Preprocessing Search Spaces for Branch and Bound Search

by

Qiang Yang, Dana S. Nau

Research Report CS-89-32
August, 1989

Preprocessing Search Spaces for Branch and Bound Search

Qiang Yang * Dana S. Nau†

Abstract

Heuristic search procedures are useful in a large number of problems of practical importance. Such procedures operate by searching several paths in a search space at the same time, expanding some paths more quickly than others depending on which paths look most promising. Often large amounts of time are required in keeping track of the control knowledge.

For some problems, this overhead can be greatly reduced by preprocessing the problem in appropriate ways. In particular, we discuss a data structure called a *threaded decision graph*, which can be created by preprocessing the search space for some problems, and which captures the control knowledge for problem solving. We show how this can be done, and we present an analysis showing that by using such a method, a great deal of time can be saved during problem solving processes.

*Computer Science Department, University of Waterloo,Waterloo, Ontario, N2L 3G1 *E-mail address: qyang@dragon.waterloo.edu*

†Computer Science Department, Institute for Advanced Computer Studies, and Systems Research Center, University of Maryland, College Park, Md. 20742. This work was supported in part by an NSF Presidential Young Investigator award, with matching funds provided by Texas Instruments and General Motors Research Laboratories, and in part by NSF Grant NSFD CDR-88003012 to the Systems Research Center. *E-mail address: nau@mimsy.umd.edu.*

1 Introduction

Heuristic search procedures are useful in a large number of problem domains. Most heuristic search procedures (for example, A*, SSS*, B*, AO*, and alpha-beta) have been shown to be special cases of best-first Branch-and-Bound search [3, 5]. These procedures search several solution paths at the same time, expanding some paths more quickly than others depending on which paths look most promising.

One source of computational overhead during such search is the time spent keeping track of the alternate partial solutions that the procedure is examining. Typically, the search algorithm stores these partial solutions on a list called the *agenda*, *open list*, or *active list*, in order of their estimated cost. Any time a new partial solution is generated, its estimated cost must be compared with the estimated costs of the other partial solutions already on the list, in order to find the appropriate place to put it on the list. Thus, significant computational overhead is required just to maintain the active list.

Certain problem domains have special properties which allow us to eliminate this overhead. In particular, in some problem domains it is possible to do automatic preprocessing or "compiling" of the search space, to extract control knowledge which can be used to do the heuristic search without having to maintain the active list explicitly. In this paper, we discuss what kind of domains allow us to do such preprocessing, how to do the preprocessing, how to use the information gathered from the preprocessing, and how much time can be saved by doing this preprocessing. We also present an example from a problem domain of particular interest to us: generative process planning for the manufacture of machined parts.

2 Problem Characteristics

Preprocessing of the search space is possible whenever the following conditions are satisfied:

1. With the exception of feasibility/infeasibility of nodes, the search space has the same shape regardless of the particular problem instance being considered. Thus, if P and P' are two problem instances, then there is a one-to-one mapping between the nodes of their search spaces S and

S', such that if some feasible node a in S has n children b_i, $i = 1 \ldots n$, then its corresponding node a' in S' will either have n children b'_i, for $i = 1 \ldots n$ such that b_i corresponds to b'_i, or be infeasible.

2. Corresponding nodes need not have the same cost—but if two nodes a and b in S_1 have $\text{cost}(a) > \text{cost}(b)$, then for the corresponding nodes a' and b' in S', we must have $\text{cost}(a') > \text{cost}(b')$.

Nearly all heuristic search problems satisfy the first condition above, but fewer satisfy the second one. However, there are still many problems which do satisfy it. One example is generative process planning for the manufacture of machined parts. In our generative process planning system SIPS[6], a machinable part is considered to be a collection of machinable features. For each feature, SIPS finds an optimal plan for that feature via a best-first Branch-and-Bound search in its knowledge base. SIPS's knowledge base consists of information on a large number of different machining processes, organized in a taxonomic hierarchy. For two different machinable features, different machining processes may have different costs–but if process a costs more than process b on feature f, it generally also costs more than process b on feature f' as well.

Robotic route planning[4] is another problem domain which also satisfies the second condition. In this domain, a route map corresponds to a graph, in which a node represents a region and an arc represents connectivity between two regions. The cost of a node can be the time needed to pass through the region it represents, and the feasibility of a node indicates whether or not the region is passable. In general, although the passability of a region may change from one problem instance to another, the relative costs of the regions stay the same. In such cases, preprocessing is possible.

3 Threaded Decision Graphs

Suppose that for a given problem instance the search space is as shown in Figure 1. The goal is to search for the cheapest leaf node for which all the nodes along the path from the root are feasible. In Figure 1, suppose the problem solver is checking the conditions associated with node c. If c succeeds, we know that the next node to check will be f, since c must be the current minimum costly node to be checked, and f's cost is the same as c's.

Similarly, if c fails, the next node to check in the search space must be j. This is because at the time c is being checked, nodes a, b, d, e, h and k must all have been checked, since they all have costs less than that of c's. Control information like this is independent of the particular problem instances, and can be gathered before the problem solving process starts. Therefore, we can assign a *success thread* from node c to node f, and *failure thread* from c to j. During the course of finding the cheapest actions to achieve a given goal, once we reach node c, if all the conditions are satisfied, we can follow the success thread of c to get to the next node to be checked. On the other hand, if they are not satisfied, then the failure thread at c can be followed to get to the next node to be checked. Figure 2 shows such a structure.

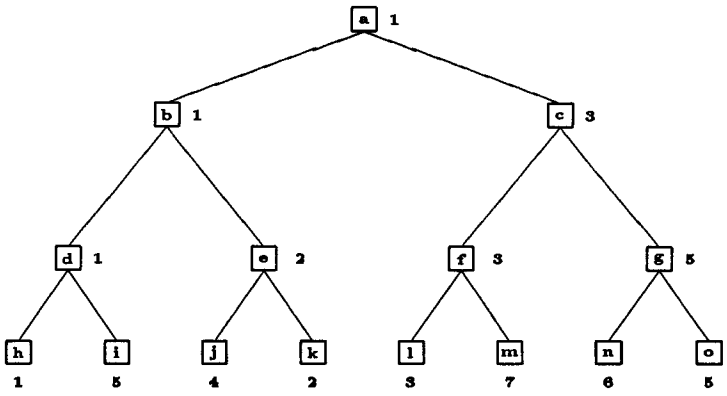

Figure 1: An example search space. The numbers associated with the nodes are their cost values.

If all the deterministic control information is gathered, it is no longer necessary to maintain an active list for storing control information. Before problem solving starts, the information can be used to construct a data structure that contains one or more occurrences of each action, and for each action occurrence there is a success pointer and a failure pointer. We will call such a data structure a *threaded decision graph* of the original search space. Problem solving can then be completely guided by its threaded decision graph. Figure 3 shows the threaded decision graph for the search space in Figure 1.

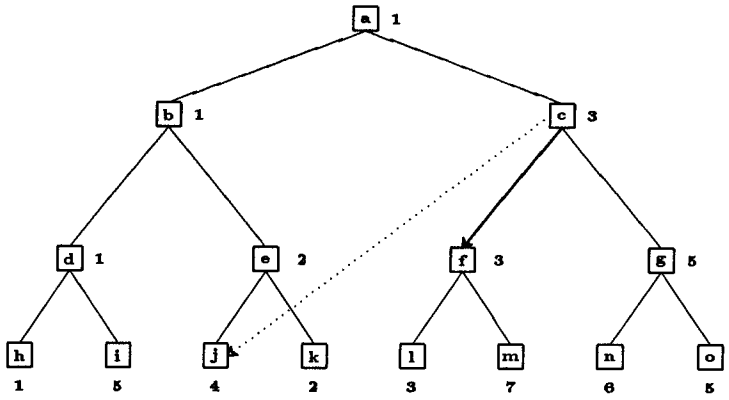

Figure 2: The search sapce in Figure 1 with threads inserted at node c. The think solid line is c's success thread, and the thick dotted line its failure thread.

Two special nodes in Figure 3 are worth noting. The node marked "success" is called a *success node*. If this node is reached through the success threads, the search process terminates with success. The path from the root node to success in the threaded decision graph contains the solution path which should be returned. The node marked "fail" is called a *failure node*, which marks the termination of the search process without success.

In the following, we first discuss in detail how problem solving is done with the help of threaded decision graphs. We then consider how such a data structure could be automatically constructed.

4 Search with the Threaded Decision Graphs

Given a search space, assume that we have constructed its threaded decision graph. A request to solve the problem would correspond to finding out the cheapest path in the tree. For the given threaded decision graph, this is done by checking the conditions associated with each node, starting from the root node. The success thread of the current node should be followed if the node is tested to be feasible, and the failure thread should be followed if it is infeasible. Each node is also marked as "success" or "failure" depending on

the result of its feasibility test.

If, during the traversal process, a success node is reached via a success thread, then the search ends with success.

If a failure node is reached through failure thread, then no actions satisfying the criteria exist. In this case, search is terminated with failure.

After tracing down the failure thread of some node, it is possible to reach some other node whose ancestor in the original search tree failed the feasibility test. For example, suppose node b in figure 3 is infeasible and node f is being checked. If f is infeasible then the next node to be checked should be j. Since b is an ancestor of j which failed, no feasibility test need be done at j, and the failure thread of it should automatically be followed. In general, whenever a node n is reached via some failure thread, it should always be checked to see if n has any ancestor m which failed the feasibility test. If so, the failure thread of node n should be followed, without conducting its feasibility test.

To check if a given node has any infeasible ancestor is straightforward: we can retrieve the traversal information associated with the parent node of the given node in the original search space. It has an infeasible ancestor if its parent node is unvisited or marked as infeasible.

5 The Construction of Threaded Decision Graphs

In this section, we consider the problem of how the threaded decision graphs are constructed. For simplicity, we assume the search space is finite. Later we will discuss cases where the search space is not finite.

With the above restrictions, the following algorithm generates the threaded decision graph. This algorithm is a modified version of best-first search.

procedure *Construct*
  $A := \{ a_0 \}$
(A is the active list, and a_0 is the root of the search space)
  While A is not empty do
  begin
    $n :=$ pop(A);
    IfA is empty, then

```
    Failure-thread(n):=Failure-node
  else Failure-thread(n):= head(A);
  If n is a leaf node, then
    Success-thread(n):=Success-node
  else begin
          A:=insert(A, Children(n));
          Success-thread(n):=head(A)
  end{else}
 end{while}
end Construct
```

During preprocessing of the search space, each node in the tree appears at the head of the active list once and only once, and the resultant threaded decision graph has the same number of nodes as the original search space. Since every node has only two threads, the success and the failure threads, the number of threads of the threaded decision graph is twice the number of nodes as that of the search space. The same argument also guarantees the absence of cycles of threads.

6 Very Large or Infinite Search Spaces

If the search space is very large, the time it takes to preprocess it may be prohibitive. In such cases, and in cases when the search space is infinite, we can preprocess a finite portion of it. This partial threaded decision graph can be created by running the algorithm *Construct* for a finite number of iterations, until one or more goal states appear in the graph.

During the problem solving process, the threaded decision graph can be used in the same manner as described in the previous sections, until a node which has no success and failure threads is reached. At this point, conventional Branch-and-Bound search can be utilized as follows: An active list is constructed by including all of the successor nodes of the current node and the children of the feasible nodes along the path back to the root node in the state space. This list is sorted according to the costs of its nodes, and least costly of these is expanded.

The construction of the threaded decision graph may also be done with the following strategy: During search through the search space for each particular

problem instance, every time a node is tested we will look for its success or failure threads. If they do not exist, we will use the search procedure to find out which node we should test next. The success or failure threads can then be assigned to this node. In other words, the threaded decision graph is partially built on the job.

7 Analysis

By gathering all the possible static control information beforehand, preprocessing of the search space can greatly improve problem solving efficiency. In order to see this claim more quantitatively, consider a simplified example. Assume that the search space is in the shape of a tree with a branching factor of m and depth k (the root of the tree has a depth of 1). To see how much time the problem solver spends on manipulating the list of incomplete paths using conventional search methods, consider the following two extreme cases.

In the best case, the search procedure expands only one path down the tree. The time for manipulating the active list is $O(km \log m)$.

In the worst case, the search procedure expands the tree in a breadth first manner. The total time spent on manipulating the list in this case is bounded by $m \times \sum_{i=1}^{m^{(k-1)}} \log(i \times m) = O(m^k \log m^k)$.

On the other hand, search with a threaded decision graph is guided completely by the success and failure threads, and no additional computational effort is needed for manipulating the search control information. Therefore, the time saved by using the threaded decision graph to guide the search procedure can be between $O(km \log m)$ and $O(m^k \log m^k)$, where k is the depth of the tree and m is the branching factor.

8 Discussion

In this paper we presented a technique for preprocessing search spaces in order to gather control information for problem solving. We also discussed a number of conditions under which the method works. Through complexity analysis, we demonstrated that by using preprocessing, a great deal of computational effort is saved during the search process.

Our idea of preprocessing search spaces has some similarity to that of

threading of binary trees for tree traversal[2]. However, the way such threading is done, and the way it is used, are both clearly different from our threaded decision graphs.

As described in the paper, our technique only works on state-space graphs. An extension of the work is to allow the search space to be an AND/OR graph. When AND branches are allowed, the problem can be very complicated depending on how the costs are assigned its nodes. In the worst case, the threaded decision graph can contain an exponential number of the occurrences of nodes in the original search space. An example of this situation is given in Figure 4. The problem is that the i^{th} and $i+1^{th}$ best paths, for $i = 1, 2, \ldots$, are not next to each other, but on the left and right subtrees respectively. One way to tackle this problem is to *partially* preprocess the search space. For example, we can assign success and failure threads only to the nodes which are in the first k best paths.

Another problem which needs more attention is that the relative costs of some nodes in different problems may not be the same, but may change with different situations. However, if the cost of nodes change with different situations in some predictable manner, and if the number of such changes is finite, the preprocessing technique can still be made to work. For example, it may be possible to divide the situations into separate classes, each with a different threaded decision graph. Many domains satisfy this property, including process planning in automated manufacturing and robotic path planning.

References

[1] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, , Computer Science Press, Potomac, MD, 1978.

[2] D.E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Weseley, Reading, Mass., 1968.

[3] V. Kumar and L. Kanal, A General Branch and Bound Formulation for Understanding and Synthesizing And/Or Tree Search Procedures, *Artificial Intelligence,* (21), 1983, 179–198.

[4] T. A. Linden and J. Glicksman, Contingency Planning for an Autonomous Land Vehicle, In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1047–1054, Milan, August 1987. International Joint Committee on Artificial Intelligence.

[5] D.S. Nau, V. Kumar and L. Kanal, General Branch and Bound, and Its Relation to A* and AO* *Artificial Intelligence,* (23), 1984, 29–58.

[6] D.S. Nau, Automated Process Planning Using Hierarchical Abstraction. *TI Technical Journal*, 1987, 39–46, Award winner, Texas Instruments 1987 Call for Papers on Industrial Automation.

[7] N. Nilsson, *Principles of Artificial Intelligence*, Chapters 2 and 3, Tioga Publishing Co., 1980.

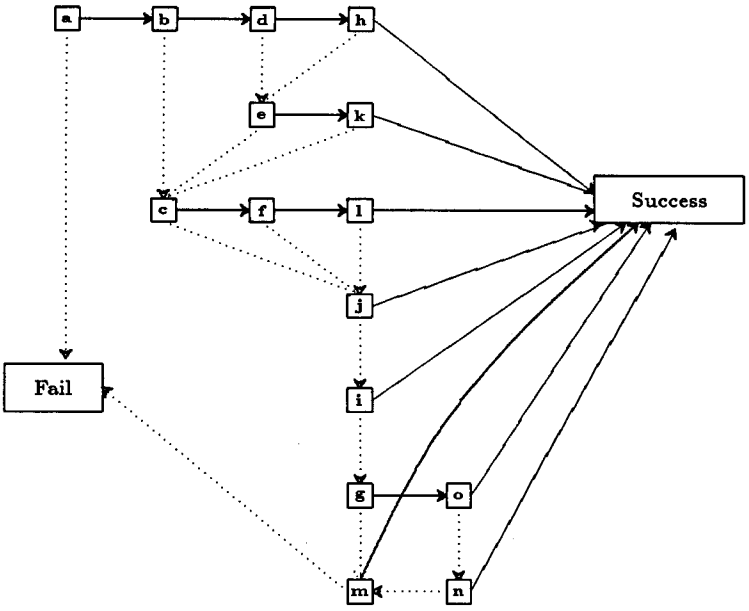

Figure 3: The threaded decision graph for the search space in Figure 1. In this figure, solid lines represent success threads, while the dotted ones the failure ones.

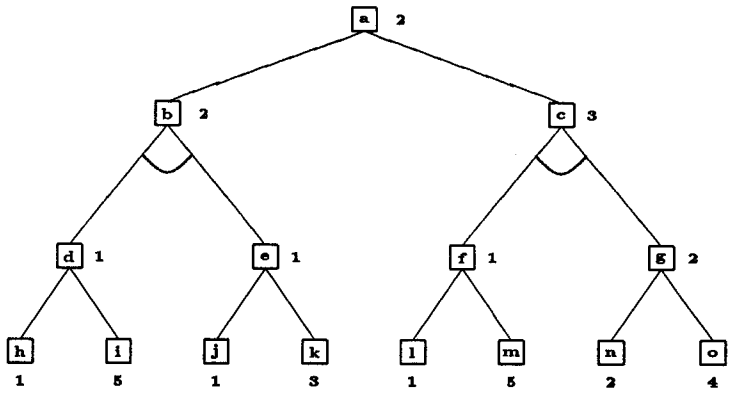

Figure 4: An example And/Or tree in the worst case situation.