Assignment 1

Roslan Arslanouk, Daniel Fraser, Arkady Shapir

February 27, 2018

Part 0

In this part, we generated fifty maze/corridor-like structures for testing our program. In order to generate the mazes, we used numpy.random.choice function in Python. Using this function, we created fifty int 2-D arrays of size 101x101, where 0 stands for unblocked and 1 for blocked, and placed the agent in the top-left corner and target at the bottom-right corner. We used 30 percent probability to block a cell. As a result, some of the mazes are not solvable since the path from top-left corner ([0,0]) to bottom-right corner ([101,101]) would be blocked.

Part 1

a)

In the example search problem (figure 8), initially the agent assumes that the space (maze) does not have any obstacles (blocked cell) tries to find the shortest path to the target cell. Therefore, since the target cell is two blocks away to the east of the agent, it would keep going to the east while observing its surrounding till it reaches a blocked cell in front of it, in which case the agent would run the search algorithm again taking in consideration the obstacles the agent saw on the way to the current cell. This process would be repeated as many times as needed till either the agent gets to the target or till it tries all possible paths and reaches the conclusion that all the possible paths to the target are blocked.

b)

In the beginning, we first check the neighboring cells of agent's location. If the neighboring cell is blocked, then it is marked as blocked. Otherwise, we check if the neighboring cells are neither in the open list or closed list, if they are in neither list, we add all of them to the open list (queue). Then we choose the cell from the top of the queue and it is added to the closed list, so it will not be visited again. After that, we check its neighbors and repeat the process again. Hence every time we would reach a dead end (blocked path), we would not go down that path again since it would be already in the closed list. Therefore, if

the number of cells in the maze are finite, then it would take a finite time to either find a path to the target or report that all the paths to it are blocked.

Part 2

In this part, we are considering two cases when breaking the ties between cells with same smallest f values: in favor of cells with smallest g values or in favor of cells with greater g value. After implementing and trying both of the methods, we have observed that breaking the tie in favor of the cells with greater g values is much faster than the other one. The reason for it is that g value represent how far the agent has traveled so far. Hence, when choosing the cell with the greater g value, we are choosing the cell that is already closer to the target cell since we always travel in the direction toward the target cell, unless the path in the direction to it is blocked in which case we go "around" the blocked cell. When choosing the smallest g value, the algorithm looks more similar to BFS (Breadth First Search), which would take longer time to get from agents location to the target since the agent would be checking in multiple possible shortest paths simultaneously rather then checking just one, and most likely finding the target from it. Therefore, breaking the ties in f values is better in favor of the cells with the greater g values since we would be expanding less cells hence reaching the target cell faster.

Part 3

In this part, we are looking at the difference between Repeated Forward A* vs Repeated Backward A*. After implementing both of the algorithm and testing them out, we have reached the conclusion that their run time is a little different and the Repeated Backward A* is slower. The reason for it is that the the Repeated Backward A* has less knowledge due to the agent being closer to the start, so the Repeated Forward A* has to restart less while the Backward A* does not notice the obstacles till nearer the start.

Part 4

A heuristic function is called consistent, if the following holds true: $\forall(n, a, n') : h(n) \leq c(n, a, n') + h(n')$, where $c(n, a, n')$ is step cost for going from n to n' using action a. The project argues that "the Manhattan distances are consistent in grid-worlds in which the agent can move only in the four main compass directions." In order to prove this, we will consider two cases. The first case is, the more obvious one, if the cell n is closer to the target cell than the cell n'. In this case, the $h(n) \leq h(n') \implies h(n) \leq h(n') + c(n, a, n')$ since it is located closer to the target cell. The second case is if the cell n' is located somewhere between the cell n and the target cell. In this case, $h(n') \leq h(n)$ since it is closer to target cell. However, since $c(n, a, n')$ is the cost for going from n to $n' \implies h(n) \leq h(n') + c(n, a, n')$ because after getting from n to

n', the rest of the path to the target cell cannot be more than $h(n')$. Therefore, the Manhattan distances are consistent.□

Furthermore, because every time the Adaptive A* algorithm is ran and $h_{new}(s)$ is calculated, it updates the $h(s)$ not only of the cell where the agent is located, but for all of the cells that it expanded on its way to the target cell from the agent's cell.Therefore, the h-values $h_{new}(s)$ are not only admissible but also consistent.□

Part 5

In this part, we are comparing Repeated Forward A* vs Adaptive A* algorithms. After implementing both of the algorithms and running them, we have noticed that the Adaptive A* algorithm is faster than Repeated Forward A*. For example, the following are the number of cells expanded in Repeated Forward A* vs Adaptive A*, respectively, in couple of the mazes we have generated: maze00: 8225 vs 8172 maze01: 7478 vs 7223. As the results show, the Adaptive A* algorithm is a bit faster since it does not expend as many cells as the Repeated Forward A* because each time it finds the path to the target cell, it updates the h value of cells on its path. Therefore, when the agent follows that path and runs into a blocked cell and runs the algorithm again in order to find a new path, the algorithm would use the updated h values instead of the initial ones.

Part 6

In our implementation, we have added all the improvements that we could think of as a possibility. As a result, we are using a 2-D Byte array for the maze as well as a node structure for the agent to follow to reach the target cell. The node structure includes the location (coordinates of the node) stored as integer array, the parent of the node (from which cell we got into the current cell) as a pointer, three integers for f, g, and h values, and a boolean for deciding whether to break ties between the f values in favor of the smaller or greater g values. This structure in total uses 56 Bytes of memory.

Now, in order to operate within 4 MBytes, we would have to be using a maze of a size not more than 264x264. In order to calculate the size, we used the following equation: $n^2+(n^2*56) = 4*10^6$, where n is the number of cells, 56 is the number of Bytes needed per each cell in our implementation, and $4 * 10^6$ is the 4 MBytes represented in Bytes. The first n^2 represents the 2-D array of the maze, then $n^2 * 56$ corresponds to the number of nodes times the size in Bytes of the node structure. As a result of solving this equation for n, we get that $n = 2 * 10^3/\sqrt{57} \implies n = 264.91$. Therefore, the largest maze size that we could operate on within the memory limit of 4 MBytes is 264x264.