Sonal Unadkat CMPT 310: Assignment 1 Report Fall 2016

To complete this assignment, there were many possibilities of implementation. Search strategies and different programming structures have been able to produce the same results, however, the goal was to provide the most optimal solution. For my assignment, I decided to implement the A* search. It was not only an optimal search, but it also took into account the usage of an admissible heuristic. The way I implemented A* search was by using the Manhattan distance as a heuristic. Although there were many other possible heuristics, but because we have only 4 possible movements, the Manhattan distance seemed to be sufficient enough. Because this is an offline- navigation problem, we must make sure we have complete knowledge of our current state, which in my implementation has been implemented by using Nodes.

This program has been coded so that the user enters the initial and goal coordinates, and chooses the option of whether to use Landmarks or not. The use of landmarks will be discussed later, along with the implementation of Part 2 of the assignment. A* has been implemented using a Node class, which contains information about the own coordinates. Here, we estimate the total cost by calculating the current cost from the Initial node to the current coordinates, as well as our heuristic calculation, which will tell us how far the current coordinates are from our goal. This ensures that we are not under-estimating the cost towards our goal. Our node class also contains a coordinates of the parent node, which are the coordinates it came from. This will be useful for when we must retrace our shortest path, once we have found our goal. We also use the usage of queues and vectors to keep track of the evaluated nodes, along with the nodes that still need to be evaluated in our Frontier list.

Essentially, how this algorithm works is by first inserting our Initial node into a queue, which essentially was the frontier of "A* search tree". Until the queue empties or we find our goal, we will evaluate the next node in our "Frontier" list. We first check if we the current Node's coordinates are our goal's coordinates. If not, then we will create its adjacent nodes called neighbour_nodes, and add them to our Frontier. Each time we finish evaluating our current node, we keep it on our Closed Node queue. For every adjacent neighbour node created must be handled differently if they satisfy any of the following cases:

Case 1: Our created neighbour node is part of our inaccessible nodes (blocked) or that the coordinates of that particular node were inadmissible (out of range coordinates). In both cases, we ignore the node and do not add it to any list. This allows us to avoid any blocked/inadmissible nodes and also ensures that we are staying in the range of the given grid.

Case 2: The neighbour node's coordinates have already been evaluated by. In a case such as this one, we will check the total score of the evaluated node. If the evaluated node's total score is higher than the currently created neighbour node, then we will update the evaluated node with the lower total cost as well as update the parent of the node. This ensures

that every node is evaluated, and that we have allocated the shortest distance (total cost) to that node.

Case 3: The neighbour node does not satisfy the first two cases, therefore we add it to our Frontier List.

After evaluating all 4 possible adjacent nodes of our current node, we update our Queues accordingly, and then before taking out the next node, we will sort our Open List (Frontier) for nodes with the lowest total score. By sorting our queue for the lowest Total-Cost (estimated cost), we are truly implementing A* search. This is what makes it optimal, and therefore, we are always evaluating the node with the lowest total cost (path) to our goal. Hence, therefore, we reached an evaluated node, we will update the previously evaluated node if we have found a shorter way to reach that once previously evaluated node.

For the second part of the assignment, we use our goal and initial coordinates to find the shortest path to a landmark in our grid. Then, we use precomputed distances between landmarks to attain the shortest distance. The goal of this is to make our Search strategy faster, although, the search strategy may still not be the most optimal. This trade-off was implemented a few ways, in my program. The first part of this program is to find the shortest distance to the landmark. To do so, we must compare the distances of our coordinates to 4 landmarks. By using our A* search each time to calculate the distance towards a certain landmark, we are creating much more overhead and therefore much slower. What I decided to do, was use the same Manhattan distance- heuristics to compare the distance of our node to each of the landmarks. Although this may be not as optimal as using A* search, it is much faster to compute numbers than create A* search trees. Once we find the two landmarks, we can use pre-computed costs to find the shortest route. This, in turn, increases our speed because we are no longer looking at all possible routes to our goal, but estimating costs towards a particular landmark and adding the shortest distance between them.

Not all inputs, produce the shortest path results by going to the closest landmark. For this program, it will decipher would it be shorter to go the goal using landmarks, no landmarks, or possibly the same landmarks. The three possible cases are outlined here:

Case 1: The cost to go directly to the goal is cheaper than using landmarks. For example, if our initial coordinates are (1, 1) and goal coordinates is (2, 2). Even though the closest landmark is (5, 5), it is cheaper to go directly to the goal coordinates, rather using the landmark.

Case 2: The cost to use the same landmark is cheaper than going to two different landmarks. For example we use the case of initial coordinates of (8, 5) and goal coordinates of (13, 5). Using two landmarks, our closest landmark to the initial coordinates would be (5, 5) and for the goal landmark it would be (12, 5). It is cheaper though if our initial coordinates meet the goal coordinates at the same landmark (12, 5). Therefore, our program will choose to meet at the same landmark.

Case 3: This is where our initial and goal coordinates do not meet the above 2 cases. It will then use the two landmarks and the cost between to the two landmarks to create the shortest path.

The implementation of our algorithm, focuses on using landmarks to narrow down the shortest path, and decide what is most ideal. By using heuristics as estimated costs, and choosing our cases accordingly to heuristics, we can faster, narrow down the shortest distance to our goal. With the use of landmarks, we can choose a more direct way to get to the landmark, therefore adding less nodes to the frontier, and directly faster. In the base case testing of (0, 0) to (17, 17), using landmarks is significantly faster using landmarks. The choice to use heuristics to determine which landmark to use, is where we trade off optimally, for speed.