AI - Assignment 1- Part 1

maya.barathy

February 2022

1 Introduction

a) Create an interface so as to create and visualize the 50 eight-neighbor grids you are going to use for the experiments. Your software should also be able to visualize: the start and the goal location, the path computed by an A*-family algorithm. Visualize the values h, g and f computed by A*-family algorithms on each cell (e.g., after selecting with the mouse a specific cell, or after using the keyboard to specify which cell's information to display). Use the images in this report from the traces of algorithms as a guide on how to design your visualization. Highlight in your report your visualization, its capabilities and what implemented for it.

The steps that we followed to set up the environment are (i)Open terminal in bash environment
(ii)Enter the following command: export PATH = "$PATH$: $/koko/system/anaconda/bin$"
$(iii) Enter the following command : source activate python 38$
$(iv) Navigate to directory containing file project.py$
$(v) To run the file, enter the following command : python 3 assignment_1.pytest/a$

In order to properly demonstrate the correct path points, we implemented a grid-like graphical user interface that was able to display angles and straight passes through vertices in all four general directions. We utilized the matplotlib and numpy libraries to design a interactive grid display using simple plot lines as node vertices and lines connecting them each. Additionally, i implemented a path order animation to demonstrate correct nodes visited. This is applicable for each algorithm.

b) (i) Manually compute and show a shortest grid path and and a shortest any angle path for the example search problem from figure 7.

Shortest grid path = (1 + 1.4 + 1.4) = 3.8 units
Shortest any angle path = (1.4 + 2.23) = 3.63 units

(ii) Manually compute and show traces of A* with the h-values from Equation 1.

Using A* the shortest path is from A4 to B3 to C2 to C1 A4 - g(n) = 0, h(n) = 3.8, A4 = 0 + 3.8 = 3.8 B3 - g(n) = 1.4, h(n) = 2.4, B3 = 1.4 + 2.4 = 3.8 C2 - g(n) = 1, h(n) = 2, C2 = 1 + 2 = 3 C1 - g(n) 3.8, h(n) = 0, C1 = 0 + 3.8 = 3.8

The shortest path using A* = 3.8 + 3.8 + 3 + 3.8 = 14.4 units

(iii) Manually compute and show traces of Theta*.
Using Theta* the shortest path is from A4 to B3 to C1

A4 - g(n) = 0, h(n) = 1.4, A4 = 1.4

B3 - g(n) = 1.4 h(n) = 2.24, B3 = 3.64

C1 - g(n) = 2.24, h(n) = 0, C1 = 2.24

The shortest path using Theta* = 1.4 + 3.64 + 2.24 = 7.28 units

c) Implement the A* algorithm for a given start and goal location for the grid environments. Describe in your report what you had to implement in order to have the A* algorithm working.
The first step in implementing our A* algorithm was creating an object to hold each vertex's coordinate points, their calculated f(v), g(v), and h(v) values, their neighbors (which need to account for blocked neighbor paths), and their parent vertex in the path. Through the implementation of A*, we also noticed the need for certain boolean flags (notAdded and newNeighbors) to keep track of certain information I'll address later. After reading the input from the text files, such as start node, goal node, and grid size, we created a 2D matrix holding all vertex nodes in the grid with default information other than their coordinate point. The next step was making a similar but more simple matrix holding 0s and 1s indicating whether or not an entire cell was blocked. With all input loaded into their respective data structures, we moved onto helper functions. The functions to generate the g(v) and h(v) values were fairly straightforward, so that g(v) would equal the g value of a node's parent (starting at 0 for the start node) + the cost between that node and its parent (being either 1 or sqrt(2)) and then using the given heuristic formula of the "Manhattan distance" between a node and the goal. The trickiest helper function was dd neighbors, wherein we added the neighbors of a given node once we reached it (as to ensure no unnecessary work was done) while accounting for blocked cells, which affect different nodes differently based on their orientation in the grid. We ended up using a index arithmetic similar to that in line of sight for Theta* to make sure only valid neighbors were added to a specific node's neighbor array. Then, following the structure of the pseudocode we added the start node to an open list priority queue, flagged its notAdded flag as False, popped it off to added to the final closed list of nodes in the path, added its neighbors to the open list, flagged those neighbors as added, and continued the process as the next node was popped off

into the closed list. We then added a second flag called newNeighbors that was default as False and would be made True once it added its first new, unadded neighbor to the open list. If the flag remained False after searching all the neighbors, it would not be added to the final path list as it reached a deadend / did not contribute any new neighbors to the path. This now refined version of the closed list became our shortest path from point A to point B using grid distances. The time complexity of A* depends on the heuristic values. The worst case is that the search algorithm needs to expand all the nodes which means that the big O for A* is O(bd) where b is the maximum branching factor and d is the depth of the shallowest goal state.The space complexity of A* is also O(bd).

d) Describe in your report what you had to implement in order to have the Theta* algorithm working.
The implementation of Theta* was similar to A* in that the same main and helper functions and node object were used. The main difference between the two is that Theta* used a new heuristic function (simply just the good, old distance formula) and the line of sight function to find if there was a shorter path using a different line of sight angle (other than 45 or 90 degrees). Changing the heuristic implementation was simple as we just added a command line flag to indicate whether the current path was running A* or Theta*, so the correct corresponding heuristic function could be used. Again, the loop implementation of popping off a node from the priority queue and searching its neighbors remained almost nearly the same, except if the line of sight function returned True (if it was False, the implementation for that neighbor at least in calculating g(v) was the same as A*). The line of sight function also used the index arithmetic to calculate whether the new path at a non-45 or non-90 degree angle would be blocked. If it was unblocked the node's parent would not be the direct neighbor node but rather its previous neighbor's parent. This created a closed list consisting of a similar if not the same path as A*, except the node's parents were not the same; they all shared a few common ancestors that could be directly jumped to by skipping the in between ones. So, to scrub through this closed list, we simply went from start node to goal node while only adding the common parent nodes (to skip the middleman). This created a final shortest distance path using any angle. The space and time complexity for Theta* is $O(b^d)$.

e) Give a proof (=concise but rigorous argument) why A* with the h-values from equation 1 is guaranteed to find the shortest grid paths.

When using A* we use the formula f(n) = g(n)+ h(n) to compute the shortest distance where g(n) is the distance from the initial state to node n which we can manually compute using addition and h(n) is the distance from the current node to the final state. The h values that we have obtained from Equation 1 provide us with the shortest path because it is admissible which means that it never overestimates the distance it will take to reach the goal state. Hence it is also optimally efficient and cost efficient. Another reason why A* is efficient is because it prunes away search tree nodes that do not help to find an optimal solution. A* will always expand all the nodes on the optimal path before expanding all the non opti-

mal paths.