University of Texas at Arlington

Project #1: Pathfinding

Project Writeup

Benjamin Knight

Rogelio Chapa

November 3, 2021

This Assignment is submitted towards and in support of the partial completion of the requirements for the Autonomous Robotics Course.

Table of Contents

Hardware Design Choices

Overall Design

The overall design revolves around trying to make the robot as easy as possible to program for. After all, this class is focused on the programming of the robots *rather* than the actual building of them. Our robot specifically would be classified as a **unicycle** type robot.

Besides being a unicycle type robot, there was one other key aspect, weight distribution. To worry less about friction and have our robot perform what we actually tell it, the weight distribution was very important. Since our robot is a unicycle type, it has to use the steel ball caster seen in figure 1. The more weight that is on this ball, the more friction that is applied so the bulk of our weight (the brick itself) is positioned directly above the wheels.

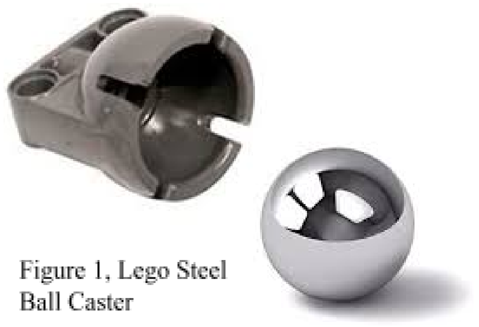


Figure 1, Lego Steel Ball Caster

Why a Unicycle Robot?

Unicycle type robots can be treated as holonomic robots with one extra step in between movement, turning. Again this allows us to focus on the programming and algorithm itself rather than how to control the robot.

Software Design Choices

Why Python?

Python was a no brainer to us after having **major** problems with Eclipse not only on Windows but the Virtual Machine/Linux. It increased our productivity exponentially allowing us to code and test our algorithms rather than trying to get the development environment up and running. We did not use any libraries other than the ones required to move our robot. Theoretically you could copy our program’s logic almost line by line into C and it would still work.

Pathfinding Algorithm: Manhattan Distance

Manhattan distance is easily translatable into code especially with how our map is laid out on the tiles. Because all obstacles, start point, and goal point are on the lines of the tiles you can directly map them to a grid. This algorithm also has an extensive amount of documentation available online while things like trapezoidal decomposition are much harder to find.

Program Flow

Map Building

The first step of the program is to build out our map. For demonstration purposes the maps shown will be 3x5 instead of the actual 10x16. In our implementation, cells are ½ foot since obstacles can be placed in between tiles, but you can imagine that the grid depicted also depicts the physical tiles of the map. Every cell in the array corresponds to it's top left coordinate marked by the red circles in figure 3. You will notice that this leaves out the bottom and right sides of the map, this is fine however because we do not want to ride the wall of the map (since it is considered an obstacle.) Later on we will be telling our robot to move from coordinate to coordinate (red circles).

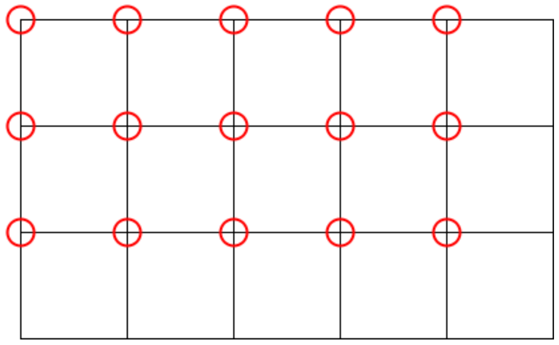


Figure 3, Initial Map

Second, we want to mark our obstacles on the map. We know obstacles will lie on the intersections so we can just iterate through all the cells checking to see if it's coordinate is represented in the obstacle array. Because we also know the exact size and shape of obstacles, (and through our planning of the array) we can then mark the neighboring cells/coordinates as obstacles too rather than just the center coordinate provided, see figure 4. The dark squares are cells in our mapp that are marked as obstacles, but the light grey square would be the object in the physical world. We also mark the edges of the map as obstacles to avoid driving into them.

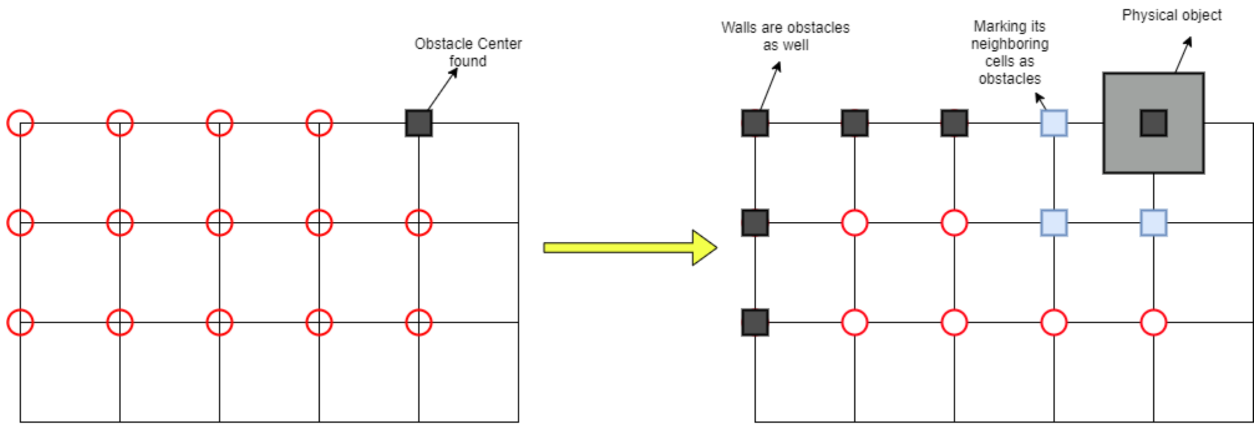

Obstacle Center found
Walls are obstacles as well
Marking its neighboring cells as obstacles
Physical object

Figure 4, Placing obstacles onto the map

Lastly, we need to mark our start and end goals. This is done the exact same way (because coordinates given lie on intersections). Just iterate through the cells and mark it as a start/goal if the coordinates match, figure 5. Now we have a map that denotes freespace, obstacles, a start, and a goal; Manhattan distance can easily be computed now.

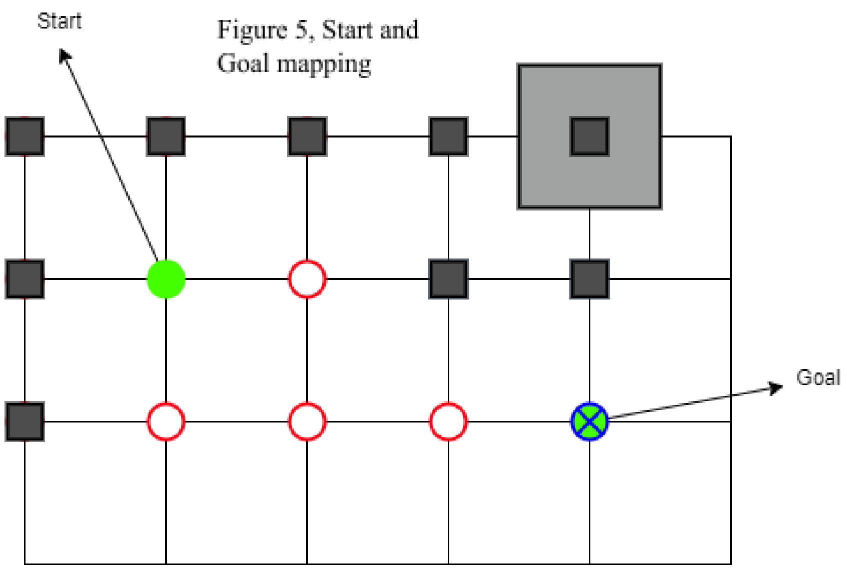


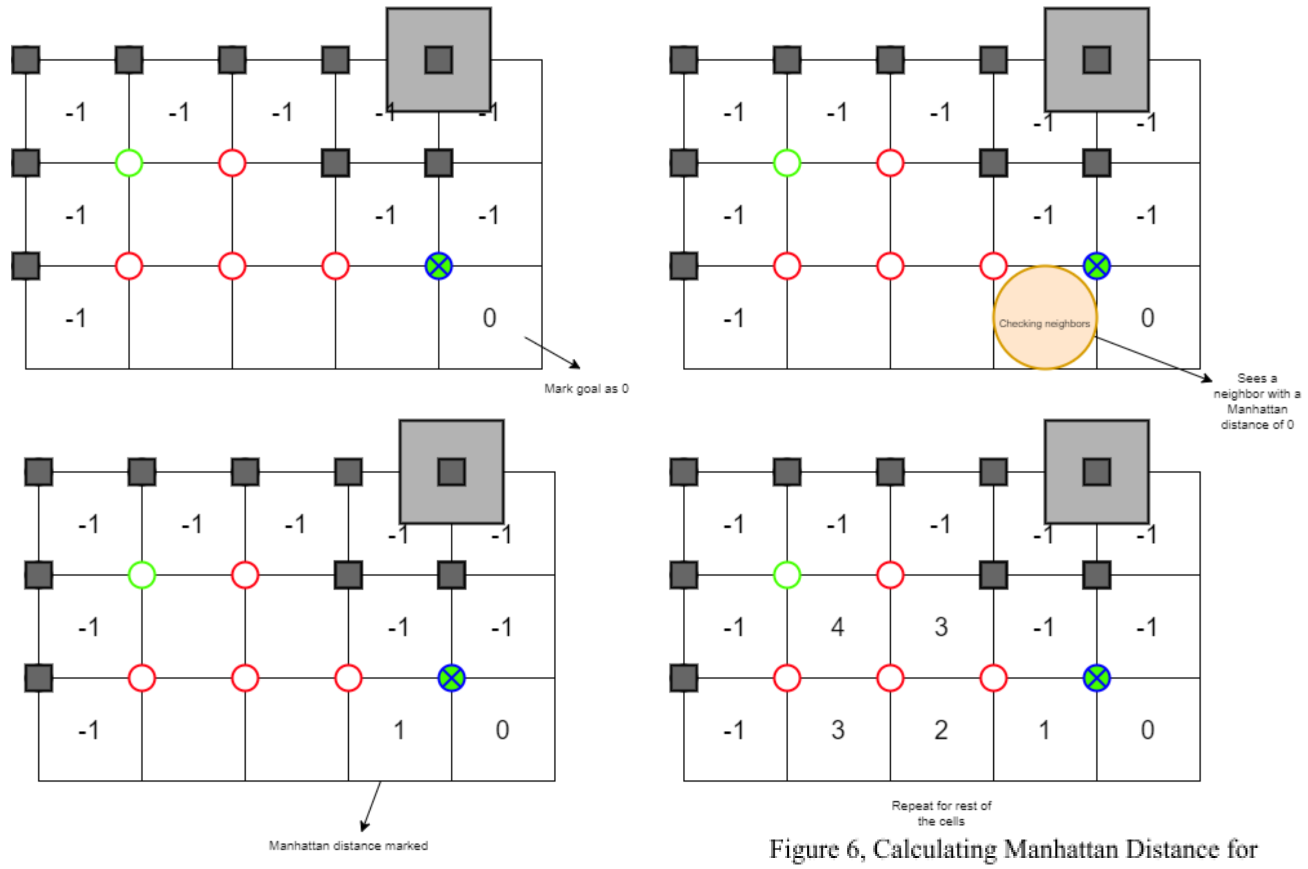
Figure 5, Start and Goal mapping

Manhattan Distance

To start with the Manhattan distance we must have a base case and that is our goal cell. This cell will have a distance of 0. Afterwards we can iterate through the array and for each cell we will:

1. Check if it already has a manhattan distance, if so skip this cell
2. Check if any of its neighbors have a manhattan distance and **is NOT an object** (denoted by -1):
    a. If the neighbors do have a manhattan distance, mark the **current cell's** manhattan distance as the neighbor cell's distance + 1
    b. Else, continue

This process can be seen in figure 6 where the Manhattan distance is denoted by the number in the middle of the cell (*remember though that the coordinate is corresponding to the top left)*.


Mark goal as 0
Checking neighbors
Sees a neighbor with a Manhattan distance of 0
Manhattan distance marked
Repeat for rest of the cells
Figure 6, Calculating Manhattan Distance for Every Cell

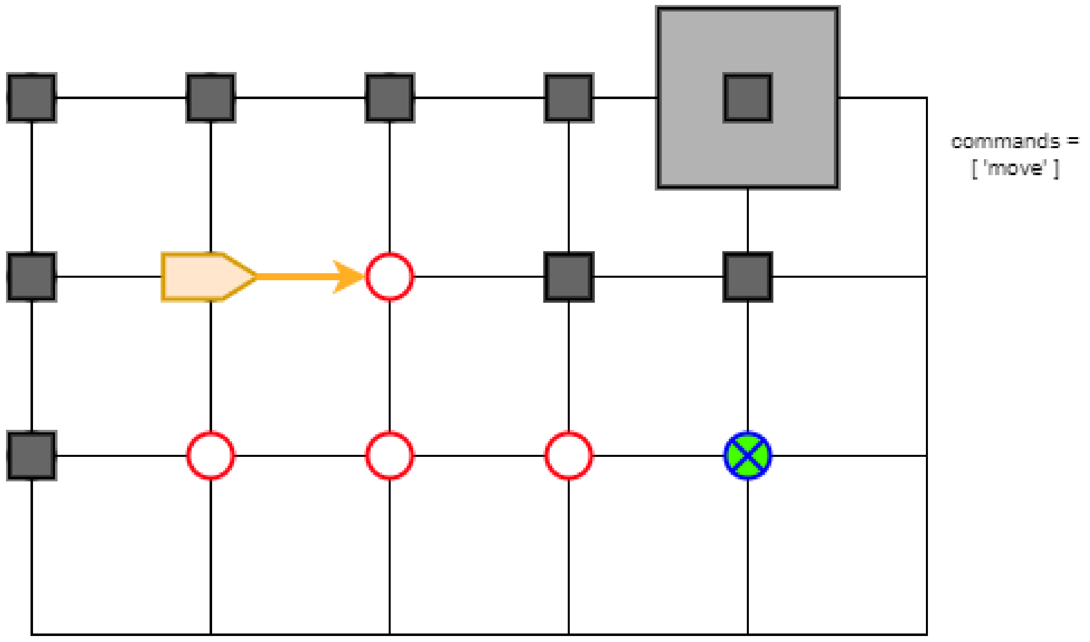
Robot Movement

To actually move our robot we have 3 commands: **turn right, turn left, and move**. This covers the entire spectrum of needed movement for the program. The turns are always 90° and movement is a complete cell (½ foot in our case). At no point will we have to move backwards or turn completely around. To achieve these movements we have to keep track of our orientation, for simplicity we decided on the cardinal directions North, East, South, West. We will always default to east and assume that is how the robot starts as well. The robot does not have a path pre-built, instead it looks at its neighboring cells and always chooses the one with the shortest Manhattan distance. If two neighboring cells have the same Manhattan distance it will default in the order that is the following cardinal direction E, S, W, N.

The process is as follows:

1. Gather all your neighbors that are valid cells (no obstacles)
2. Pick the one with the smallest manhattan distance, defaulting to a certain direction if two neighbors have the same distance
3. Turn **if needed** by comparing your orientation to the target cell's. For example if you are facing East but your target cell is South of you, you need to turn right once.
4. Move
5. Repeat 1-4 until you are at the goal.

You would put this process into a while loop checking each time if you have reached the goal cell with something like `while(!visited_goal()):`. An example can be seen in figure 8.



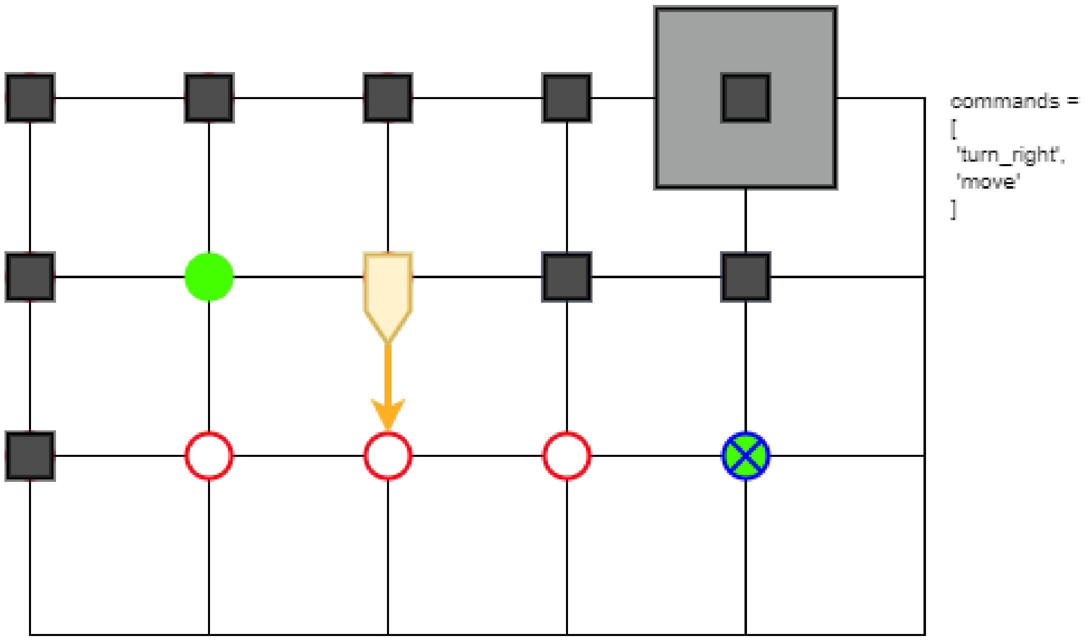


Figure 8, Robot Movement