Christopher Johnson - 40275286

Artificial Intelligence Coursework Report

For this coursework I needed to use an appropriate algorithm to find the shortest possible path between multiple nodes, in this case caverns, conjoined by either one or two-way paths (caves). I decided to use the A* algorithm, programmed in Python. Two other algorithms that I considered for the problem were Dijkstra's algorithm and the Greedy algorithm. In this report I will describe these methods as well as the one I used and evaluate each of their performance as solutions to search problems.

Method Descriptions

Dijkstra's Algorithm

One of the algorithms for finding the shortest path from a start node to an end node in a scenario like this one is Dijkstra's algorithm. The algorithm, published in 1959, is named after the Dutch computer scientist Edsger Dijkstra who was the algorithm's creator.

The algorithm functions as follows:

1. Create a new list of all the nodes, this list will include every node that has not yet been visited. Set the provisional distance to each node to infinite.
2. Retrieve the starting node and make that the current node.
3. For each of the nodes neighbours calculate the distance to get to the neighbour. If this distance combined with the current node's provisional distance is lower than the neighbour's current provisional distance, make it the node's new provisional distance.
4. Mark the current node as visited and remove it from the list of nodes that have not yet been visited.
5. If the end node is marked as visited, the algorithm is complete and has found the shortest route from the start node to the end node successfully. If there is not a connection between the start and end node then the smallest provisional distance in the left over unvisited nodes will be infinite. If either of these events occur, then stop.
6. Otherwise, set the current node to the node with the shortest provisional distance and return to step 3.

Dijkstra's will always return the shortest possible path for this scenario, assuming a valid path exists between the start and end nodes. It is not as efficient, however, as the A* algorithm.

Best First Search (Informed Search)

A best first search algorithm is an algorithm that takes the first available shortest path from the starting, and each subsequent node. With a greedy best first search algorithm the node

with the shortest heuristic distance from the goal node will be explored next. This distance is calculated as the Euclidean distance between the current node and the end node.

1. For each node work out and assign itself it's distance to the end node.
2. Create an empty open list and an empty closed list
3. Add the start node to the open list.
4. Set the current node to the neighbour of the current node with minimum heuristic distance and set the current node as its parent.
5. Remove the current node from the open list and add it to the closed list
6. If the current node is the end node then the algorithm is done, work out the path backwards by getting the ID of and then setting the current node to its parent node. When you reach the start node, reverse the list of ID's to retrieve the shortest path.
7. Otherwise for each neighbour of the current node, if the node is not already in the closed list, assign its distance to the distance of the previous node plus the distance to the end goal and add the neighbour node to the open list
8. Return to step 4. If the open list is empty, then no goal node was found.

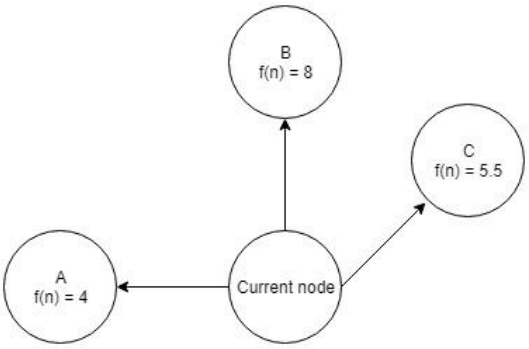

Figure 1 For best first search the next node to be examined would be A as it has the lowest distance (f(n)) to the end node

This specific type of best first search, the greedy first search rarely produces an optimal route from the start node to the end node and so I chose not to use for this problem.

A* Algorithm

I chose to use the A* algorithm to solve the cavern navigation problem because, like Dijkstra's, it will always find the shortest possible path from A to B. A* is more efficient than Dijkstra's because it has an additional heuristic, a function that ranks alternatives within the algorithm based on available information, that is considered.

Dijkstra's only considers the distance of the current node from the starting node and ignores the distance left to travel. The A* algorithm considers the distance from the current node to the end node. As we do not know the distance, via nodes, to the end node the Euclidean distance between the current node and the end node is used. This is the straight-line distance and is calculated as follows:

$$\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

This gives the A* algorithm an advantage as nodes that are further away from the end node are given a higher weight and will not be expanded until later, favouring nodes that are closer, and therefore show a higher chance of being in the right direction.

The algorithm functions as follows:

1. Create a closed list (for nodes already evaluated) and an open list (for nodes that have been discovered but have not been evaluated).
2. Get the starting node, the only discovered node at this point and add it to the closed list
3. If the current node is the end node then the algorithm is done, work out the path backwards by getting the ID of and then setting the current node to its parent node. When you reach the start node, reverse the list of ID's to retrieve the shortest path.
4. Find the node's neighbours and for each neighbour calculate the following values the node has, as well as setting its parent to the current node:
    - g = the distance between the current node and the neighbour node
    - h = the Euclidean distance between the neighbour node and the end node
    - f = g + h
5. If the neighbour is not already in the closed list and the same node does not exist in the open list with a lower value of g then add the neighbour to the open list
6. Set the current node to the node in open list with the lowest f value and return to step 3.

This algorithm was published in 1968 and is often seen as an extension of Dijkstra's algorithm. A* was created as part of a project that had the aim of building a robot capable of planning its own actions and is now commonly used for path finding in video games.

Method Evaluations

Dijkstra's Algorithm

This method of path finding will always produce the shortest possible path, where available. This makes it a solid choice for path finding problems. The A* algorithm is very similar in nature but far more efficient and so I chose A* over Dijkstra's.

Best first Search Algorithm

The best first search algorithm allows us to switch between paths by gaining the benefits of both breadth first and depth first search. This is because the solution can be found without examining all the nodes in the graph, but the algorithm also will not get trapped at dead ends. This means it would have been possible to solve the cavern traversing problem with this method but can, however, cover more distance than necessary which means it will not always output the shortest path possible.

If this problem took place on a grid then best first search would be the most suitable algorithm as on a grid if you move closer to the end node you are never going to back track, however in the cave model used caves are not evenly spaced out.

A* Algorithm

The A* algorithm combines the consistency of Dijkstra's with the greedy nature of the Greedy algorithm. Using the additional heuristic function, it acts as a more informed version of Dijkstra's and, in the context of this problem, makes a better choice for a path finding algorithm because of this.