Technical Document – Assignment 1

Fareha Sultan – 100968491

Please answer the following questions in the technical document. Explain why your answers are correct.

1. Briefly describe how your implementation works. Include information on any important algorithms, design decisions, data structures, etc. used in your implementation. [10 marks]

Converting the given input to a graph with coordinates

My implementation of A* search in Python follows the pseudo-code from class. It starts off by taking in a m x n grid which is saved into a results[] array where each line is a different row. From there, in order to avoid multiple looping later on, I decided to save results[] into a graph[] array where each letter will represent a x-y coordinate as a tuple. As the coordinates are being saved into graph[] , I also do some checking to see if the letter at that coordinate is the start state, a wall, a hazard, or the goal state. I create a variable for start, goal and obstacles[] array which includes walls, hazards and coordinates adjacent to hazards. To check for coordinates adjacent to hazard, I use a directions[] array which checks the coordinates north, south, east and west and then makes sure they fall within the m x n graph.

A* algorithm's choice of data structures

For the algorithm, I make use of dictionaries for frontier, parent, and cost. The frontier's key is a node 's coordinate and the value is the function f(n) = n.path_cost + h(n). For parent, the key is again a node that happens to be a neighbour of a current node that is being explored , so the value of the parent dictionary is the current node's coordinate. For cost, the key is a node which happens to be a neighbour of a current being explored and the value is the current node's cost up until now plus 1 . I also have a variable called explored [] which is just an array that stores the coordinates that have been explored and don't need revisiting.

A* Algorithm in action

It starts off by adding the start coordinate to the frontier with a value f(n) = 0 and also adds to parent dictionary with parent value being None. The algorithm runs as long as the frontier is not empty which is done using a while loop. Once inside that while loop, I have my own implementation of a priority queue because I tried using the one in the Python Library but it didn't give the correct results and did not allow to iterate over it. The way I implemented the priority queue may not be the most efficient way but it works. So, we find the smallest value in frontier and then find its key and use that key as the current node. Then check to see if current node is the goal state, if not keep going with the rest of the while loop. We delete the current node from frontier, store it in the explored[]. Next, we find all its neighbours using a function findNeighbours(curr,rows,cols), which has the same basis as we used to find the adjacent spots for hazard. Next, we make sure that the neighbour isn't in the obstacles[] nor in the explored[], because we can't traverse them nor revisit them. We find a new cost for the neighbour using the cost of the current node plus 1. Then we check , if that neighbour is already in the cost dictionary and if so, is the new cost lower than the one stored for it. This way, we make sure to store the lower of the cost for that neighbour node. The next is storing that neighbour node in the frontier using the f(n)

value. The f(n) value require the heuristic which is calculated using the function heuristic(point_a, point_b), which takes in the node and the goal state and calculates the difference in y and difference in x. I decided to use the Manhattan Distance for the heuristic instead of the Euclidean distance because we are not moving diagonally and we get a maximum of 4 neighbours. Once that is done, we keep going in the while loop until the current node is the goal node.

Reconstructing the path and cost

When we reach the goal state, we can output the explored list and the cost using the dictionary with the key goal. To reconstruct the path , I store it in a path array. I use a while loop and the parent dictionary to trace back the parents in reverse order. So, start by finding the parent of goal state and use that as the key for the next parent and so on until we reach the start state. Then, take the path[] and reverse it so that the optimal path is in order from start to goal and output it in the terminal and .txt file. Voila, we are done!

2. What type of agent have you implemented (simple reflex agent, model-based reflex agent, goal-based agent, or utility-based agent)? [3 marks]

The A* Search Algorithm is similar to the example we saw in class of a Maze Solving Agent. In this case , I have implemented a utility -based agent which has a utility function that considers the cost and the heuristic , how close we are from the current node to the goal state. A graph may have more than one way to reach the goal state, but in terms of A* we consider the shortest path that costs us the least amount of steps. It keeps track of all the previous nodes and their costs and tries to minimize the cost. The utility function also allows to measure the probability of achieving the goal state from a particular node using the heuristic. Also, in case a graph has more than one goal, which is not the case for our assignment, the utility function can be used to check which goal returns a high utility. Overall, the A* algorithm is a utility-based agent and allows us to find an optimal path.

3. What heuristic should be used for A search for this environment? Show that this heuristic is consistent. Show that this heuristic is more informed than some alternative heuristic. [8 marks]*

A consistent heuristic should be used for A* as it guarantees to find the optimal solution. This also implies that the heuristic also has to be admissible as it is consistent (proof in class notes(. An admissible heuristic never overestimates the cost to get to the goal state.

In my case, I decided to use the Manhattan distance for the heuristic as we are only moving either is a vertical direction or horizontal direction and there is no weight associated with the edges. To show that it is a consistent heuristic , I can do a small example using a grid with 4 points where (0,0) is the start State and (1,1) is the goal state. We know from class that :

A heuristic is called consistent if for all nodes n and all successors n',

$$h(n) \leq cost(n, n') + h(n')$$

and

$$h(goal) = 0$$

So h(g)= heuristic((1,1),(1,1)) from my implementation using the Manhattan distance which gives 0. Next, to prove that $h(n) \leq cost(n, n') + h(n')$, we know that each move costs 1 so if we are moving from

start to (0,1) , we get cost ((0,0),(0,1)) = 1 and the heuristic ((0,1)(goal)) = abs(0-1)+abs(1-1) = 1 which gives an f(n) of 2 and h(n) is 2. Therefore it satisfies the $h(n) \leq cost(n, n') + h(n')$.

Another heuristic we covered in class is the Euclidean distance but in our case, because we are not moving diagonally, it will not give us an accurate estimate to the goal state as we are only allowed axial movement. It may predict to get to the goal state faster than what we actually can making it less informed as it is off from the true estimate. In case of the Manhattan distance, it gives a larger value than that of Euclidean distance but it still underestimates the true path cost making it an informed heuristic. And we know that, the more informed a heuristic is, A* finds the optimal solution faster.

4. Suggest a particular instance of this problem where A search would find the optimal solution faster than uniform cost search. [4 marks]*

From class notes , we know that UCS is always going to get us a solution with the lowest cost. But It turns out that if we pick the heuristic correctly, A* search will still going to give us the optimal solution but it's going to do it faster than uniform cost search. That means that as long as heuristic does not overestimate distances and is informed. A particular instance where A* would be useful and find the optimal solution in this scenario would be if time was an issue. If we account for time in our problem, A* would find the optimal solution faster than UCS. So, A* finds the right answer, like uniform cost search does, but it will explores less than UCS costing it less time. However, it won't give the optimal solution faster if the heuristic is inconsistent and less informed.

5. Suggest a particular instance of this problem where a greedy heuristic search would not find the optimal solution. [4 marks]

If we are looking for the shortest path that costs less, then greedy search may not end up finding the optimal path. Although, greedy heuristic search explores in directions closer to goal state, but it might not find the shortest path. Although it uses less space than A*, as it does not store information about previous costs, it does not guarantee to find an optimal and complete solution. In our case, if we looking to find the optimal path , something that will cost us the least amount of steps, greedy heuristic search may not be able to do that since it's f(n) = h(n) and does not keep track of previous costs.

Search	Frontier	Finds solution?	Finds optimal solution?	Time Complexity	Space Complexity
Breadth First	FIFO	Yes	No	$O(b^d)$	$O(b^d)$
Depth First	LIFO	No	No	$O(b^m)$	O(bm)
Uniform Cost	f(n) = g(n)	Yes	Yes	$O(b^m)$	$O(b^m)$
Greedy Heuristic	f(n) = h(n)	No	No	$O(b^m)$	$O(b^m)$
A*	f(n) = g(n) + h(n)	Yes	Yes	$O(b^{*d})$	$O(b^{*d})$

(Table is from class notes)

6. What strategy should be used to break ties when two nodes on the frontier have equal priority function? [3 marks]

If we assume that our heuristic is consistent and informed, then we can use the lowest heuristic of the two priority functions as a tie breaker.

7. Consider a variant of this problem where the agent can move to adjacent diagonal squares. What would be the best heuristic to use in A search if the agent could also make such diagonal moves? [8 marks]*

In this case, having axial movement and diagonal movement, the Manhattan distance will no longer be admissible not consistent as it will overestimate diagonal costs. It will no longer satisfy this $h(n) \leq cost(n, n') + h(n')$. Therefore, in this scenario, it would be best to use Euclidean distance . We will have a maximum of 8 directions instead of 4 . Also, the implementation would have to take into account a different cost if we are moving to a neighbour that is diagonally across the current node. The new cost would be sqrt(2) which is around 1.41 . Overall, having diagonals will speed up the search and can reach the goal faster. Using the same example as I did in Question 3, for a graph with 4 nodes, it is consistent and admissible.