CS 4710: Artificial Intelligence
Homework 2: Pathfinding
David Amin (dma3fq), Rupali Vohra (rv5rr)

For the certain case, we ping the entire map. Once we have seen every spot, we use that information to run A* on the graph. This has the benefit of always being correct and finding the optimal path, though it does use a lot of pings – a number directly proportional to the size of the map. When calculating the costs for the various moves, we treat all moves as equal in cost. The heuristic of the algorithm is the Chebyshev distance from a node to the finish location. Since our robot can move diagonally, this is the minimum number of moves it would take to reach the goal in the absence of any obstacles. From this we can infer that the heuristic is admissible, as it will never overestimate the distance to the goal; it will, at best, be exactly the same. We have tested the algorithm on many different maps, and it always finds the optimal path. That said, the algorithm does use a lot of pings, and it has to ping every position before it can start to find a path, so it can be quite slow.

For the uncertain case, we are in essence still using A*, but with a few adjustments to handle the uncertainty. Given that the robot gets a more accurate result from pinging by being closer to the target, we wrote our code in such a way that the robot plans a little, then moves a little, then plans again. The exact details of this were easy for us to alter, allowing us to get different results and to compare performance with the varied parameters. In essence, our algorithm is to have the robot plan using A*, then when the planning exceeds some maximum distance, it moves to the last spot it looked at. Upon getting to that spot, or hitting an unexpected obstacle, the robot forgets all the planning it did and starts over. If at any point it examines the final node, the robot tries to move to it. If successful, the program ends; if the robot hits an unexpected obstacle, it plans again. When planning, the robot tries to determine what spots are open or closed by pinging them multiple times. If the majority is “O,” it assumes “O.” Otherwise it assumes “X.” The robot pings a spot more if the spot is further away. This is one of the parameters we vary in order to get different results on different maps, but the simplest thing to do is ping once for each square the target is from the robot. We also experimented with changing the proportion of pings needed in order to conclude a location’s value (“X” or “O”). We also changed the maximum distance a search could go for before moving.

Our algorithm, in general, finds pretty good paths. These paths are usually within a few moves of the optimal path found by A* with certainty. It makes rather heavy use of pings, though, often using more pings than there are positions on the map. This was deliberate; we wanted to do something that pinged a lot to try to be sure about movement, especially because our solution to hitting an unexpected obstacle is less than elegant. Our algorithm does have the flaw that it sometimes fails to find a path. It can get stuck in some situations, continually examining the same area. This is a problem, but it wasn’t very frequent in our testing. Only on maps with a very obscure path that requires going far out of the way does this problem occur.

**Results:**

In the following discussion of results, the term "window size" is used to refer to the size of the radius around the robot where the robot pings before moving.

We tested our algorithm on a few different maps. The first has big blocky obstacles, in a roughly grid based layout. The idea was to do pathfinding in an area like a city. For a start and finish position that were 20 moves away, our algorithm performs fairly well. In a sample of 5 runs, it got to the goal every time with an average of 25.4 moves per run, and 627.6 pings. Compared to our implementation with certainty, which found a path of length 20 using 1024 pings, this is decent. This was using a window size of 1, only looking at the points exactly adjacent to the current position. Upping the window size to 3 results in an average of 24.8 moves, and 1915.4 pings. This trend continues with greater increases to window size. The number of pings increases drastically, as one would expect, but the algorithm finds slightly better paths. The down side is that it occasionally also finds terrible paths, 31 being the highest we saw in our tests. It is unclear if this occurs because the robot happens to get invalid information as a result of the pings, or if it is attempting to go down paths that are suboptimal but only become available when it has a large enough search space.

We tried altering how the algorithm pings; instead of pinging a location the number of times equal to the distance from the robot, we tried pinging a location by twice that amount. This had no noticeable impact on the number of moves, but the number of pings skyrocketed. It was around 5,000 on a 32x32 map. It seems that increasing the number of pings isn't really all that helpful. This is likely because the certainty for nearby elements is always pretty good, so doing more tests to be more sure isn't very helpful. If our window size were very large, this might make more of a difference, but larger window sizes seem to lead to huge growths in the number of pings we do, without actually getting better results most of the time.

For the sake of comparison, we also tried our robot on a map that looks a bit like a maze, with several wrong potential paths, obstacles that are harder to navigate around, and decision points where going the wrong way would be hard to recover from. For comparison purposes, we tried the same changes we made to the algorithm on the first test map. The optimal solution was 31 moves, and our implementation used 1024 pings. Using a window size of 1, and directly proportional pings to distance, our results averaged 39.6 moves and 975.6 pings. However, it often failed to find the goal at all, this average was only from successful runs. Upping the window size to 3 yielded averages of 40.2 moves and 3093.4 pings, but didn't fail to find the solution at all. Changing the pinging scheme to take double the distance pings gave us 38.4 moves and 7773 pings on average. It seems that for this map, the changes to the pinging strategy actually did make a difference, but again, it wasn't very much.

Our final test map was a bit different. Where the previous two were 32x32 in size, we tried a map that was 512x512 in size. It was very sparsely populated with obstacles, but the start

and finish were far from each other. The optimal solution found by our implementation that used certainty was 433 moves, using 130560 pings, and taking quite a while to do so. For comparison, our uncertainty algorithm (which got perfectly consistent results, it never changed) got the optimal solution in 11946 pings for window size of 1, and 27971 pings for window size of 3. These results aren't incredibly interesting, as the path finding itself isn't challenging on such a sparsely populated map, but the important point to consider is that the uncertain case doesn't ping the whole map, so in situations like this it is preferable to the certain solution that is known to be optimal in terms of path, but considers every possible square on the map.

**Modifications:**

We realized that we could save the results of our pinging to cut down on the number of pings we do overall. The problem with doing this is that it means an incorrect finding stays incorrect, so we have to trust that our results are reliable. By adding this, we saw minor drops in the number of pings used, but not significant changes. We made it so a found result is reused once, and then the pinging process is done again. It's possible that reusing more would have led to less pings, but it also increases the chances of an incorrect ping result leading the robot down a less than optimal path. For this reason, because our algorithm is designed to prefer large numbers of pings for shorter paths, we decided not to pursue this further.

**Conclusions:**

This approach, while it seemed feasible in practice, proved less than desirable. The path it finds is usually pretty good, sometimes even close to optimal. That said, ping usage is incredibly high. Often in the thousands for fairly small maps. It makes sense that the number of pings in uncertain pathfinding would be greater, but not that much greater. Dealing with uncertainty does require some tradeoffs though, it's unreasonable to expect identical performance to the certainty case.

Aside from looking at the algorithm overall, it was interesting to see how changes to the window size and the number of repeating pings impacted the pathfinding. Bigger window sizes helped to prevent the robot from getting stuck, and often helped find better paths. It occasionally led to extremely bad paths being found as well, which makes sense given that it has a much wider range of possibilities to consider, and less certainty about the further away points. More pings didn't have as much of an impact as we would have expected, but this also seems feasible. As our robot only considers things within the window size and the window size was often fairly small (no more than 10 in any of our tests) there wasn't a great need to try to ping a given spot a lot. They were usually close enough that the results of a given ping were reliable.

MyRobotClassUncertain2.java

Our second approach guided the robot's movement based on two factors. The first was the general direction of the finish location relative to the robot's current location. The second was the general crowdedness of a section of the map near the robot. The contributions of these two factors in determining the robot's path was varied in testing by adjusting each one's weight.

The basic concept of this approach was to split the map up into grids of maximum size 15x15, with smaller sizes for smaller maps. Each grid was treated like its own node in that we once again calculated the Chebyshev distance from the grid to the grid containing **f**inish (unless, of course, the start and finish nodes were in the same grid). We then created a probability that the robot would move in that direction proportional to the Chebyshev distance that was calculated. **(Need to handle finish grid as neighbor to start grid – make the probability SUPER high or something)**.

**We then randomly sampled some amount of nodes in each neighboring grid a number of times, searching for a majority estimate. Both the number of nodes in each grid and the number of times each node was sampled were varied in our testing to see the impact they would have on the robot. Based on these results, we knew which ones of the neighboring grids were the most likely to have a clear path through them. Based on a ratio of "O"/totalNodesSampled, we created another probability for the grids.**

**These two probabilities were then combined to determine which path was the best one for the robot to take. It is likely that the robot will perform better in relatively square maps. This is because the smaller grids are squares, and they are resized based on the smaller side of the map. If a map is very wide and not very tall, for example, there will be many grids going across the map, causing the robot to make many pings as it attempts to make its way across the map in order to calculate the crowdedness probability.**

**Information about the code:**

Our certain case algorithm was implemented in MyRobotClass.java

Our uncertain case algorithm was implemented in MyRobotClassUncertain1.java

**Our second approach to the uncertain case was implemented in MyRobotClassUncertain2.java**

The robot takes in the rows in the world, columns in the world, the starting Point, and the destination Point, and the first uncertain case takes in two ints. One is the maximum window size, and the other is which strategy to use for pinging. A 1 uses 2 times the distance number of pings. A 2 uses the distance squared number of pings. Any other int uses the same number of pings as the distance.