Artificial Intelligence Coursework

Roderick Ewles

40330977

Breadth First Search

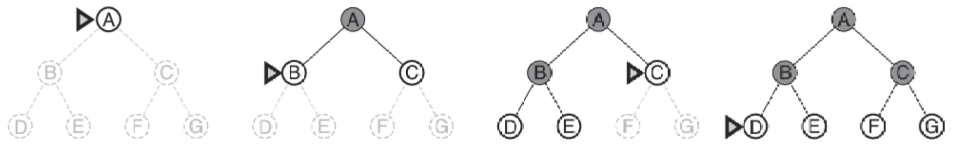
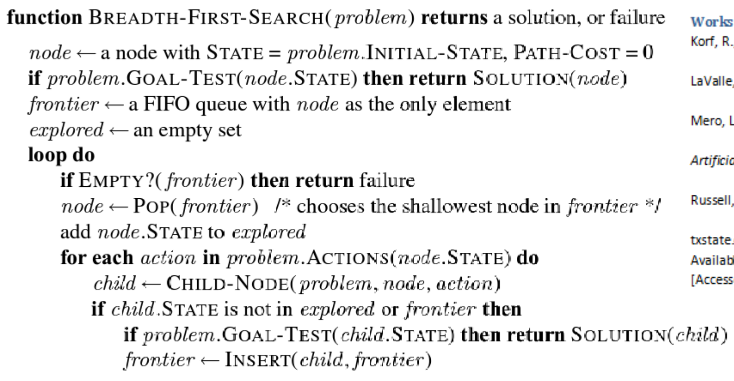


Figure 1: Breadth First search Russel, S; et al

Breadth first search was invented by Konrad Zuse in 1945, for the purpose of finding connected components of a graph. It was reinvented in 1959 to find the shortest route out of a maze.

Breadth first search works by examining all child nodes of a layer before moving onto the next layer. This is clearly illustrated in the picture above (Russell & Norvig, 2016), First A is expanded, following this B and C are considered for expansion and finally D, E, F and G will be expanded. This algorithm is optimal if all paths have the same cost and it will generated one of the shortest routes to the goal state.

There are some drawbacks to this algorithm. The first to be considered is the complexity as the number of nodes generated is an exponential function (if each node has b children) O(b^d) (Korf, 1985). This is because each new layer expanded increments d in the above formula by 1 until the required depth d is reached. This means that for complicated problems the algorithm can use up memory very quickly. As well as this time begins to become a major consideration these limitations are illustrated very clearly in a table in the book by (Russell & Norvig, 2016). They state that "In general, exponential complexity search problems cannot be solved by uninformed methods for any but the smallest instances."



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the shallowest node in frontier */
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
              frontier ← INSERT(child, frontier)
```

Figure 2: Breadth First Search Pseudocode (Russel, s; et al)

Works Cited

Korf, R., 1985. Depth-First. *Artificial intelligence,* Volume 27, pp. 97-109.

LaValle, S., 2006. *Planning Algorithms.* 1st ed. Cambridge: Cambridge University Press.

Mero, L., 1984. A Heuristic Search Algorithm with Modifiable Estimate.

Artificial Intelligence, Volume 23, pp. 13-27.

Russell, S. & Norvig, P., 2016. *Artificial Intelligence a modern approach.* 3rd ed. Harlow: Pearson.

txstate.edu, n.d. *txstate.edu.* [Online]
Available at: https://userweb.cs.txstate.edu/~ma04/files/CS5346/SMA%20search.pdf
[Accessed 18 11 2018].

A*

A* was first published in 1968 and was developed as part of the Shakey Project at the Artificial Intelligence Centre of the Stanford Research Institute. Initially only a heuristic function was going to be used but it was decided that Dijkstra's algorithm should be combined with this, resulting in the development of A*.

Figure 3 is taken from artificial intelligence a modern approach, and it shows the scoring process of A* path finding from Arad to Bucharest. The score is calculated by adding the distance travelled to the node being considered and a heuristic value of the distance between that node to the target.

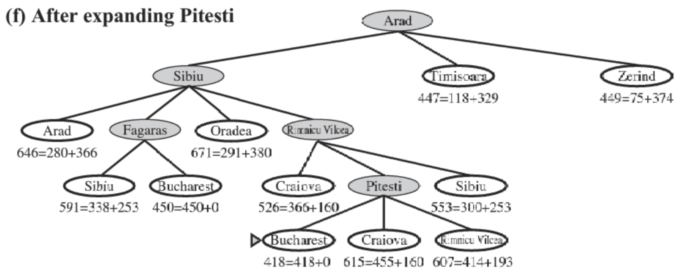
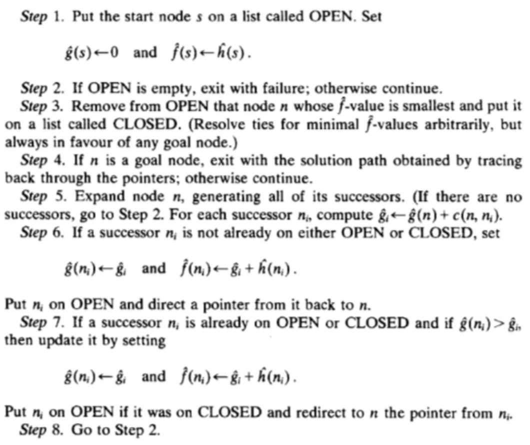


Figure 3: A* search Russell, S; et al

This heuristic must adhere to certain criteria to ensure that an optimal result will be found. The first is that it must be admissible (Russell & Norvig, 2016), meaning that it must always underestimate the cost of reaching the target while being as close as possible to the actual cost. It must also conform to consistency, in that the heuristic value of a parent node must be less than or equal to the cost of travelling from the parent to child node plus the heuristic cost of the child. However, it has been stated that it can be difficult or even impossible "...to find a heuristic that is both efficient to evaluate and provides good search guidance." (LaValle, 2006).

Figure 4 on the right (Mero, 1984) shows the A* algorithm. The algorithm works by first placing the start node in the open list. So long as the open list is not empty the algorithm will continue. The score for all nodes in the open list is calculated as described above, and the lowest scoring node is chosen and placed in the closed list. If this node is the target node the solution has been found. Otherwise, Each of this node's children are then considered. If it does not appear in either list then it is added to the open list. If it does appear on the open list but this new score is better, then update its parent and score. This version of A* includes a case where the child node is found in the closed list with its new score being lower.



Step 1. Put the start node s on a list called OPEN. Set

$$\hat{g}(s) \leftarrow 0 \quad \text{and} \quad \hat{f}(s) \leftarrow \hat{h}(s).$$

Step 2. If OPEN is empty, exit with failure; otherwise continue.

Step 3. Remove from OPEN that node n whose $\hat{f}$-value is smallest and put it on a list called CLOSED. (Resolve ties for minimal $\hat{f}$-values arbitrarily, but always in favour of any goal node.)

Step 4. If n is a goal node, exit with the solution path obtained by tracing back through the pointers; otherwise continue.

Step 5. Expand node n, generating all of its successors. (If there are no successors, go to Step 2. For each successor n_i, compute $\hat{g}_i \leftarrow \hat{g}(n) + c(n, n_i)$.

Step 6. If a successor n_i is not already on either OPEN or CLOSED, set

$$\hat{g}(n_i) \leftarrow \hat{g}_i \quad \text{and} \quad \hat{f}(n_i) \leftarrow \hat{g}_i + \hat{h}(n_i).$$

Put n_i on OPEN and direct a pointer from it back to n.

Step 7. If a successor n_i is already on OPEN or CLOSED and if $\hat{g}(n_i) > \hat{g}_i$, then update it by setting

$$\hat{g}(n_i) \leftarrow \hat{g}_i \quad \text{and} \quad \hat{f}(n_i) \leftarrow \hat{g}_i + \hat{h}(n_i).$$

Put n_i on OPEN if it was on CLOSED and redirect to n the pointer from n_i.

Step 8. Go to Step 2.

Figure 4: A* algorithm Mero, Laszlo

In figure 3 Bucharest appears twice but the shortest path is chosen as the generated score is lower even though that node is deeper than the other Bucharest node. The real power of this algorithm over Dijkstra's algorithm is that the directionality imparted by the heuristic value reduces the number of nodes that are visited. A* can also be modified in a variety of ways depending on what the user requires the algorithm to do, for example it can be modified to run in an even faster yet less accurate way if desired.

SMA* (Simple Memory Bounded A*)

Simple memory bounded A* is a development of A* that uses available memory to carry out its search. It is arguably better than IDA*, which doesn't remember the current shortest path or costs for visited nodes, instead storing one path at a time. SMA* by contrast functions in a similar way to A* but with an additional subtlety, it will store nodes until available memory is full. After this it will then start deleting the shallowest highest scoring node before adding the deepest and lowest scoring node. In effect it "...expands the newest best leaf and deletes the oldest worse leaf" (Russell & Norvig, 2016). SMA* is "...optimal if enough memory is available to store the shallowest optimal solution path." (txstate.edu, n.d.).

Figure 5 shows an example of SMA* being applied with a maximum memory of three nodes. A the start node is added, then it's first child node (B) is added when G is added all of A's children have been examined. A is then updated with the lowest score of the two child nodes(13). The next node to be examined is G as it has the lowest score. As the memory is full a leaf needs to be deleted. B's score is backed up to A and H is examined next. H's score is calculated to be 18 but as it is not a goal node and is at the maximum depth memory will allow its score is set to infinity. H's score is then backed up to G, H is deleted and I is considered. I's score is calculated as 24, as this is as far as the memory constraint will allow us to go down the G branch G is updated with I's score of 24. Following this B is added again. B is chosen with the lower score and G is backed up to A then deleted. C is then considered given a score of infinity as it is at max depth and not a goal. It is backed up then deleted to allow D to be considered. D is calculated to have a score of 20 which B is updated with. Two goals have been found of which D is selected as it has the lowest score.

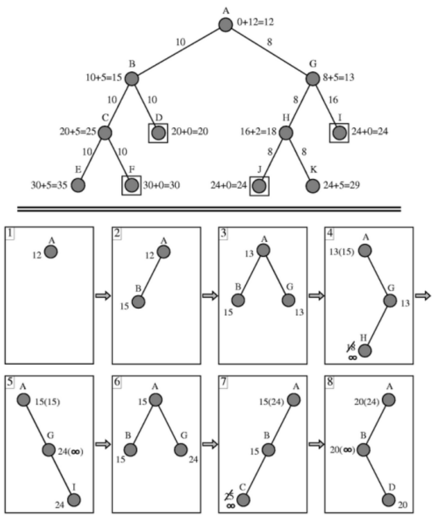
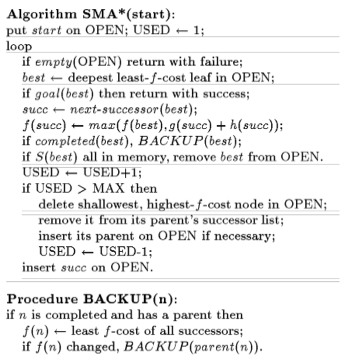


Figure 5: SMA* Example http://www.massey.ac.nz/~mjjohnso/notes/59302/l04.html

Most of SMA*'s limitations are directly tied to the available memory. It will find a solution provided there is enough memory to reach the required depth where the solution can be found. The same is true of the optimal solution, it must have the required memory to be able to find it. If there is not sufficient memory to reach the optimal solution it will return the best solution it can get to with the memory available.



```
Algorithm SMA*(start):
put start on OPEN; USED ← 1;
loop
    if empty(OPEN) return with failure;
    best ← deepest least-f-cost leaf in OPEN;
    if goal(best) then return with success;
    succ ← next-successor(best);
    f(succ) ← max(f(best), g(succ) + h(succ));
    if completed(best), BACKUP(best);
    if S(best) all in memory, remove best from OPEN.
    USED ← USED+1;
    if USED > MAX then
        delete shallowest, highest-f-cost node in OPEN;
        remove it from its parent's successor list;
        insert its parent on OPEN if necessary;
        USED ← USED-1;
    insert succ on OPEN.

Procedure BACKUP(n):
if n is completed and has a parent then
    f(n) ← least f-cost of all successors;
    if f(n) changed, BACKUP(parent(n)).
```

Figure 6: SMA* algorithm Russell, S

SMA* does have another significant problem to be considered which is that if the problem is complicated enough it will switch back and forth between branches which will cause the algorithm to continuously delete and regenerate nodes due to its memory limit. An important consideration of this is that if A* were given enough memory it would solve the problem more efficiently than SMA* due to the extra time required by SMA* to delete and recreate nodes.

Search Methods Applied to Cave Route Problem

Breadth First search

If the problem was simple enough it might be possible that breadth first would be sufficient. However, as it is stated in the coursework brief that the distance between nodes can be calculated using Euclidian distance there is an implication that the nodes aren't all a uniform distance apart which means that breadth first search will not return the optimal path.

Another consideration to take into account is that if the complexity gets too high then the time to run the algorithm will also increase. As there is a strict time limit of one minute for the code to run before the program is stopped, this increased amount of time to find the solution for problems of increasing complexity is a major concern. As the complexity is given by the average number of branches on a node (b) to the power of the depth at which the goal node is found (d), if the problem contained 1000 caves it could be possible that both b and d are quite high numbers.

SMA*

SMA* would be an improvement on breadth first search as it would be able to deal with the fact that the caves are different distances apart. This is due to the fact that it is a modified version of A*. Therefore it can handle both differing distances, and also be "guided" by a heuristic.

However, like breadth first search it is possible for this algorithm to run into problems with problems of increasing complexity if it doesn't have enough memory. The problem presented in the coursework brief carefully considered and the increased complexity in implementing A* was deemed to be too much given that there would probably be enough memory available to implement A* with regard to this problem.

A*

A* was chosen as the algorithm to be implemented. It can handle differing distances between caves, and in addition it is easy to calculate an acceptable heuristic as the cave locations are given in Cartesian coordinates. Hence, the heuristic for a given child node is the Euclidian distance between the child node being considered and the target node. As the distances between nodes is trivial to calculate using Euclidian distance it would be possible to implement Dijkstra's algorithm, however, the increased efficiency of A* was chosen as it is not a great deal more complicated to implement when compared with Dijkstra.

In addition to this A* can handle complicated cases of the given problem without the requirement of modifying the code to be able to handle these cases. A* should return the most efficient route every time. Aside from the increased efficiency A* offers over breadth first search, the fact it will find the optimal route without any need for modification is another clear advantage A* has over breadth first search. Finally, A* is a commonly implemented solution to path finding problems. Therefore, A* was chosen as the preferred method to solve this problem.