AI Project 1 Phase 1

Harshil Shah and Emmanuel Baah

a) Refer to Code
b) Refer to Code
c) Our optimizations of the pseudo code given come from a few sources, one of them being the heap we ourselves wrote to be used as the open list. The next optimization came in the form of the closed list. We created a 2D array of booleans and used that to check if we have expanded a node before or not. After all, this array look up will be $O(1)$ constant time, every time.This is more efficient than creating a list and iterating through. Our final optimization has to do with the criteria we are using to update each node. Instead of setting $g(s') := \infty$, we do $f(s') := \infty$ in line 15 of the pseudo code. Then when we enter the updateVertex method, we check to see if $f(s)$ plus the cost of transitioning plus the $h(s')$ is less than $f(s')$. Since $g(x) + h(x) = f(x)$ for any given x, $f(s) + cost + h(s)$ is what $f(s)$ will be if we choose to advance to that node. So this way, we compute what $f(s)$ will be for each node and set the parent accordingly to the one with the smallest f value
d)

1. In this grid world, the best admissible and consistent heuristic we used was the *EUCLIDEAN* distance heuristic, the distance between the current point and the goal point. Since we are able to traverse in a diagonal manner, as well as taking into account the fact that there are points that cannot be moved through scattered around the grid, the Euclidean distance will always be less than or equal to the true distance we must travel to reach the goal point.

2. We implemented a heuristic called *MANHATTAN* which gets the Manhattan distance from the current point to the goal point. This heuristic is inadmissible because it overestimates the minimal cost path from the current node to the goal node since it does not take into account the fact that we can travel diagonally. Traveling diagonally will usually incur a lower cost than traveling across two cells to reach the same destination.

3. The next heuristic implemented is called *BORDERPOINT*, where we split the grid into four equal pieces. If the goal point is in the top left quadrant, then we focus our search, driving the path taken to the top left corner. If the goal point is in the top right quadrant, we drive the path towards that top right quadrant. The same is done with the bottom left and bottom right quadrants. This is also inadmissible because the heuristic is driving the path to one of the corners of the grid, not the goal point specifically. This means that regardless of which quadrant the goal is in, the heuristic is always overestimating the cost to reach the goal because the distance from the current point to any of the corners of the grid will always be higher than the distance of the current point to the goal. The only time this is not the case is if the goal point is also one of the corner points.

4. Our heuristic, *CENTER*, drives the path taken to veer towards the center. The logic here is that because of the way the start and goal points are generated, it can occur fairly often that the start and goal points will be across from each other in a way where the path taken will probably cross the halfway, center point in the grid. It is inadmissible because it will veer the path towards the center, always overestimating the cost when there are situations where the shortest path did not need to go through the center.

5. The fifth heuristic was *RANDOM*; a heuristic where every individual point S, where S is the current point, computes the distance from S to a random point on the grid, and slightly veers towards that point. The logic behind this one stems from the idea that random walks are probabilistically complete. So we are guaranteed that if there is a solution, our algorithm will find it. For the obvious reason, very similar to the reasons given above, this heuristic overestimates the cost from the current node to the goal node.

6. The final heuristic was *AVOIDH2T*; a heuristic where the current node s checks to see if s' is a hard to traverse terrain. If the terrain of s' is hard to traverse, the cost which is usually double the amount of traversing between hard to traverse regions will now be 1000 times the amount. So this heuristic aims to avoid hard to traverse points as much as possible. For the obvious reason, very similar to the reasons given above, this heuristic overestimates the cost from the current node to the goal node because the goal node may very well be around a cluster of hard to traverse cells. However, this heuristic will take a path that

tries its best to avoid those cells, overestimating the true cost.

e) Excel file of data results attached
f) f) Across the different maps and different heuristics, there were definitely interesting trends noted. For example, the heuristic which was admissible and consistent would have lower costs. However, our inadmissible heuristics, the Manhattan heuristic for example, would expand far less nodes, and have a cost that would be higher than the admissible heuristic. So the tradeoff was very present. The admissible and consistent heuristic is more efficient in terms of the total path cost, but it expands more nodes than an inadmissible heuristic, sometimes. Some of our heuristics would expand more nodes than the admissible heuristic, but this is because of the nature of our heuristics themselves. In terms of how weighted A* differed from regular A*, it our data shows that weighted A* made the algorithm more greedy and incur higher total costs, but as a tradeoff, the algorithm expanded fewer nodes than the regular A* algorithm. We compared weighted A* values 1.5 and 3. In all cases, the higher the weight, the higher the overall cost, but at the same time, fewer nodes were expanded. At the same time, some maps that were generated favored some heuristics just based on the location of the start and goal as well as the distribution of hard to traverse regions, so there are some outliers in the data. In terms of execution time, for any given map, we saw an improvement in execution when we went from regular A* to weighted A*. This is due to the greedy nature of the heuristic holding more of a say in the direction of the path.
g) Refer to code
h) As in the first phase, the data structures we are making use of are quite efficient in their nature. We are using an array of A*s in our sequential and integrated searches, each A* having a different heuristic, and using each A*s open list. Each A*s open list is a heap, written by us, running in $O(1)$ access time to peek at what's on top of the tree. Also, to check for containment, deletion, or popping in the heap, the heap will run in worst case $O(log(n))$. The same is true for the closed lists we are using. We utilize the closed list of each individual A*. Since each closed list is a 2D array of booleans,we have $O(1)$ lookup time, and a lower memory usage cost as opposed to other possible methods of implementing a closed list.
i)

1. We prove this by induction. The basis step is to assume that in weighted A*, the g value of any state s expanded by the algorithm is at most w1-suboptimal. For any state s for which it is true that$Key(s, 0) \leq Key(u, 0) \forall u \in OPEN$ It holds that $g0(s) \leq w1 * c*(s)$ where $c*(s)$

is the optimum cost to state s. We know this to be true so that fulfills the base step. Assume that there is an anchor key for a state s. Next, we expand s, $key(s) = f(s) = g(s) + h(s)$ Now, since f(s) is composed of g(s) and h(s), g(s) is smaller than f(s). Also, because $w1 \leq 1$, $w1 * g * (sgoal)$ will be bigger than Key(s,0).

2. Prove that the solution cost obtained by the algorithm is bounded by a $w1w2$ sub-optimality factor

j)