ADVANCED ANALYSIS OF ALGORITHMS

Assignment 1

Vhutali Tshavhungwe 1715260

01/09/2019

Task 3

2. The difference between A* and Dijkstra is that A* consists of a heuristic function whereas Dijkstra can be seen as a special case of A* with its heuristic function set to 0. There is also the fact that A* looks for the one optimal or best path from the source node to destination node but Dijkstra finds the best path from our source node to all other nodes. This implies that A* is more efficient in handling problems that focuses on reaching the destination rather than covering all the nodes. In terms of similarities, both make use of an actual cost function, this is usually the weight or distance between two nodes. Both algorithms also find a path from our source to destination if it exists.

3. Direct Proof

The algorithm takes in a list of points(arr), and an adjacency list(adjmatrix) containing the neighbours of the points. The algorithm intialises an *open* list (keeps track of unvisited points) on line 17 and a *closed* list(keeps track of visited points) on line 18. Then the algorithm moves to a *for* loop on line 20 where i ranges from 0 to arr.size()-1, if $i \geq 0$ and $i \leq$arr.size()-1 then in this loop the heuristic cost (an attribute) of the point is set to the euclidean distance between the current point and end point on line 21, and the solution attribute of the point is set to false on line 22, this is done for all the points in the input list. If $i >$arr.size()-1 the algorithm exits the loop then sets the final cost of the point at index 0 to 0 on line 26 and adds the point at index 0 to the open list on line 28.

The algorithm then intializes a dummy point (called current) to be used on line 29, it then moves onto the *while* loop on line 31 and checks the conditon, there are two cases to consider (1a) the *open* list is not empty and (1b) the *open* list is empty (there are no points to evaluate). If case (1a) is true, we enter the while loop and current is set to the point at index 0 (line 32) in the *open* list (the value is also removed from the *open* list in this step) and algorithm then adds the current point to the closed list on line 34. The algorithm then moves onto an *if* statement on line 36 and checks the condition, there are two cases to consider, (2a) current is not equal to the point at index 1 of the input list or (2b) if current is equal to the point at index 1 in our input list (the end point is always placed at index 1). If case (2a) is true therefore we do not enter the *if* statement and the algorithm moves on to line 41 and int q is set to the index of our current point(its position in our input list).

On the line 42 there is a *for* loop where int w ranges from 0 to adjmatrix.length-1 (where adjmatrix is our adjacency matrix), if $w \geq 0$ and $w \leq$adjmatrix.length-1 we enter the *for* loop, on line 43 we initialise and set Point d to null. We then have an *if* statement on line 44, the algorithm checks the condition, there are two cases (3a) if adjmatrix[q][w] == 1(means current point in row q has a neighbour at index/column w in adjacency matrix) or (3b) if adjmatrix[q][w] != 1 (means current point in row q does not have a neighbour at index/column w in adjacency matrix). If case (3a) is true point d is set to point at index w in input list, if (3b) is true the algorithm does not enter the *if* statement and the algorithm continues.

The algorithm moves onto line 49 where there is an *if* statement and checks the condition, there are two cases to consider (4a) if d is null and 4(b) if d is not null. If (4a) is true this means the *for* loop on line 42 continues, but if (4b) is true the algorithm enters the *if* statement and current and d are parsed into the *update* function on line 59, in this function on line 60 the algorithm sets Point temp to current. On line 61 there is an *if* statement and the algorithm checks the condition, there are two cases to consider (5a) if d is null and (5b) if d is not null. If (5a) is true the algortihm returns and continues in the astarAlg function on line 42, if (5b) is true the algorithm continues to line 64 where totalhCost is set to current's heuristic cost,. The algorithm then moves onto line 65 where there is a *while* loop and the condition is checked, there are two cases to consider, (6a) if current.parent = null (at starting point) or (6b) if current.parent != null. If (6a) is true we do not enter the loop and the algorithm continues to line 72, if (6b) is true we enter the loop, on line 66 totalhCost is incremented by current.parent.heuristicCost and on line 67 current is updated to current.parent (back tracking to get total heuristic cost). On line 72 there is an *if* statement and the condition is checked, there are four cases

to consider, (7a) if d !=null and !closed.contains(d) (if d is not null and d has not been visited) or (7b) if d==null or !closed.contains(d) (d is null or d has not been visited) or (7c) if d==null or closed.contains(d) (d is null or d has been visited) or (7d) if d !=null or closed.contains(d) (d is not null or d has been visited). If case (7b) or (7c) or (7d) is true we enter the *if* statement and return on line 73 (back to line 42 loop), if case (7a) is true we do not enter the if statement and the algorithm continues to line 76 where FinalCost is set to the sum of totalhCost, d.heuristicCost and the manhattan distance between d and temp.

The algorithm moves to line 77 where notVisited is set to true or false depending on whether *open* list contains d. On line 79 there is another *if* statement and the condition is checked, there are four cases to consider (8a) if notVisited and FinalCost>d.finalCost or (8b) if !notVisited or FinalCost>d.finalCost or (8c) if !notVisited or FinalCost<d.finalCost or (8d) if notVisited or FinalCost<d.finalCost. If (8a) is true we do not enter the body of the *if* statement and algortihm returns to line 42. If (8b), (8c) or (8d) is true the algorithm enters the *if* statement body and on line 80 d.finalCost is updated to FinalCost, on line 81 d.parent is updated to temp. On line 83 there is an *if* statement, there are two cases to consider (9a) if notVisited is false(open list does not contain d) or (9b) if notVisited is true (open list contains d). If case (9a) is true the algorithm enters the *if* statement body and d is added to the *open* list on line 84, if case (9b) is true the algorithm does not enter the *if* statement and the algorithm returns to line 42. If w>adjmatrix.length-1 on line 42 the algorithm continues abd the *while* loop condition on line 31 is checked again. If case (1b) or (2b) is true the algorithm ends the astarAlg function.

The algorithm returns a correct answer, this answer is provided by the solution function on line 89, it keeps a path list and first checks whether the *closed* list contains the end point, if it does not the function outputs -1, this is a correct answer. If it does contain the end point we set a temporary variable (current) to the end point and add its index to our path list and its solution variable is set to true. Next there is a *while* loop on line 98 and if the condition where current.parent != null we add the index of current.parent to our path list on line 99, on line 100 current.parent.solution is set to true and current is updated to current.parent on line 101. If current.parent=null the algorithm moves to line 107 where it reverses the order of the path list, it then returns the path list on line 108. This is a correct answer.

4. Best case and Worst case:
   The complexity of the algorithm depends on the number of vertices we have in our graph (list) and how many edges we have to traverse to get there. Our heuristic function also plays a major role in the complexity as it helps reduce the number of points we expand on. Therefore the complexity is exponential as it depends on how many points in the graph we traverse too and branch off to as we expand all the neighbouring points. $O(V + |E|)$ where V is the number of vertices and E is the number of edges traversed which is exponential.
   Average case:
   This is equal to the best and worst case since we always have to expand the neighbours of a point regardless of where the end point may be.

5. From the included graphs it is observable that the graphs exhibit the behaviour of an exponential curve, multiplied by a constant theta where theta is greater than 0 but less than 1 shifted up by a constant lamda.Therefore as the number of sample points increases so does the number of edges traversed which becomes exponential. The empirical analysis agrees with our theoretical analysis.