John Zhang, Tyler Ealy
jwz2kn@virginia.edu, tre7ca@virginia.edu
March 14th, 2016
Homework 2 Report

Describe your basic path finding algorithm. Show a brief analysis of how well it works on a few different datasets that you produced. What kinds of data sets are more inefficient? Why is that the case?

We used an A* algorithm as described in class. We generally followed the pseudo-code from https://en.wikipedia.org/wiki/A*_search_algorithm to implement the code. The basic idea of the algorithm is that you use an admissible heuristic to estimate cost between traversing nodes. The algorithm, a best-first search, then searches along each possible path to the solution and tries to minimize the cost of traversing nodes, thus finding the path with optimal number of moves.

For the 4 input files provided by the instructors, we measured how well our algorithm performed in the following three ways: Time it took to generate the path, number of moves in the path, and number of map pings. The times were measured on a 2015 Macbook Pro running the latest OSX.

	myInputFile1.txt	myInputFile2.txt	myInputFile3.txt	myInputFile4.txt
Time (s)	0.001740248	0.0027793450	0.00677699	0.0170871300
# moves	4	8	14	39
# pings	7	53	144	418

As we understand it, these are all the shortest possible paths to the goal, and it seems to ping the map a relatively small number of times.

A*, in general, is an efficient algorithm, but is less efficient for larger map sizes, and maps with more walls, because there's more possible paths to evaluate. As a matter of fact, map sizes increase rather quickly, so A*'s efficiency would experience large drop offs in efficiency for huge maps.

Describe how you adapted your algorithm when dealing with uncertain situations. How did you deal with the fact that the robot sometimes incorrectly viewed a space in the world?

At first, we thought that we might need to implement a different search algorithm. Because there's uncertainty, we couldn't just plan out an optimal path ahead of time---- We would need to reevaluate the path based on walls that we found. Thus, we tried to implement D* Lite, an incremental heuristic search algorithm that was based partially on lifelong planning A*. We looked at pseudo-code from http://pub1.willowgarage.com/~konolige/cs225b/dlite_tro05.pdf and information from https://en.wikipedia.org/wiki/D*. This implementation didn't work because the pseudo-code didn't match up to data structures we had been using, and because this might have required extra moves to test for walls, which we ultimately decided against!

Instead, we implemented a somewhat probabilistic solution. In our heuristic, which originally was a modified Manhattan distance that considered straight line moves cost 1 and diagonal moves cost 1.4, we added a probabilistic component. If there were uncertainty, we would ping the map for up to max(map columns, map rows)^2. If over 50% of the pings returned a wall, then we'd mark that spot on the map as a wall. We know that this will work because we're guaranteed a correct answer at least part of the time. So, unless the probability for a completely correct answer is super low, (chance of correct answer) + (number of correct answers / number of random answers) will get us to at least 50% total correct answers. So, all in all, this implementation is not strictly deterministic, or even guaranteed to work for all cases, but the chance for failure is negligible. Here's a table of results below:

	myInputFile1.txt	myInputFile2.txt	myInputFile3.txt	myInputFile4.txt
Time (s)	0.0029248380	0.00728506	0.018928693	0.0378833870
# moves	4	8	14	39
# pings	304	~4300	~35000	~200000

Note that the ~ symbol indicates approximate number of pings.

Notice that the number of moves is still optimal, but the number of pings is much larger, due to us using probability. This is by design--- It's much more important to optimize number of moves, because in reality, that's what could hinder a robot's movements!

Produce data that shows how well your algorithm performs on different inputs. What happens if you slightly tweak or change your algorithm? How do these changes affect the performance and why?

Because we used an A* search for both cases, we'll talk mostly about changes to the heuristic. In our A* search, we did remove all null nodes (i.e., off the map) for when the current node is looking at its neighbors, and this reduced number of pings greatly.

We used a form of tiebreaking between paths. Essentially, if a certain node was not already traversed, and yet has the same cost as a node already evaluated, the new node will be considered a lower cost. This has the effect of making our A* algorithm prefer untraveled nodes to evaluate, and breaks ties between nodes of the same cost. This doesn't really change number of pings much, but helps in producing a more "natural" or straight looking path, especially for test cases 1 and 3.

We decided from the beginning that our heuristic for cost was going to be based on distance between points. We knew that because we're traveling in blocks, we'd want to use some sort of modified Manhattan distance. As it turns out, if you consider the cost of moving diagonally the same as moving straight, the algorithm makes no distinction between the two, and you get some funny looking paths for test cases 1 and 3, where you ideally would see a robot traverse straight. We made a diagonal movement cost 1.4, while a straight move cost 1. This has the effect of making the

algorithm prefer straight paths, thus producing a more natural path for test cases 1 and 3, at the cost of some more pings. We made a conscious choice to give a straight path, because it would realistically be better for an actual robot--- A real robot won't want to turn diagonally a lot of times, because that means it would be covering more ground!

Lastly, when we looked at the uncertain case and pinged many times, we had to choose how many times to ping a node. We based this on the size of the map--- we should always be pinging the map for up to max(map columns, map rows)^2. If over 50% of those pings return a wall, then we'll consider that spot a wall! Originally, we pinged the walls many times, but later on, added an optimization by marking off walls in a list such that we don't even bother doing calculations on spots we already know to be walls. That was a 20% optimization in number of pings.

In conclusion, the A* search algorithm works well and produces an optimal path, with or without uncertainty.