For the certain case, we simply ping the entire map. Once we’ve 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. When calculating the costs for the various moves, we treat all moves as equal in cost, and for the heuristic portion of the algorithm, we used the Chebyshev distance from a node to the finish. As 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’ve 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 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, getting different results and comparing performance. 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 it 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. We vary how this works exactly to get different results, but the simplest thing to do is ping once for each square the target is from the robot. We also experimented with changing the value needed to declare a value. We tried not requiring a simple majority but instead 60% of the pings one way or the other. This was easy to change on the fly for evaluation. We also changed the maximum distance a search could go for before moving.

Our algorithm, in general, finds pretty good paths. 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 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:

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’s 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 number of times equal to the distance from the robot, we tried pinging twice that. This had no noticeable impact on the number of moves, but the number of pings skyrocketed. It was around 5000. 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.