Kevin Qian (kq4hy), Jiayi Wang (jw6dz)
CS4710: Artificial Intelligence
Homework 2: Path Finding
Fall 2015 – 9/28/15

- 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?

The first step to implementing our basic path finding algorithm is to override addToWorld and storing the information about the world for future access by travelToDestination. We need the dimensions of the world, which we store in cols and rows, the start and goal positions of the robot, which we store in start_pos and end_pos. Before we call super.addToWorld, we also create a world_map, which is a 2D array that holds Block objects that represent the x-y location. A 10 by 10 world will be represented by a world_map holding 100 Blocks.

The Block class contains the fields cost, heuristic, total, parent, and its x and y coordinates. The parent Block represents the Block "ahead" of the current Block in terms of pathing and will be explained more later. The cost field stores the cost of movement from the starting point to the current Block, with each move to an adjacent Block having a flat cost of 1. The heuristic is the Manhattan distance from the current Block to the Block at the destination.

Next, we fill in the heuristic field for each Block in the world_map by iterating through world_map using a nested loop, assigning a heuristic value to the Block based on the Manhattan distance if ping_map at that Block location doesn't return "X". Blocks on obstacles will have their heuristic value set to -10 so later when they're iterated over, they won't become parents or be on any path. After the Blocks of the world_map have all been set with their appropriate heuristic, we're going to construct a path from the robot's starting position to the end position by the add_adjacent_blocks function and the open_list and closed_list.

We use the open_list to keep track of the Blocks that we've reached at a certain point in the algorithm, moving Blocks to the closed_list after visiting them and processing them. We use a loop to iterate through the Blocks in the open_list while there is still a Block, or if the current Block is the destination Block. We start by putting the Block at the starting position in the open_list and calling add_adjacent_blocks on it, which evaluates the (up to) 8 Blocks surrounding it, calculating the total (cost) for the Block, which is its heuristic plus the cost (steps the robot have to move) to get to that Block. In add_adjacent_blocks, after we process and calculate the totals for the surrounding Blocks, we add them to the open_list (if they're not already on there). If any of the adjacent Blocks are on the closed_list already, that means they have already been processed and we'll skip them in this iteration. Once the end position Block has been reached by this algorithm, the loop exits because an efficient path has been generated. We then just step backwards through the parents of the Blocks starting with the end position Block to find the path we want, and then iterating through that list of positions to move the robot to the destination.

This algorithm is highly efficient when there is one easily determined path where each Block has minimal neighboring non-obstacle blocks, thus reducing the amount of computation required for the path selection. For example, the following maps will be faster for the robot to traverse because of the high obstacle to open block ratio:

```
S X X 0 X X X 0 X X F
0 X 0 X 0 X 0 X 0 X 0
0 X 0 X 0 X 0 X 0 X 0
0 X 0 X 0 X 0 X 0 X 0
0 X 0 X 0 X 0 X 0 X 0
X 0 X X X 0 X X X 0 X
```

```
X X X X X X X X X X X
X X X X X X X X X X X
S 0 0 0 0 0 0 0 0 0 F
X X X X X X X X X X X
```

The algorithm is most efficient when the path is deterministic. When there are dead ends close to the destination, more blocks have to be evaluated as the algorithm “back-tracks” to find another path from the dead end it went into. For small maps, this additional computation cost is negligible, but for bigger maps where there are potential for multiple optimal path like dead ends, the cost will start to ramp up. For maps similar to this, the algorithm will take a longer time:

```
S 0 0 0 0 0 0 0 0 0 0
0 X X X X X X X X X 0
0 X 0 0 0 X 0 0 0 X 0
0 X 0 X 0 X 0 X 0 X 0
0 X 0 X 0 X 0 X 0 X 0
0 X 0 X 0 X 0 X 0 X X
0 0 0 X 0 0 0 X 0 0 F
```

- 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?

To deal with uncertainty, we modified our algorithm to prioritize finding openings in walls of obstacles to traverse to, traversing through openings until we reach our destination. First we calculate the direction from the starting block to the goal block to be either north, south, east, west, north-east, north-west, south-east, or south-west. Then with each move, we recalculate the direction to handle changes based on the current block’s relationship to the destination, moving closer and closer to the destination until we’re there. When we run into a wall (when move is called but the position of the robot doesn’t change), we evaluate the wall based on the direction we’re traveling. If we’re travelling N S E or W, we construct the wall from the obstacle in front of us by iterating both ways until we find an opening. If the direction is a diagonal direction, the wall we’re evaluating is an “L” shaped wall. For example, if we’re traveling south-west and hit an obstacle, we’ll evaluate the Blocks north of the obstacle until we find an opening and Blocks east of the obstacle Block until we find an opening. We’ll compare the two openings based on their distance towards the end destination and attempt to move the Robot towards the opening closer to the end destination. If the wall scouting returns no openings, then we know that somewhere pingMap returned incorrectly, so we keep rescouting the wall until an opening is found. Once an opening is decided on, we move the robot towards it. If it hits an unforeseen obstacle, we’ll call our wall-scouting function set_ping_direc again and repeat the process, finding the robot a direction to move towards and moving it until we’ve moved the robot to its destination.

- 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?

Ideally, the algorithm works best for mazes that do not have any obstacles in the most optimal path. That is most ideal because the algorithm will never ping the map and will move directly towards the destination. Essentially as the input sizes and the number of obstacles increase, the algorithm slows down because of the number of pings. Furthermore, as the success rate of the pings decreases, a wall that is extremely long will often yield a path that will require backtracking. For example, if we are trying to go Southeast and the South wall is 30 blocks before the first opening and the East wall is 25 blocks with no opening and our algorithm decides that the first opening is on the East wall, then we would travel 25 blocks before we end up backtracking. One way we can deal with this is we ping more times (possibly n^2 or 2^n number of pings where n is the levels out), average our results, and hope that the probability of getting the correct number is greater than 50%. That way, we at least have a more accurate result than just pinging the map once no matter how far out we are pinging.

- Was the GUI helpful to you? How did having the GUI influence how you tested and developed your AI (if at all)?

The GUI was very helpful. It allowed for us to test which way the robot was moving and to visually see whether or not our algorithm was working or if it was getting the robot stuck. Seeing the robot move visually let us debug choices in our algorithm’s logic with more ease.