**Iterative Deepening**

|  |  |  |  |
| --- | --- | --- | --- |
| Strategy | Selection from Frontier | Halts? | Space |
| Depth-first | Last node added | No | Linear |
| Breadth-first | First node added | Yes | Exponential |
| Best-first | Globally minimal *h(p)* | No | Exponential |
| Lowest-cost-first | Minimal *cost(p)* | Yes | Exponential |
| *A\** | Minimal *cost(p)+h(p)* | Yes | Exponential |

So far, none of the methods discussed have been ideal; the only ones that guarantee that a path will be found require exponential space . One way to combine the space efficiency of depth-first search with the optimality of breadth-first methods is to use iterative deepening. The idea is to recompute the elements of the frontier rather than storing them. Each recomputation can be a depth-first search, which thus uses less space.

Consider making a breadth-first search into an iterative deepening search. This is carried out by having a depth-first searcher, which searches only to a limited depth. It can first do a depth-first search to depth 1 by building paths of length 1 in a depth-first manner. Then it can build paths to depth 2, then depth 3, and so on. It can throw away all of the previous computation each time and start again. Eventually it will find a solution if one exists, and, as it is enumerating paths in order, the path with the fewest arcs will always be found first.

When implementing an iterative deepening search, you have to distinguish between

failure because the depth bound was reached and

failure that does not involve reaching the depth bound.

In the first case, the search must be retried with a larger depth bound. In the second case, it is a waste of time to try again with a larger depth bound, because no path exists no matter what the depth. We say that failure due to reaching the depth bound is failing unnaturally, and failure without reaching the depth bound is failing naturally.

1: Procedure IdSearch(G,s,goal)   
2:           Inputs  
3:                     G: graph with nodes N and arcs A   
4:                     s: set of start nodes   
5:                     goal: Boolean function on states   
6:           Output  
7:                     path from s to a node for which goal is true   
8:                     or ⊥ if there is no such path   
9:           Local  
10:                     natural\_failure: Boolean   
11:                     bound: integer   
12:                     Procedure dbsearch(⟨n0,...,nk⟩,b)   
13:                               Inputs  
14:                                         ⟨n0,...,nk⟩: path   
15:                                         b: integer, b ≥ 0   
16:                               Output  
17:                                         path to goal of length k+b   
18:                               if (b>0) then   
19:                                         for each arc ⟨nk,n⟩∈A do   
20:                                                   dbsearch(⟨n0,...,nk,n⟩,b-1)   
21:                               else if (goal(nk)) then   
22:                                         return ⟨n0,...,nk⟩   
23:                               else if (nk has any neighbors) then   
24:                                         natural\_failure←false   
25:           bound ←0   
26:           repeat  
27:                     natural\_failure←true   
28:                     dbsearch({⟨s⟩:s∈S},bound)   
29:                     bound ←bound+1   
30:           until natural\_failure   
31:           return ⊥

An implementation of iterative-deepening search, IdSearch, is presented. The local procedured bsearch implements a depth-bounded depth-first search (using recursion to keep the stack) that places a limit on the length of the paths for which it is searching. It uses a depth-first search to find all paths of length k+b, where k is the path length of the given path from the start and b is a non-negative integer. The iterative-deepening searcher calls this for increasing depth bounds. This program finds the paths to goal nodes in the same order as does the breadth-first search. As in the generic graph searching algorithm, to find more solutions after the return , the search can continue .

The iterative-deepening search fails whenever the breadth-first search would fail. When asked for multiple answers, it only returns each successful path once, even though it may be rediscovered in subsequent iterations. Halting is achieved by keeping track of when increasing the bound could help find an answer:

The depth bound is increased if the depth bound search was truncated by reaching the depth bound. In this case, the search failed unnaturally. The search failed naturally if the search did not prune any paths due to the depth bound. In this case, the program can stop and report no (more) paths.

The search only reports a solution path if that path would not have been reported in the previous iteration. Thus, it only reports paths whose length is the depth bound.

The obvious problem with iterative deepening is the wasted computation that occurs at each step. This, however, may not be as bad as one might think, particularly if the branching factor is high. Consider the running time of the algorithm. Assume a constant branching factor of b>1. Consider the search where the bound is k. At depth k, there are bk nodes; each of these has been generated once. The nodes at depth k-1have been generated twice, those at depth k-2 have been generated three times, and so on, and the nodes at depth 1 have been generated k times. Thus, the total number of nodes generated is

|  |
| --- |
| bk+2bk-1+3bk-2+···+kb |
|  | = | bk(1+2b-1+3b-2+···+kb1-k) |
|  | ≤ | bk(∑i=1∞ib(1-i)) |
|  | = | bk(b/(b-1))2. |

As there are (b/(b-1)) bk nodes in the tree, iterative deepening has a constant overhead of (b/(b-1)) times the cost of generating the nodes at depth n. When b=2 there is an overhead factor of 2, and when b=3 there is an overhead factor of 1.5 over generating the frontier. This algorithm is O( bk) and there cannot be an asymptotically better uninformed search strategy. Note that, if the branching factor is close to 1, this analysis does not work (because then the denominator would be close to zero);

Iterative deepening can also be applied to an A\* search. Iterative deepening A\* (IDA\*) performs repeated depth-bounded depth-first searches. Instead of the bound being on the number of arcs in the path, it is a bound on the value of f(n). The threshold starts at the value of f(s), where s is the starting node with minimal h-value. IDA\* then carries out a depth-first depth-bounded search but never expands a node with a higher f-value than the current bound. If the depth-bounded search fails unnaturally, the next bound is the minimum of the f-values that exceeded the previous bound. IDA\* thus checks the same nodes as A\* but recomputes them with a depth-first search instead of storing them.