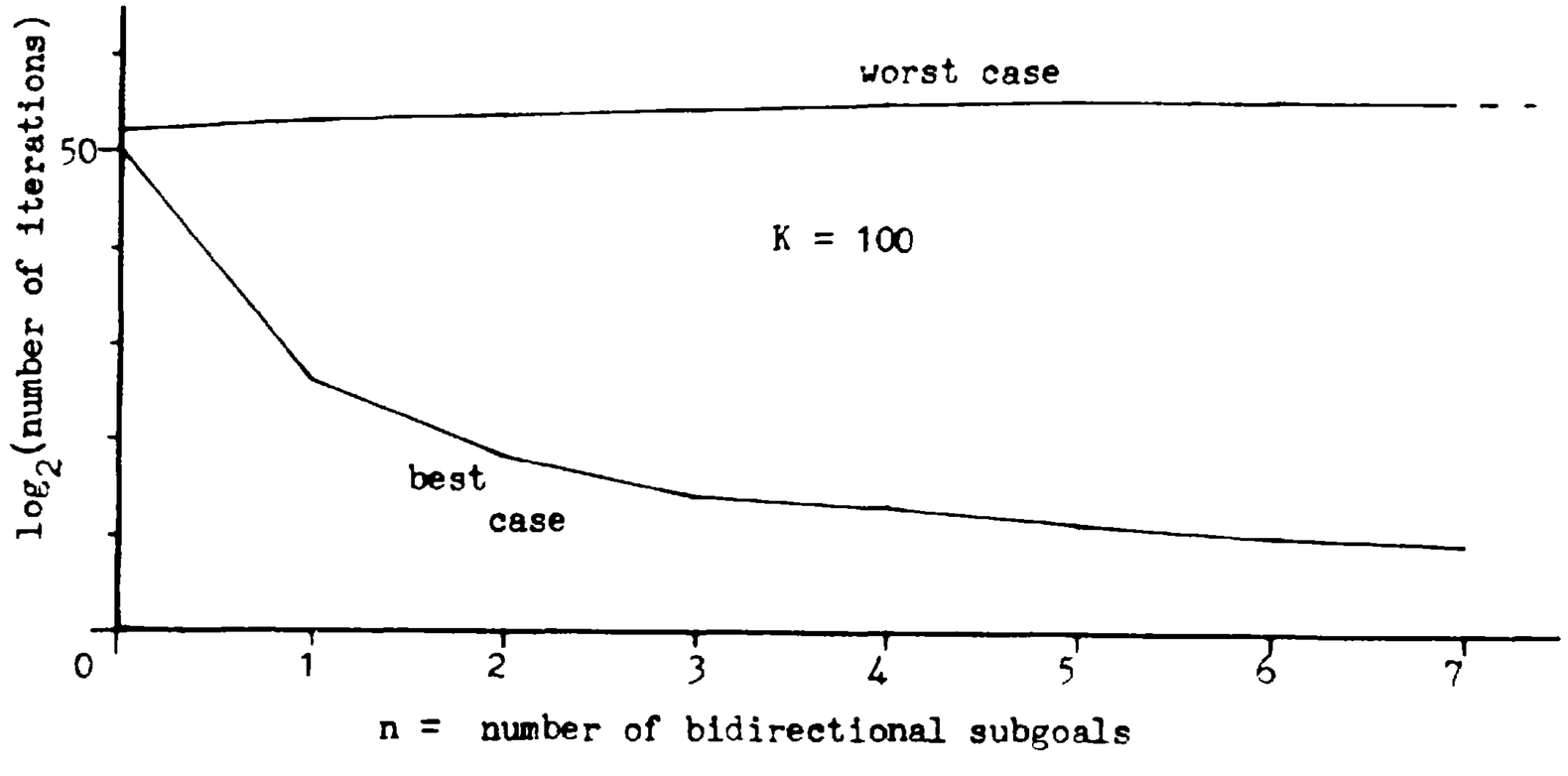


FIGURE 5. Savings in the search of a binary tree using subgoals. Index of performance, $\log_2$(number of iterations), plotted against number of bidirectional subgoals (n). n=0 means simple bidirectional search. Unidirectional search has performance index of 100.



The algorithm will be the full multiple branch-and-bound version of the previous bidirectional graph search, but with one generalisation. All except one of the partitions will be permitted to be partial, by which we shall understand that while the sets of the individual partitions must still be disjoint, they need not necessarily unite to the complete set; the appropriate generalisation of the product partition will be

$$\Pi = \{A = \bigcap_{i=1}^{n} A_i : A_i \in \pi_i, \text{ or } A_i = X$$

if TT. is partial}.

Note that Il is not necessarily a partition since its component subsets are not necessarily disjoint. But this does not matter, and using this n at Step 3.4 gives the required algorithm.

To illustrate this algorithm, Figure 3 displays the search of the graph of Figure 2(a) working forwards from s, backwards from t, and backwards from a single subgoal c. Note that the subgoal only helped here because it was on the solution path: the saving is small, because the problem is small. The advantages of the extension to subgoals will only be apparent on large problems, and especially where looking for any path, when the fortuitous choice of the subgoal on the minimal path is not necessary.

To obtain some insight into the savings possible with subgoals, let us analyse a special class of graphs. Let us define a binary tree as a graph in which all nodes have exactly two edges entering them and two edges leaving them; all edges have unit weight and there are no loops. Trees must hence have infinitely many nodes. Two examples are shown in Figure 4. (Note that we are not intending rooted trees here.)

In a binary tree, starting at a point s there are exactly 2^K paths of length k. There may be 2^k or less nodes at distance k, depending on whether paths converge or not. Suppose that we are required to search a binary tree for a route between two nodes, s and t, of minimal distance K apart (there may well be routes of length greater than K as well). We shall consider only non-heuristic searches with each partition taken in turn, and no coarsening to remove redundant paths. Then a unidirectional search will undergo between 2^{K-1} and 2^K iterations.

Simple bidirectional search will undergo between $2.2^{[K/2]-1}$ and $2.2^{[K/2]+1}$ iterations, where [K/2] means the integral part of K/2.

If we have a bidirectional subgoal (i.e. a subgoal searched in both directions, thus defining a pair of partitions) situated halfway along a minimal path, the number of iterations will be between $4.2^{[K/4]-1}$ and $4.2^{[K/4]+1}$. In general with n-1 bidirectional subgoals at equal distances along the same minimal path the search will involve between $2n.2^{[K/2n]-1}$ and $2n.2^{[K/2n]+1}$ iterations. An arbitrary set of n-1 bidirectional subgoals cannot be better than this last case, but cannot be worse than for n times the figure for simple bidirectional search. A graphical comparison is given in Figure S, for the case K = 100.

The use of subgoals in problem solving is a gamble, for you may take longer to solve the problem, but you may well save dramatically: however, you cannot as a result fail completely if the subgoals are used within the framework of a multiple branch-and-bound algorithm, for if all else fails, the search must succeed as a unidirectional search.

0 References

1. Barrow,H.C. and Popplestone,P.J ., "Relational Descriptions in Picture Processing", Dept. of Machine Intelligence and Perception, Univ. of Edinburgh. July 1970.
2. Purstal],P.M.,"Tree searching methods with an application to a network design problem", Machine Intelligence 1, pp65-87. Oliver & Boyd, Edinburgh. 1967
3. Doran,J.,"An approach to automatic problem-solving", Opus sit. pp1C5-123. 1967
4. Dijkstra,E.,"A note on two problems in connection with graphs", Numerische Mathematik. 1, pp269-271. 1959.
5. Hart,P.,Nilsson,N.J. and Raphael,B.,"A formal basis for the heuristic determination of minimum cost paths", IEEE. SSC-I. pp1C0-1C7, July 1968. =
6. Lawler,E.L. and Wood,D.E.,"Branch-and-bound methods: a survey", Operations Research.14. PP699-719. 1966. =
7. Montanari,Ugo,"Heuristically guided search and chromosome matching", Artificial Intelligence - 1, PP227-245. 1970.
8. "Morris,R."Scatter Storage Techniques", Comm. ACM,11,1,pp38-44,Jan.3968.
9. PohI7Ira,"Bi-directional and Heuristic Search in Path Problems", Stanford Linear Accelerator Centre Report 104, May 1969.
10. Tewarson,R.P. "Computation with Sparse Matrices", SIAM Review. 12.4.PP 527-544. 1970.