**MP2: Configuration Space Planning**

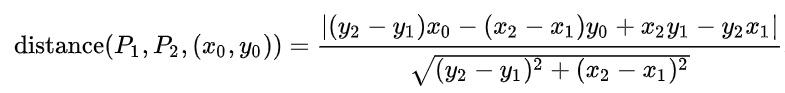
Bum Jun Kim [*bkim102*] ——— Samuel Hung [*shung5]*

**Part I: Geometry**

The first function computeCoordinate was fairly straightforward to implement — we took the given angle and length to calculate the final position using simple trigonometry.

The next function doesArmTouchObstacles was a bit more mathematical than the first. The equation we used to calculate this is the distance of a point from a line defined by two points.

We implemented the following formula we found on Wikipedia for the distance from a point and an infinite line:



We added some modifications to this formula — the first one is that the line in the formula is infinite line. This is not what we want since the robot arm is a finite length. The second adjustment is that the distance we are checking must be less than the radius of the obstacle, since the obstacle is larger than a single point.

The third function doesArmTouchGoals takes the distance between two points (goal position & arm tip position) and checks to see if that distance is less than the radius of the goal.

The last function isArmWithinWindow was fairly simple to implement as well — we took all the arm start & end positions and did some boundary checking.

**Part II: Transformation to Maze**

To transform the configuration space to a maze, we first checked how many arms were going to be required to solve the problem. This would determine the number of dimensions the maze space would occupy.

We set the angle resolution to reflect the upper and lower physical limitations of the arm. We then took the difference between the upper and lower bounds, divided that by the granularity and added one — we set that to be the size of the respective dimension in the maze. For the case of a two dimensional maze, we took the alpha angle difference, divided by the granularity, and added one to give us the number of rows in our maze. The same procedure occurs with the beta angle difference to give us the number of columns in our maze.

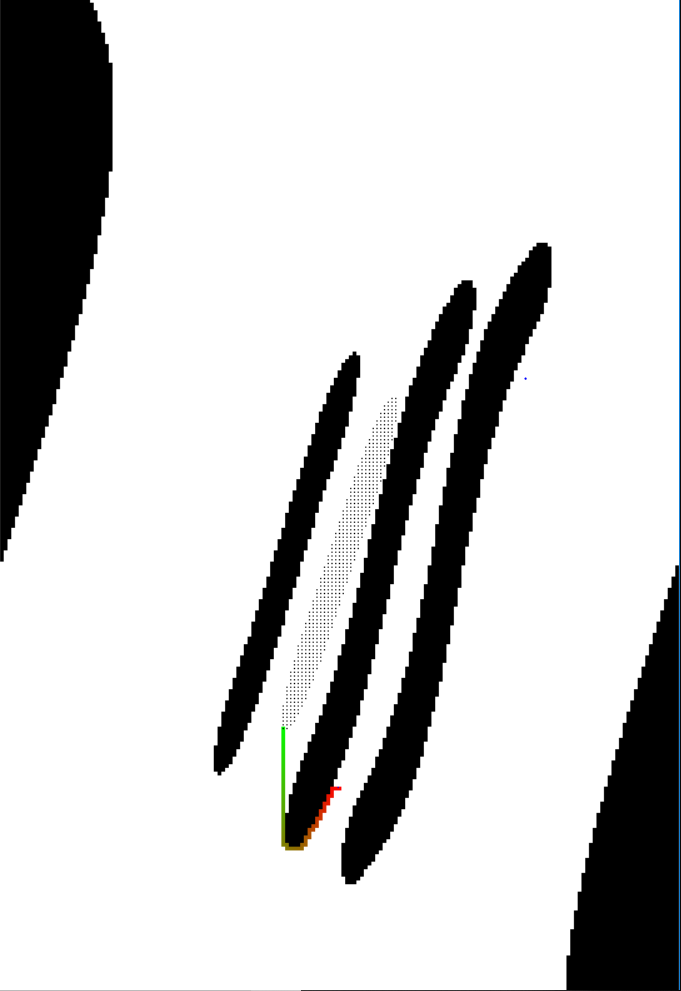
We proceed into a loop which iterates over the number of arms in the problem. We store the arm limitation information in a list structure and check the different conditions for the angle range of that arm. We have one main condition — we check whether the given angle for the arm is a valid angle. If the given angle is invalid, we set that position to a wall character. Otherwise, we check for a few more conditions.

The next few conditions are as follows: the first condition checks to see if the current angle matches the starting angle for that arm. If so, we set that position to be START\_CHAR. The next condition checks whether the arm is within bounds of the window. If the arm is out of bounds, we set that position in the maze to be WALL\_CHAR. We also then do obstacle checking — including if the non-tip section of the arm is touching a goal. That is marked as WALL\_CHAR as well. Lastly, we check to see if the tip of the arm is touching the objective. If yes, then we mark the position as OBJECTIVE\_CHAR.

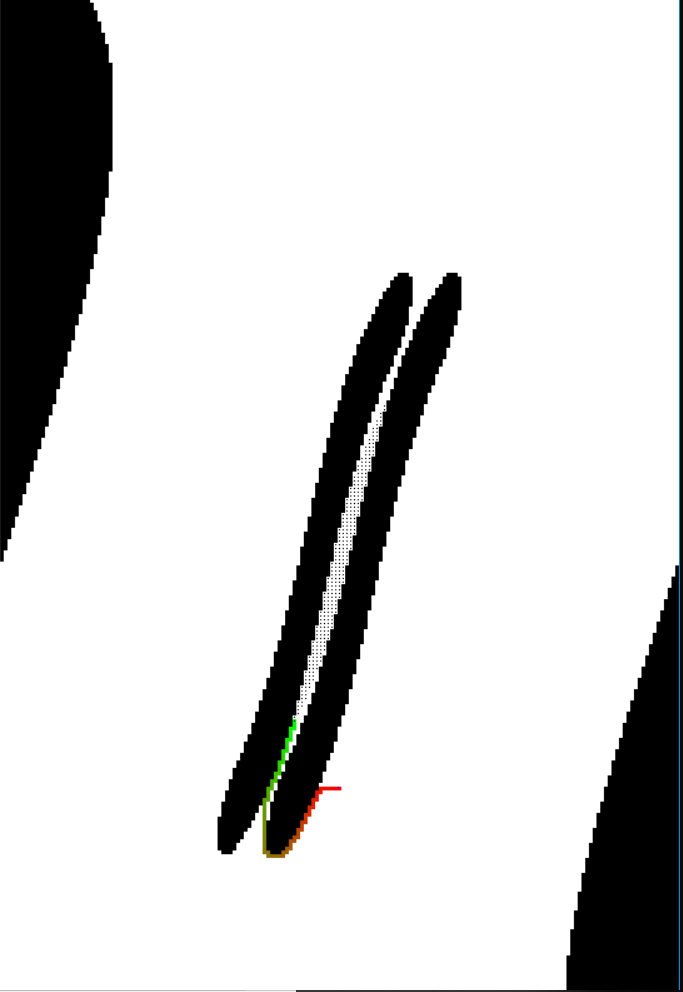
Map3 and Map5 were not able to be saved, since the mazes had no “goal” point, since either the arm body was touching the goal before the end was touching, thus it never had any states that qualified as a goal position.

Map4 was able to be generated, but the goal position was reachable only when the arms went outside of the walls, thus on the generated maze map, the goal is covered by wall characters.

Map6 and Map7 were generated, but is irrelevant since it is only a slice of the 3D map.

**Generated Maze Text Maps:**

Basic Map maze (granularity = 1)



Map 1 maze, granularity = 1



Map2 maze granularity = 1

Map3, 5 has no possible “goal state” when represented in 2D map, thus map is not saved.



Map 4 maze. (granularity=1) as seen on the left bottom side, the goal exists, but they are not reachable.

Map 6, 7 cannot be displayed since it is 3D.

**Part III: Searching the Path in Maze**

We implemented an A\* algorithm to search the maze we generated from our configuration space. In our implementation, we use push all of the nodes we want to visit to the stack.

In addition to the position of the node, the stack also stores the distance to the current node (*g)*, the total cost (*f*), and the path taken to get to that node.

This stack is constantly resorted to ensure that the node with the lowest cost is at the top of the stack to be popped out first. We calculate cost using a function *f* = *g* + *h*, where *g* is the existing path length to the current node and *h* is the Manhattan distance between the current node and the objective.

We have a continuous loop which continues until all the elements in the stack are popped out (meaning that there are no more nodes to explore), we hit a magic number threshold (to prevent infinite looping), or until we reach our objective.

Although it is quite memory intensive, the solution path is returned immediately after we reach the objective, due to constantly storing the current path taken to each node. Since we are resorting the stack for each node popped off the stack, we can guarantee that we are choosing the nodes with the lowest cost — hence giving us our solution path since it is part of the data stored in the stack.

**Part IV: Extra Credit**

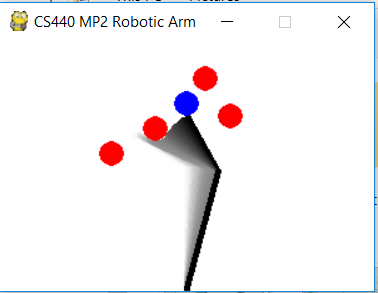
We extended the program to work on all cases: 1, 2, 3 arms. The some extra helper functions were implemented to work with 1 and 3-arm cases. By looking at the length of the coordinates, such as startposition, the function can determine what dimension the problem is. A\* search function also extended for three different dimensions, due to problems in unpacking coordinates. The A\* search method also got some tweaks, such as prohibiting placing a new coordinate in the stack if it was already in the stack to optimize the search.(it was allowed in MP1 due to comparing two splitted path merging onto a same coordinate, and comparing which route was faster) Also, the heuristic changed so that it would be measured by closest goal only.

For the 3-arm problems, especially map6 (resulted in 46169 states explored with granularity of 3, and 10775 states explored with granularity of 5), resulted in very long runtime with granularity of 1 which we never got to see the end of, and even with 2 I had to leave my laptop on overnight to see the outcome.

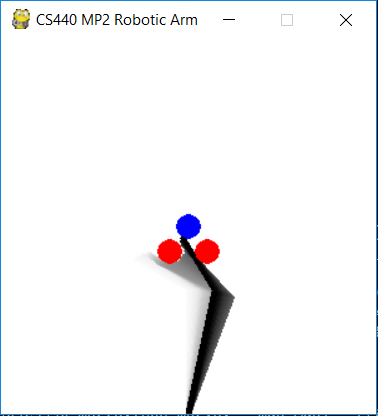
**Solution Table**

|  |  |  |
| --- | --- | --- |
|  | ***Length of Solution Path*** | ***# States Explored*** |
| **BasicMap(gran=1)** | 64 | 1403 |
| **Map1(gran=1)** | 83 | 6176 |
| **Map2(gran=1)** | 121 | 5221 |
| **Map3** | N/A | N/A |
| **Map4** | N/A | N/A |
| **Map5** | N/A | N/A |
| **Map6(gran=5)** | 46 | 10775 |
| **Map7(gran=5)** | 39 | 2376 |

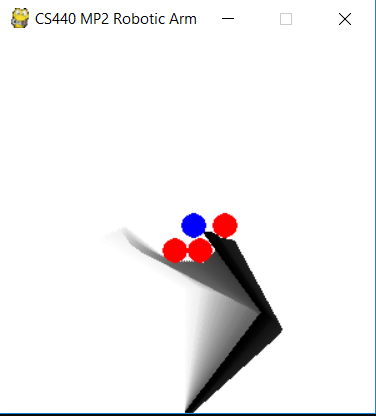
**Solutions/Explored Paths:**



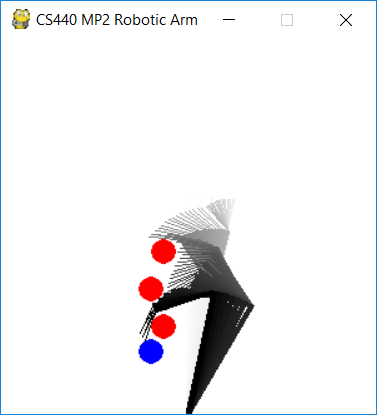
Basic Map



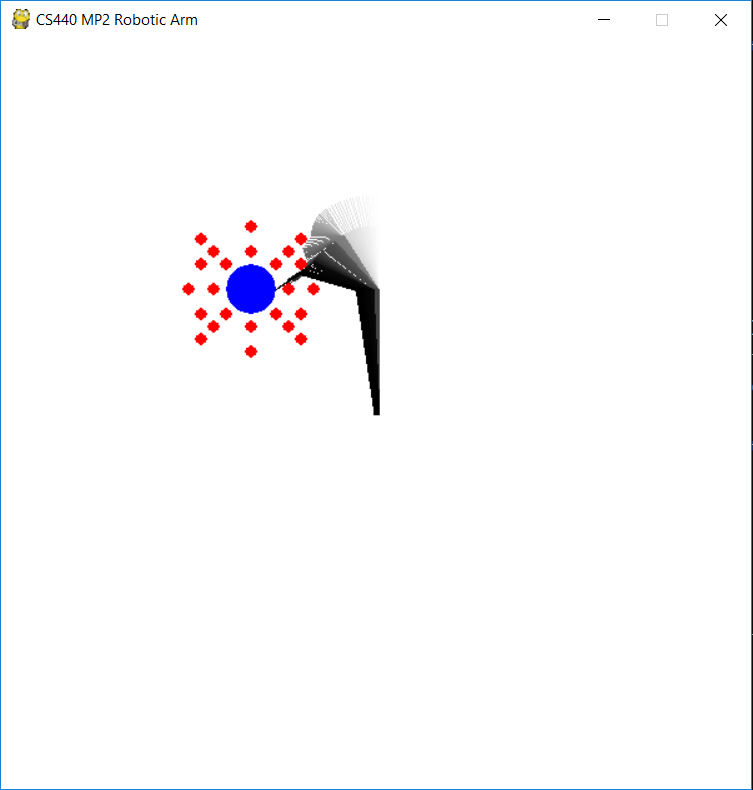
Map1



Map2



Map6



Map 7