Optimal Search

A search algorithm is optimal if no other search algorithm uses less time or space or expands fewer nodes, both with a guarantee of solution quality. The optimal search algorithm would be one that picks the correct node at each choice.

A* Algorithm

The best-first search algorithm is a simplification of an algorithm called A*, which was first presented by Hart [1968; 1972]. This algorithm uses the same f ', g, and h' functions, as well as the lists OPEN and CLOSED.

Algorithm:

1. The algorithm maintains 2 sets

a) OPEN list: It keeps track of those nodes that need to be examined

b) CLOSED list: It keeps track of nodes that have already been examined

2. Initially, OPEN list contains just the initial node and the CLOSED list is empty. Each node n maintains the following:

    a) g(n) = the cost of getting from the initial node to n

    b) h(n) = the estimate, according to the heuristic function of the cost of getting from n to the goal node

    c) f(n) = g(n) + h(n); this is the estimate of the best solution that goes through n.

3. Each node also maintains a pointer to its parent, so that later the best solution if found can be retrieved. A* has a main loop that repeatedly gets the node, call it n, with the lowest f(n) value from the OPEN list. If n is the goal node, then we are done and the solution is given by backtracking from n. Otherwise, n is removed from the OPEN list and added to the CLOSED list. Next all the possible successor node of n is generated.

4. For each successor node n, if it is already in the CLOSED list and the copy there has an equal or lower f estimate, then we can safely discard the newly generated n and move on. Similarly, if n is already in the OPEN list and the copy there has an equal or lower f estimate, we can discard the newly generated n and move on. If no better

version of n exists on either the CLOSED or OPEN lists, we remove the inferior copies from the two lists and set n as the parent of n. Also, calculate the cost estimates for n as follows:

a) Set g(n) to g(n) + the cost of getting from n to n

b) Set h(n) to the heuristic estimate of getting from n to the goal node

c) Set f(n) to g(n) + h(n)

Lastly add n to the OPEN list and return to the beginning of the main loop.

Some important observations about this algorithm are:

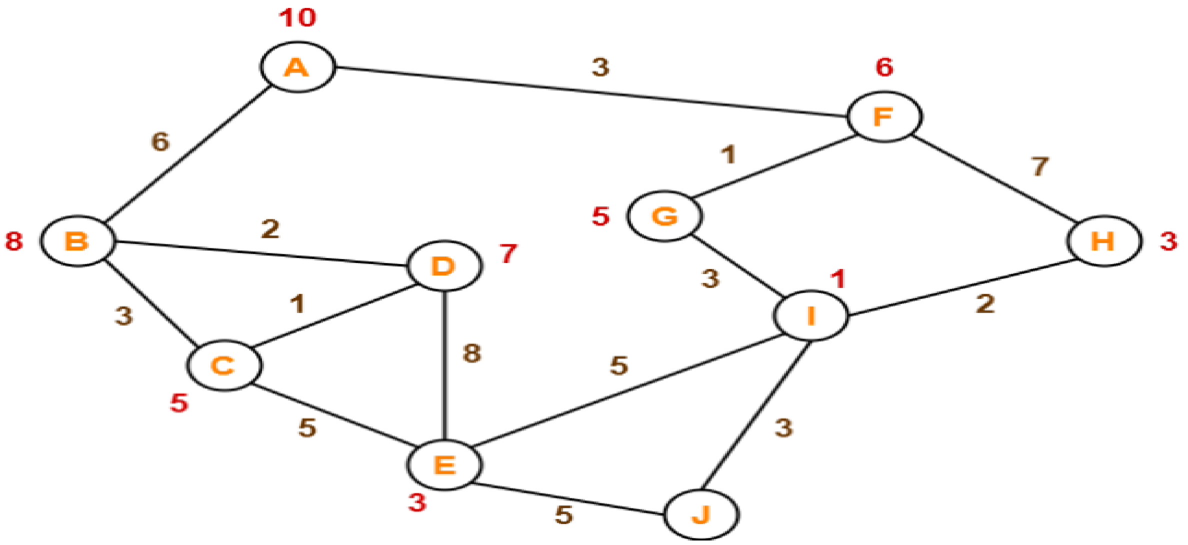
- In this algorithm, the role of the g function allows us to choose which node to expand next on the basis not only of how good the node itself looks (as measured by h), but also on the basis of how good the path to the node was. If we only care about getting to a solution somehow, we can define **g always to be 0**, thus always choosing the node that seems closest to a goal. If we want to find a path involving the fewest number of steps, then we set the cost of going from a node to its successor as a constant, usually 1.

- If h is the perfect estimator of the distance of a node to the goal, then A* will converge immediately to the goal with no search. The better the h is, the closer we will get to that direct approach. If, on the other hand, the value of **h is always 0**, the search will be controlled by g. If the value of g is also 0, the search strategy will be random. If the value of **g is always 1**, the search will be breadth first. If **h is always perfect**, in that case the A* algorithm is guaranteed to find an optimal path to a goal of one exists.

- By underestimating h for a particular node, we may waste some efforts in some cases. On the other hand, if we overestimate h, we cannot be guaranteed of finding the cheapest path solution unless we expand the entire graph until all paths are longer than the best solution.

- The A* algorithm was stated in its most general form as it applies to graphs. It can also be simplified to apply to trees by not bothering to check whether a new node is already on OPEN or CLOSED. This makes it faster to generate nodes but may result in the same search being conducted many times if nodes are often duplicated.

Consider the following graph- The numbers written on edges represent the distance between the nodes. The numbers written on nodes represent the heuristic value. Find the most cost-effective path to reach from start state A to final state J using A* Algorithm.



Solution-

Step-01: We start with node A. Node B and Node F can be reached from node A. A* Algorithm calculates f(B) and f(F).

f(B) = 6 + 8 = 14

f(F) = 3 + 6 = 9

Since f(F) < f(B), so it decides to go to node F. **Path- A → F**

Step-02: Node G and Node H can be reached from node F. A* Algorithm calculates f(G) and f(H).

f(G) = (3+1) + 5 = 9

f(H) = (3+7) + 3 = 13

Since f(G) < f(H), so it decides to go to node G. **Path- A → F → G**

Step-03: Node I can be reached from node G. A* Algorithm calculates f(I).

f(I) = (3+1+3) + 1 = 8

It decides to go to node I. **Path- A → F → G → I**

Step-04: Node E, Node H and Node J can be reached from node I. A* Algorithm calculates f(E), f(H) and f(J).

f(E) = (3+1+3+5) + 3 = 15

f(H) = (3+1+3+2) + 3 = 12

f(J) = (3+1+3+3) + 0 = 10

Since f(J) is least, so it decides to go to node J. **Path- A → F → G → I → J**

Iterative Deepening A*

Introduction

The A* has proven itself to the extent that other algorithms also adopted the good characteristics of A*. When A* is combined with Iterative Deepening, it is called IDA* or Iterative Deepening A*. The algorithm works the same as before but chooses better paths based on the heuristic values. IDA* is to A* what a depth bound search to DFS. The property of the A*, it gets an optimal solution if the heuristic used is optimal, also holds for IDA*. It is designed to make sure that the memory utilization is minimal. In that, one better node, other than children is kept and chosen for exploration if the children are found not better. It has linear space search strategy that means over a period of time, the amount of memory needed increases linearly. It provide linear space search at the cost of additional time spent in exploring same node multiple times. While the standard iterative deepening depth-first search uses search depth as the cutoff for each iteration, the IDA* uses the more informative f(n)=g(n)+h(n).

Iterative-deepening-A* works as follows: The IDA* is nearer to Depth First Depth Limited search rather than Iterative Deepening. Unlike both of them, IDA* does not explores all children level by level. It explores them by logical level of g + h'. It explores the search space in depth first way and calculate the g + h' values of each of the path being explored. Unlike conventional iterative deepening, where the exploration is of one arc every time, here it is the path with the typical length g + h'. For example we may decide that we will explore for g + h' value to be maximum 3, now we apply depth first search with applying heuristic to each one of the children and stop when it becomes greater than 3, and pick up another branch like we did in Depth First Depth Limited search. After exploring all children and the branches associated with them, we check if we get a solution somewhere. If we get the solution, we may quit. If not, we can either increase the maximum limit or stop if we cannot proceed further. We may, in above example, now increment the g + h' value to 6 and restart exploring from the root node. When we have explored and revised our estimates and so on, we have far more accurate measurements next time and it is more likely that our search leads to better direction this time. In fact this process demands more time but gets an optimal path. One can understand this process to be an Iterative deepening search where the algorithm proceeds by levels by pre-decided g + h' values each time and not physical levels.

IDA* Algorithm: IDA* is similar to iterative-deepening depth-first, but with the following modifications: The depth bound modified to be an f-limit

1. Start with limit = h(start)

2. Prune any node if f(node) > f-limit

3. Next f-limit=minimum cost of any node pruned

The cut-off for nodes expanded in an iteration is decided by the f-value of the nodes.

Like A*, IDA* is guaranteed to find the shortest path leading from the given start node to any goal node in the problem graph, if the heuristic function h is admissible, that is

$h(n) <= h^*(n)$

for all nodes n, where h* is the true cost of the shortest path from n to the nearest goal.

IDA* is beneficial when the problem is memory constrained. A* search keeps a large queue of unexplored nodes that can quickly fill up memory. By contrast, because IDA* does not remember any node except the ones on the current path, it requires an amount of memory that is only linear in the length of the solution that it constructs.

The formal algorithm can be described as follows:

```
1. The limit = h'(root node), current node = root node
2. Path = root node, Best node = root node
3. Pick up leftmost child of current node (if in a graph, one can pick up any directly connected node at random), and make it current
4. If it is goal node quit
5. Find g value of the child and apply h' to it.
6. If g + h' > limit
    a. update parent node estimate based on the best path and update it till the root node
    b. pick up another node as if reached to dead end (usually the right child of the parent)
    c. change path variable accordingly
7. otherwise
        a. If no node left in the tree
        b. change limit value to limit = g (best node) + h' (best node)
        c. Go to 2.
8. Otherwise,
        a. add this node to path
        b. if this node is better than best node = current node
        c. explore this node, go to 3
```

This is basically a DFS algorithm. Only when g + h' becomes greater, it stops and starts searching for another branch like Depth limited Depth First Search. We have named that variable as limit. So we explore the depth first tree till the child node's estimate goes beyond limit. This clearly helps us travel in the direction of goal. The nodes which are away from goal node have their h’ values higher than the nodes which are nearer and thus further (to goal) nodes crosses the limit value earlier than the closer (to goal) nodes. There are two different situations emerging from the exploration. First, the g + h’ value going above the limit which is similar to reaching a dead end. So we pick up the parent’s next child and explore further. If parent does not have a right child left we will pick up parent of the parent and so on. Another situation is when the limit value is reached for all branches of the root node. We will increase the limit to whatever best so far. This best so far must be higher than earlier best so far as we could not find the goal node so far. We will restart from the root node and explore the tree (or graph) yet again. This time the limit value is incremented sometimes by one node or two nodes etc based on the best node’s value.

The biggest **advantage** of IDA* is that it does not need the amount of space A* needs to have. Being depth bound, it requires the memory needed to be just one node at a time. The **other** important advantage is that the paths are explored which are near to solution as we are restricting the search based on g + h’ values and not with the length of the tree. The **third** advantage, is it uses optimized storage. The process is costly in terms of time, as it explores branches every time a fresh during a new iteration. The algorithm’s initial cut-off value can be decided based on the estimate of the root node IDA* algorithm ; i.e. the h’ value. If the h’ is admissible, i.e. underestimates h always, if we get a solution it must be optimal. If we do not get a solution, we can increment the value of cut off by the best so far node in the list of unexplored nodes. It is quite possible that the best so far node is the root node itself but with much tighter estimate. Now when we search, we are bound to get a solution which is within that limit. While h’ is admissible, the solution might not be received during the first few iterations but will never get beyond optimal.

Limitations of IDA*

First, it explores the entire tree again when searching the entire tree is failed to achieve the goal node. Though this process eliminates the storage requirement, it increases the time **Second**, as there is no real sense of direction, especially while exploring graph, many nodes are explored multiple times, as the exploration process is tree like. All paths to all nodes from every other node based on not crossing the limit are to be explored. This takes exponential amount of time if nodes are well connected with others like city routes. Thus the IDA* is better suited for the less connected graphs and not for others. **Third**, When we take an admissible heuristic function, it does not mean that we have the best possible heuristic function. We can actually choose a little worse heuristic function and make sure our optimum path is not worse than that small difference.

Recursive best first search

Another useful algorithm is recursive best first (RBF) search. Both IDA* and RBFS are memory bound searches. They are designed to make sure that the memory utilization is minimal. In that, one better node, other than children is kept and chosen for exploration if the children are found not better. The backtracking is based on depth first method and not like conventional best first search. The process only remembers one best so far node. When a node is explored, and all children are found to be of worse value of heuristic value, the remembered best so far node is picked up for exploration. Otherwise the best child is explored.

The best first search that we have seen earlier had an important advantage. It was able to move to the best node in any circumstances. It was able to keep the list of explored and unexplored nodes, able to avoid the exploration of already explored nodes by looking at the list of all explored nodes before exploring any child. It also maintained the list of unexplored nodes in the order of their merit so at any point of time, the best node can be explored.

Recursive best first cannot afford to have the list of explored as well as unexplored nodes. To limit the search space to linear, it modifies the original algorithm in a way that only one node is kept for backtracking.

The algorithm works like a normal depth first search with picking up the best node based on heuristic value. When all nodes are explored, the best node is explored but second best is preserved in memory. If the best node is explored and all children are found to be worse, the search process backtracks to the second best node (which is now best in the list including all the children as well as it). At the same point of time, the best child is kept as second best node now. At any given point of time, this search method only explores one node and remembers one more. Like depth first search, it also remembers the next node if it reaches a dead end. This algorithm quite expensive in terms of time though save on storage space.

Algorithm:

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
   return RBFS(problem,MAKE-NODE(problem.INITIAL-STATE),∞)
function RBFS(problem,node,f_limit) returns a solution, or failure and a new f-cost limit
   if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
   successors ← []
   for each action in problem.ACTIONS(node.STATE) do
         add CHILD-NODE(problem,node,action) into successors
   if successors is empty then return failure,∞
   for each s in successors do /* update f with value from previous search, if any */
         s.f ← max(s.g + s.h, node.f)
   loop do
         best ← lowest f-value node in successors
         if best.f > f_limit then return failure,best.f
         alternative ← the second-lowest f-value among successors
```

```
result,best.f ← RBFS(problem,best,min(f_limit,alternative))
if result ≠ failure then return result
```

Advantages

- More efficient than IDA*
- It is an optimal algorithm if h(n) is admissible
- Space complexity is $O(b^d)$.

Disadvantages

- It suffers from excessive node regeneration.
- Its time complexity is difficult to characterize because it depends on the accuracy of h(n) and how often the best path changes as the nodes are expanded.

Complete: Yes, like to A∗

Time: Exponential, depending both on the accuracy of f and on how often the best path changes as nodes are expanded

Space: linear in the depth of the deepest optimal solution

Optimal: Yes, like to A∗ (if h(n) is admissible)