HW2 Writeup Jiyao Zhou

I used A* algorithm to find the relatively shortest path. The basic idea is that for every node on the map, there is a Manhattan value m, which is the straightforward distance to the end;
a cost value g, which is the minimum distance from start to that node; and a final value, which is the sum of the cost and Manhattan value. In addition, I used one open list of node and one closed list to keep track of the possible nodes that are on the path. Then while the current node is not neighbor with the end node, I push current node's nonempty neighbor onto the open list, and for every node on the list, I recalculate its final cost. In addition, I set the neighbor node's parent node as current, like linked list. In the end, I choose the node with the minimum cost and set it to current till we found the end point. Finally, we trace back the parent node to find the path for the robot.

For all four of the test inputs, the algorithm proved unbeaten as for 4x8 input 2, it took 7 moves and 43 pings to reach the destination; for 20x20 input 4, it took 38 moves and 369 pings to reach the destination. As we can see, there's almost a linear dependency between numbers of moves and pings with the scale of the map, which is the best possible case. The efficiency of the algorithm does not depend on the scale of the map, but rather on the 'difficulty' of the path. By difficulty, I mean the obstacles that makes the total moves larger. Having more obstacles makes the algorithm go through more nodes along the way, and thus generate more moves and pings.

When deal with uncertainties, I change the method to check if one node is X or O. Basically for nodes that are close to the start (Manhattan distance < 12), I ping the node three times to see if they returned the same results. If so, then we label the nodes' value as the result. If not, then we label the nodes' value as 'Unsure'. For nodes that are far away from the start (h>=12), I used a for loop of size h^2 to get the possibility of the result. For instance, if X appears more than 60 percent of the time, then we set the node as X; if it appears less than 40 percent, we set the node as O; if it is within the tricky range of 40-60 percent, we set it as 'Unsure'. Finally, while the node is 'Unsure', we keep calling this method until its value is set.

This method focuses one hundred percent on minimizing the number of moves as the robot doesn't have to move to a certain position to check its value. And its performance is relatively stable as for all 4 inputs, the algorithm never failed me once for more than 50 tries I did. Yet one thing to notice is that this algorithm, even though 99 percent of the time, could find the path, does not guarantee a path finding. But simply rerunning the algorithm again would give us the shortest path unless one is extremely

unlucky. For one hundred percent guaranteed path-finding, I would use recursive travelto() methods to actually move the robot, but it would waste so much moves.

Interestingly, for 2x4 input 1 with one X, it took 3 moves and 7000 pings. For 10x10 input 3, it took 13 moves and 3147 pings instead. But if I remove that X node from input 1, it took 3 moves and 15 pings. The difference between 7000 and 15 pings lies in that one X. My hypothesis is that the more X node we have to check, the more (on a thousand level) pings we need. It does not matter whether or not the X node lies on the shortest path. After testing the codes, the excessive pings all came from the while loop that deals with 'Unsure' nodes. One explanation is that for small input (h<12), the X node tend to give wrong result so that the system has to call checkping() many times to determine whether the node is truly X. Overall, the algorithm proved movement minimum and relatively stable, and it is efficient when dealing with maps with less X nodes.

Finally, I tested two twists on the algorithm. First, I changed the size of the for loop in the checkping() method to be h instead of h^2. This reduced the number of pings needed, yet for large map like 10x10 and 20x20, the robot bumps into a X node and the system broke down. It proved that large sample pings are must for nodes that are far away from the start in order to determine their values. Second, I changed the Manhattan distance for every node to include the diagonal distance as well. For instance, for point (1,1) and end point (5,7), the old h=5-1+7-1=10; the new h=max (5-1,7-1) =6. After the change, there's no significant change in regarding of moves and pings for 5 inputs, as I suspected. My hypothesis is that as long as the heuristic value is relatively correct in terms of how far a node is from the end point, the result would be the about the same.