COMP 3106 Assignment 1 Technical Document
Oct 18 2021

Contribution:
Christopher Krol 101010495
Ian Clark 100803045
1) Yes each member of the group made significant contributions.

2)Contributions were to different parts mostly but for the most part fairly evenly split.

3)Ian Clark: Did almost all of the coding, did question 1 of technical and planned much of the heuristic and process.

Christopher Krol: Most of the technical document, helping with heuristic and formatting as well as minor changes and improvements to the code

1.
Our implementation creates a new class called "node" which stores data on the node (x,y, parent, path_cost, and label[O,X,H,S, or G])
After reading in the lines of the file and formatting it (removing any blank lines), we create a node for each object in the input and add it to a list.
We then iterate through the list and convert any hazards and their adjacent tiles to obstacles. To do this we use the neighbourhood() function which returns all neighbours adjacent to the tile.
Our neighbourhood function works by calling the heuristic() function with the current tile as the goal node, and finding any tiles that return a h(n) <= 1, which returns only the adjacent tiles. We skip any obstacle nodes (which now includes hazards and adjacent tiles).
We then find the start and goal node, by iterating through our list, and storing them in a variable.
We iterate through the list and calculate the heuristic for each node, storing it in a property on the node (node.h)
Our heuristic function calculates the manhattan distance between two given nodes.
We then call the graph_search() function, which uses code similar to what was in the lecture for A*.
The only difference is that after a node gets added to the frontier, the frontier is then sorted according to a key function: f(n) = node.path_cost + h(n), and reversed. Since our list pops off the end, we want the lowest f(n) to be at the end of the list, hence the reversal.
When our goal state is popped off the stack, we print out the Cost, Explored List, and the Optimal Path found.
Our path is a list returned by the path() function, which loops through each nodes parent, starting from the goal state.

2. We have implemented a goal-based agent where the goal is to reach the goal node.

3. Since every move is either left or right or up or down and that means that in order to get from the start to the goal the agent needs to take a number of moves along the the x and/or y axis in order to get there. So we can choose to use the Manhattan-distance as the heuristic:

```
function heuristic(node, goal)
    D = 1  # cost to move
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy)
```

We know that this heuristic is consistent since the shortest way to get to from a to b is the number of moves that you need to go horizontally plus the number of moves that you need to go vertically or simply put x + y this is exactly what the heuristic calculates. So in the best case scenario it is perfect and if you start to add other factors such as hazards and obstacles then it only increases the difference between the actual cost and the heuristic cost. This heuristic is dependent on 2 key factors both of which are met in this situation the first that all moves cost the same and second that only vertical and horizontal movement is allowed.

an alternative heuristic that could be used but that is less informed would be to take the exact distance between the node and the goal:

```
function heuristic(node, goal)
    D = 1  # cost to move
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * sqrt((dx * dx) + (dy * dy))
```

While this heuristic is also consistent it is less informed than the Manhattan-Distance and that is because the actual agent is unable to move in any direction except vertically and horizontally. Manhattan-distance takes that into account by finding the minimum distance for a path moving only vertically and horizontally, whereas the exact distance cuts corners. This means that the Manhattan-distance will always be as big or bigger than the exact distance anad because the Manhattan distance is consitent it means that it is also more informed.

4. Given that movement costs are uniform it is possible that in any situation Uniform cost search could find the optimal path in the most efficient way, so I will provide an example of when it could happen. An example of when this might happen when we use this example input

```
O,S,O
O,X,O
O,X,O
H,O,G
```

with this input A* would explore [(0,1),(0,0),(0,2),(1,2),(2,2),(3,2)] it would never explore (1,1). uniform cost search on the otherhand would very possibly visit (1,1) making it a slower search method.

5.
One example of when greedy heuristic search would not find the optimal soloution is example input 3 (I didn't copy it here because the input is very long) since our heuristic is the Manhattan-distance from the node to the goal node greedy heuristic search would take the path that goes down from the start node and would continue on that path all the way to the goal node at which point it would stop. it would achieve a path length of 38 where as the optimal is 32.

6. What strategy should be used to break ties when two nodes on the frontier have equal priority function? [3 marks]
A strategy that could be used in order to break ties would be to choose the node (among those that tied) with the lowest heuristic value. The reason why this would be a good strategy is that the one with the lower heuristic would be more likely to be closer to the goal node. This is all dependent on you having chosen a good heuristic.

7.
If the agent was able to make diagonal moves then the Manhattan-distance would no longer be consistent and would not be a option as a heuristic. So in order to solve this problem we will use the Manhattan distance as the basis and work from there. Since whenever you move diagonally you are saving a move (you are essentially moving vertically and horizontally at the

same time) we can use this to achieve our goal by going diagonal towards the goal from the start until we are in line with it and then going straight at it from there. this would leave us with this function:

```
let D = cost to move
function heuristic(node)
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * ((dx + dy) - (dx or dy))
```

it gets problematic at the end because our equation doesn't know whether to use dx or dy since we don't know what is left over once it gets in line with the goal node. This however can be infered because whichever is bigger dx or dy is going to be the one that is further from the goal and therefore the one that we will still need to traverse once we are directly in line with the goal node. So we can change the return statement to look like this:

```
return D * ((dx + dy) - min(dx, dy))
```

this can be further simplified to:

```
return D * max(dx, dy)
```

this is only possible in this scenario because the cost of moving diagonally is the same as moving horizontally or vertically.