ME 5751 Module 3: A* Algorithm

Cal Poly Pomona
Pomona, CA
Jonathan Jenkins
jjenkins@cpp.edu

***Abstract*— In this project we developed a custom python A* algorithm to take a user suppled map, start point, and goal point to generate an optimal travel path. The final path is then exported for later use in robot motion simulation. The methods used to test the controller involve a gui (graphic user interface) which simulates the robot and a custom python algorithm. The mathematics involved were sourced from "Introduction to autonomous mobile robots", 2nd Ed. by Siegwart, Nourbakhsh, and Scaramuzza. The findings in this project demonstrate our custom A* algorithm effectiveness in generating useable pathways from a wide variety of user supplied maps. Each test of our algorithm was a success, and the algorithm was capable of correctly developing cost maps, prioritize different paths based on user specifications and output an optimal pathway that would provide the needed information for future robot motion planning algorithms.**

***Keywords*—Motion Planning, Grid Map, A* Algorithm map, Prioritization, Heuristic equation**

I. INTRODUCTION

Have you ever been given multiple options to the same location while using Google Maps? How do you choose the path that you should take? Do you choose the shortest distance, the fastest travel time, or perhaps the one with the safest roads? Navigation and path planning is a major focus in modern robotic. The ability for a robot to transverse in an unknown environment relies on the ability to select a suitable path to travel.

The goal of this lab is to develop an algorithm capable of generating an optimal path to take the robot from point A to point B. This path planner must develop the optimal route for any given map and do it as quickly as possible. There are a variety of path planning algorithms and in this project, we will be implementing A* algorithm. After the development of our custom algorithm, we then conducted three experiments to determine the effectiveness of the code.

This paper will go into depth on different elements of this project including control theory, different path planning algorithms, python programing, and evaluation of the final algorithm's ability to develop optimal paths from user supplied environmental image, a starting position, and a goal position.

II. BACKGROUND

Every day you drive, bike, or walk you are actively path planning. You decide the optimal route to take depending on traffic, distance, obstacles and more. Generating a good travel path is critical to the robot's ability to navigate within its environment. As shown in Figure 1 an incorrect path planner can make the difference between reaching your desired goal and crashing.

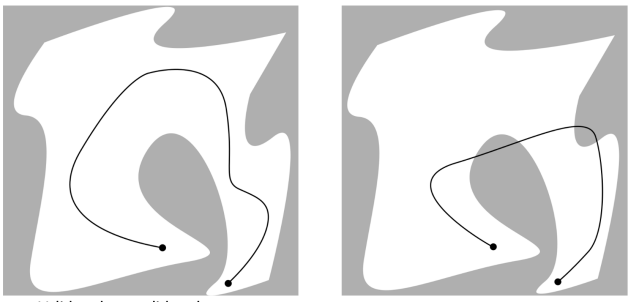


Figure 1 Good vs Bad Path Panning

A path planner begins with a map. A maps in our case is a grid that shows occupied and unoccupied spaces. Maps forms the foundation of determining the possible paths a robot can take. An example of a grid map is shown in figure 2.

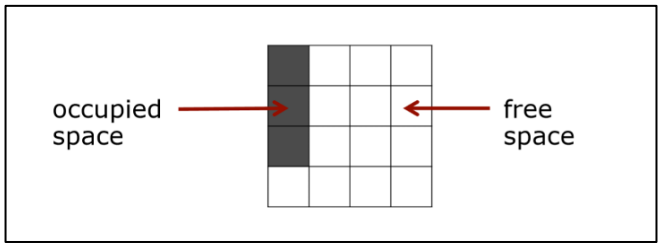


Figure 2 Position Map

Maps can be used to represent a variety of information. In this project we are developing different types of cost maps and using that information to determine the optimal path to any desired location. Cost maps assigns each

grid a value or "cost" depending on what that specific cost map is trying to represent. For example, a cost map can show the distance away from a known occupied space as shown in figure 3.

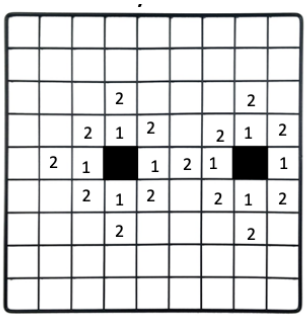


Figure 3 Distance Cost map

The distance assigned to each occupied space in this example is based on the Manhattan distance formula as shown in equation 2.1.

$$d = |y_1 - y_0| + |x_1 - x_0| \qquad (2.1)$$

In this project the initial cost map used will be a potential cost map. The potential cost maps assign a risk value to all empty spaces. Spaces at a closer distance to an occupied space will be given a higher risk or cost and spaces that are farther away are assigned a lower risk. An example of a potential cost map is shown in figure 4.

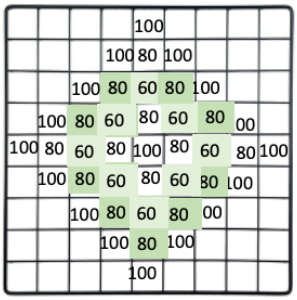


Figure 4 A Potential Cost Map

Our specific potential cost map consists of 3 important concepts. First an occupied space has the highest risk since it represents a physical object. Occupied spaces will have a risk value of 1000. Around all occupied spaces is a "no go zone" or safety zone that is treated like an occupied space. We have this space to compensate for the robot's chasse radius and prevent accidental collisions with the wall. The safety zone will have a cost between 500-800. Lastly a gradient of cost spreads from the end of the safety zone. This cost gradient becomes a cost of 0 once the distance is 1 full robot diameter away from the wall. To assign the cost to each grid a linear equation, as shown in 2.2, was used. A visual representation of our potential cost map is shown in figure 5.

$$y = mx + b \quad m = \frac{y_2 - y_1}{x_2 - x_1} \qquad (2.2)$$

Figure 5 Our Potential Cost map

With a Cost map in hand, we then use a user provided starting position and goal position as inputs to our path planner. The path planner will take these three inputs to calculate the optimal path between the start position and goal position.

Fundamentally, a path planner decides what grids should the robot move in order to reach the desired location. Each grid that can be moved to is called a node. Connections between nodes forms your path. It is up to the path planner to choose the optimal path. A visual example of nodes and pathways is shown in figure 6.

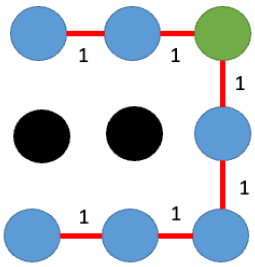


Figure 6 Nodes and Pathways

There are a variety of path planer algorithms that can be used. In this lab we will discuss 4 specific algorithms. Breadth First Search algorithm, Dijkstra's algorithm, Greedy Search algorithm, and lastly A* algorithm.

Breadth First Search (BFS) algorithm is the most basic. This algorithm searches an entire map beginning from the robot start location. As the algorithm indexes to each grid, it assigns a value corresponding to the distance that grid is to the starting location. Like the distance cost map, the BFS algorithm implements the Manhattan distance formula as shown in equation 2.1. An example of this algorithm is shown in figure 7.

Figure 7 Breadth First Search Cost Map

After all grids are given a value, the algorithm will then test different path options. Each path is assigned a total cost value equal to the sum cost of all the grids/nodes that makes up that path. The pathway with the lowest cost is considered the most optimal path. An example of this process is shown if figure 8.

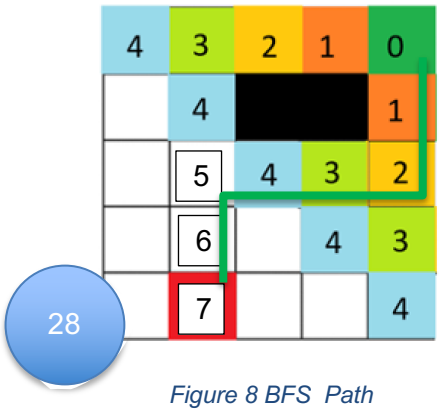


Figure 8 BFS Path

The BFS algorithm is very good at finding and selecting the shortest path between the start and goal positions. However, it expands in all directions from the start position and required completing the entire map to find a path. Dijkstra's algorithms build upon the BFS algorithm and implements prioritization. Instead of expanding evenly in all directions it will expand pathways with the lowest total cost first. Dijkstra's algorithm will take into account the potential cost map along with the BFS cost map and spread from grids with lowest total cost. The node cost can be written as shown in equation 2.3. Where the node cost [d (i,j)] is the cost assigned by the cost map [c(i,j)] . the cost map value will be the sum of both the potential cost map and BSF cost map.

$$d(i,j) = c(i,j) \tag{2.3}$$

As mentioned, the Dijkstra's Algorithm calculates the grid's distance from the start point. What if we calculate the distance from the grid and the goal point, and prioritize grids already closest to the goal point? Greedy Best First Search algorithm preforms this exact process.

The "greedy" algorithm implements a heuristic function that assigns a cost to a node based on the grid's estimated distance to the goal point. The node cost can be written as shown in equation 2.4. Where the node cost [d(i,j)] is equal to the heuristic cost [h(i,j)]. The distance cost can be defined by any method, but we will be using the manhattan distance within our script. Figure 9 shows an example of greedy algorithm using Euclid distance formula.

$$d(i,j) = H(i,j) \tag{2.4}$$

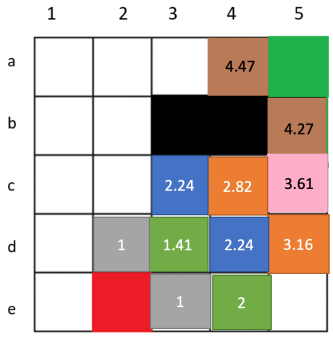


Figure 9 Greedy Algorithm Node Cost map

While Dijkstra's Algorithm calculates the distance from the start point, and the Greedy Best-First Search estimates the distance to the goal point. The A* algorithm uses both the actual distance from the start and the estimated distance to the goal. A* algorithm is design to combine the benefits of both methods.

In the A* algorithm the node cost can be written as shown in equation 2.5. Where the node cost [d (i,j)] is equal to the known cost from the start point [c(i,j)] and the estimated cost from the goal point[h(i,j)]. An example of the A* grid map is shown in figure 10.

$$d(i,j) = c(i,j) + \mu H \quad (2.5)$$

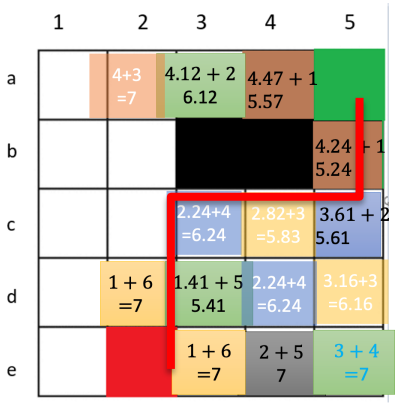


Figure 10 A Node Cost map*

With the A * algorithm we now have an efficient way to assign a cost to each grid, priorities grids that are both close to the goal position and close to the start position. Now with the optimal path found between the goal and start point the algorithm needs to reconstruct the discovered path into a clear list of instructions.

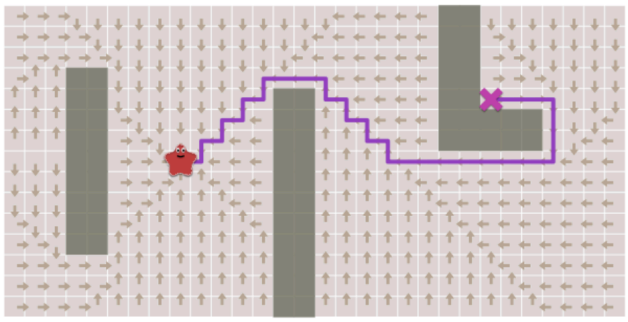


Figure 11 Path Reconstruction

As mentioned, the A* algorithm begins from the starting grid. At this starting point the program will index to all surrounding grids (neighbors) and assign then a node cost. We call the primary grid a parent grid, and all neighbors a child grid. An example of this concept is shown in figure 12.

In figure 12 the starting position is node A. Node A would be the parent node for alpha, beta, and gamma, who would be the child nodes. When the neighbors for alpha is expanded, Alpha will be considered the parent and all nodes connected to it would be considered the child.

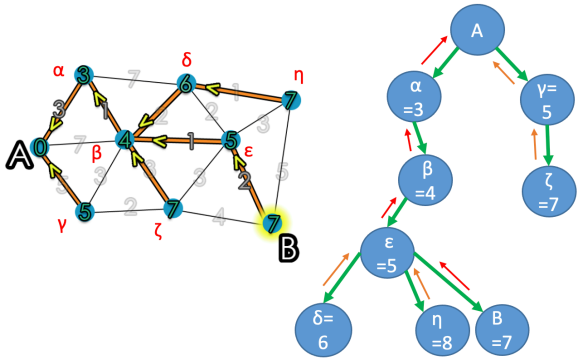


Figure 12 Parent / Child Grids

Every node now has a parent and child relationship. This relationship for each grid is stored in a dictionary queue. When reconstruction is ready, we pull the list of each parent and each parent's parent beginning from the goal position. This list will eventually lead directly from the goal position to the start position forming the desired path the algorithm calculated.

As you can imagine multiple nodes are interconnected, and many parent nodes can have the same child node. However only the parent with the lowest cost to the child will be stored. This is to ensure the smallest cost distance to each node is saved. An example of this is shown in Figure 13.

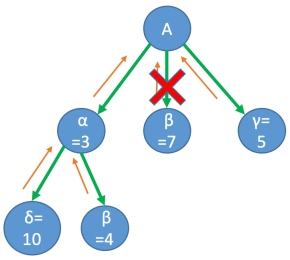


Figure 13 Parent With The Lowest Cost

III CONTROL DESIGN

The objective of our algorithm is to generate a path between our robot starting position and a goal position. Implementing A* algorithm our code will take a given grid map, cost maps, and heuristic functions to generate an optimal path relatively quickly. To implement our custom algorithm onto our virtual robot we developed a custom python script to derive the A* path into four key steps.

Before our A* algorithm can begin it needs to be given the desired starting location and goal location of the robot. Each location is given in the world coordinate space but if transferred to the image grid space for the path planner function. The code used for declaring our goal and starting location can be seen in figure 14.

```
# Specify Starting Location (0,0)
self.set_start(world_x = 0, world_y = 0)
#Set Goal Location in World coordinate ...  Location will be updated to
self.set_goal(world_x = 126.0, world_y = 59.0, world_theta = .0)
#Run Path Planner Function
self.plan_path()
# Show Calculated Path
self._show_path()
```

Figure 14 Set Start/Goal Positions

Step 1 of our code is used to setup the necessary variables, arrays, queues, and grids that our algorithm will use. We begin by importing our grid map that was generated by our previously made code *cost_map.py*. This cost map will show occupied and unoccupied spaces with assign cost values. Occupied spaces are assigned a value of 1000. While unoccupied spaces are assigned a value of 0-800. Unoccupied spaces with a cost of more than 500 are within a "no-go" zone we called safety zone. We will treat all spaces with a cost value of 500 or more as an occupied space.

Next, we will generate two additional maps. The BFS map will store the *breath-first-search* cost value for each cell. the BFS cost will be given by each cells distance from the robot starting position. The *Search_map* grid was used to store the sum cost of each grid, combining the BFS_map, original cost_map, and heuristic map together. The aggregated cost stored in the search_map grid would be what the A* algorithm uses.

In step 1 we are declaring 3 important queues that is necessary for the A* Algorithm. The priority queue called *Frontier* is used to store the grids who are actively being expanded. This queue will store the list of grid coordinates along with the current sum cost. It will sort the list with the lowest cost on top. This organizations is critical to A* search since we want to priorities expanding from the grid with the smallest cost. The code used for step one can be seen in figure 15.

```
#------------Main Code-----------------------------------------------------------------------------


print("------------")                          # CHECK OUTPUT DURING TESTING

#STEP 1:  Setup/Variables/Arrays
#Grids
grid= self.costmap.costmap                                                  # EMPTY = 0 / OCCUPIED = 1000 /
self.row_length = len(grid)                                                 # Length of the rows
self.col_length = len(grid[0])                                              # Length of the columns
search_map=np.zeros((self.row_length,self.col_length), dtype=int)           # search_map map ----- THIS MATR
bfs=np.zeros((self.row_length,self.col_length), dtype=int)              # bfs map ----- THIS MATRIX REPRESENT
#Arrays
start =(self.start_state_map.map_i,self.start_state_map.map_j)
goal =(self.goal_state_map.map_i,self.goal_state_map.map_j)
bfs_current_queue =[]                                                        # Take the list of Islands as the f
bfs_current_queue.append(start)                                              # Take the list of Islands as the fi
direction = ((-1, 0), (0, -1), (1, 0), (0, 1))                               # Left, bottom, Right, Right
#variables
distance = 1                                                                 # Node Value
nodes_search_counter=0
path_found=False
u=1
#Queues and Dictionary
frontier = PriorityQueue()                                  # the list of current spaces being expanded and
frontier.put((0,start))
came_from = dict()                                          # Stores the path to each grid
came_from[start] = None
cost_so_far = dict()                                        # Stores the Sum cost of each grid
cost_so_far[start] = 0
```

Figure 15 Step 1 Code

Step 2 begins with a check if the desired goal location exists in an occupied or unoccupied zone. If the goal location is within an occupied zone, it will inform the user to change the goal location since no path will allow the robot to reach the current position. Next, we begin with assigning each unoccupied BFS cost value. This value is given by the distance the grid is from the starting location. This process is like the brushfire algorithm we developed before so that code was reused. The Manhattan distance equation was used to calculate each grid BFS cost. The code used for step 2 is found in figure 16.

```
if  grid[goal] > 500:                                  # If Goal Location is within a wall or the saftey zone  then send message to user
    print("Your Selected Goal Location lies in a Wall -- Please Change Goal Location")                          # CHECK OUTPUT DURING TESTING

#SET 2  Reused Brushfire Algorithm used for Breadth First Search
# Perform Burstfire to give each grid a distance value away from the startpoint
while bfs_current_queue:                    # loop as long as a list of neighbors to solve for exists
    bfs_next_queue = []                     # clear the new found neighbors list
    while bfs_current_queue :                  # FIND ALL THEN NEIGHBORS AND  SAVE THEM TO THE bfs NEXT QUEUE>>> ALSO RECORD THE CURRENT NEIGHBOR DISTANCE
        (r,c) = bfs_current_queue.pop()        # get the next neighbor position  from the current neighbor list
        for i ,j in direction:                    # INDEX TO ALL NEIGHBORS AROUND THE SELECTED CURRENT POSITION
            current_r=  r + i
            current_c=  c + j
            if 0 <= current_r < self.row_length and 0 <= current_c < self.col_length and grid[current_r][current_c] == 0 and bfs[current_r][current_c] == 0:
                bfs[current_r][current_c] = round(distance, 1)
                bfs_next_queue.append((current_r, current_c))                  # Add neighbor  to next  loop queue
    distance = distance+1                  # index the distance value for the next round of neighbors
    bfs_current_queue  = bfs_next_queue            # Save the list of new neighbors to the QUEUE and rerun bfs untill all neighbors have been given a distanc
```

Figure 16 Step 2 Code

Step 3 contains the A* Algorithm. Beginning from the robot start position each neighbor that is not an occupied spaced is index too. When the function indexes to any specific grid their grid cost is compiled and stored under Search_Map[current]. As mentioned the cost of each grid is the sum of three products; The cost given by the *cost_map* , the *breath-first-search* function , and the heuristic function . The cost map was given by the relationship the grid had to the nearest occupied space. The BFS cost was given by the grids distance from the starting location using the Manhattan distance. The heuristic cost was calculated by the grids

distance from the goal position, as shown in figure 17. The heuristic cost also is given a scalar to adjust the level of influence this variable has on the overall A* algorithm.

```
def heuristic(goal,next):
    (x1, y1) = goal
    (x2, y2) = next
    manhattan= abs(x1 - x2) + abs(y1 - y2)
    return manhattan
```

Figure 17

Next the node cost for each neighbor grid is derived. The node cost is the sum of all the individual grid cost that this specific path took to this specific grid. The list of neighbors are then stored in the Frontier function, only if the grid has never been reached before or if the node cost for that grid is less than the node cost previously made using a different path. This process of looping to neighboring grids, calculating the grid cost, and the node cost is repeated until a path to the goal position is found or all grids are exhausted.

The frontier queue uses PriorityQueue which prioritizes searching grids with the lowest cost. When new grids are added to the frontier queue during the looping process they are also added to Come_From function. The Come_Frome function stores the parent grid to each child grid. This will be critical for the next step, reconstruction of path. The code used in step 3 is shown in figure 18.

```
#SET 3  A* ALGORITHM
#Perform Path Plan search and find the path with the lowest total cost:
while not frontier.empty():                          # loop as long as a list of neighbors to solve for
    priority,current=frontier.get()                  # Pull the position with the lowest cost
    (r,c) = current                                  # save current grid  position to be used when index
    if current == goal:                              # Early exit
        path_found=True                              # Variable to be used to determain what is done aft
        #break
    for i ,j in direction:                               # INDEX TO ALL NEIGHBORS AROUND THE SELECTED CU
        current_r=  r + i
        current_c=  c + j
        next=(current_r, current_c)
        if 0 <= current_r < self.row_length and 0 <= current_c < self.col_length and grid[next] < 500   :
            search_map[next] =  grid[next]+ bfs[next]+w*heuristic(goal, next)
            new_cost = cost_so_far[current] + search_map[next]
            nodes_search_counter=nodes_search_counter+1
            if next not in cost_so_far or new_cost < cost_so_far[next]:
                cost_so_far[next] = new_cost                                    # Store the
                priority = new_cost                                             # Copies co
                frontier.put((priority,next))                                   # Store the
                came_from[next] = current                                       # store the
```

Figure 18 Step 3 Code

The fourth step in the algorithm consist of reconstructing the path. Before we can reconstruct the path, we check if the A* algorithm found a goal position. If the goal was not found that means no path exist between the robot start location and goal. If a path was found, then the reconstruction can begin.

As the A* fuction calculated the path cost for all neighbor grids (child) to the "current" grid (parent) the relationship for each grid was stored in the dictionary queue called *came_from*. Now with the final position found (goal) we can search who the parent grid was for each instance all the way back to the starting position. This reconstruction is saved in an array called '*points'*. This array is then reversed so that it begins from the starting location instead of the goal location. The code used for this section is shown in figure 19.

```
#------------ ADDITONAL CODE---------------------------------

def reconstruct_path(came_from, start, goal):
    current = goal
    path = [current]
    while current != start:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path
```

Figure 19

Finally, the program will export the calculated path and the program will display a visual representation of the path in the user interface. This code is shown in figure 20.

```
#SET 4   Generate output path
# Output... Depends if path was found or not
if path_found==True:
    print("PATH TO GOAL FOUND")                                  # CHECK OUTPUT DURING TESTING
    print("the number of nodes searched with Early Exit ")       # CHECK OUTPUT DURING TESTING
    print(nodes_search_counter)                                  # CHECK OUTPUT DURING TESTING
    np.savetxt('Log/distmap.csv',bfs, fmt='%-1d', delimiter=',')  # CHECK OUTPUT DURING TESTING
    np.savetxt('Log/searchhmap.csv',search_map, fmt='%-1d', delimiter=',')  # CHECK OUTPUT DURING TESTING
    points = reconstruct_path(came_from, start, goal)            # Run function to generate path points from Came_From list
    for p in points:
        self.path.add_pose(Pose(map_i=p[0],map_j=p[1],theta=0)) #theta is wrong        # Export pathpoints to program
    self.path.save_path(file_name="Log\path.csv")                                      #export calculated path to file


if path_found==False:
    print("NO PATH TO GOAL FOUND")                       # CHECK OUTPUT DURING TESTING
```

Figure 20

IV. METHOD

After completing the development of our A* algorithm, we then moved to develop suitable challenges to test the code. The algorithm will be tested on multiple different maps, and on each map the algorithm will be evaluated by three metrics. First, we will evaluate the successful creation of the generated path without any collision with occupied spaces. A successful map generation is shown in figure 21.

Figure 21 Exported Cost Map Overlay

Second, we will evaluate the number of grids the A* algorithm inspected. A* algorithm goal is to improve upon Dijisktra and Heuristic algorithms. Ideally, the A* algorithm will allow fast path calculation like heuristic algorithms but with the superior efficiency of solving complex maps like Dijkstra algorithms. Comparing the amount of grids the algorithms calculated with and without early exit will show how many fewer grids were inspected. This node counter is shown in figure 22. A visual representation of early exit is shown in figure 23.

```
if 0 <= current_r < self.row_length and 0 <= current_c < self.col_length and grid[ne
    search_map[next] =  grid[next]+ bfs[next]+u*heuristic(goal, next)
    new_cost = cost_so_far[current] + search_map[next]
    nodes_search_counter=nodes_search_counter+1
    if next not in cost_so_far or new_cost < cost_so_far[next]:
        cost_so_far[next] = new_cost
```

Figure 22 Node Counter Variable

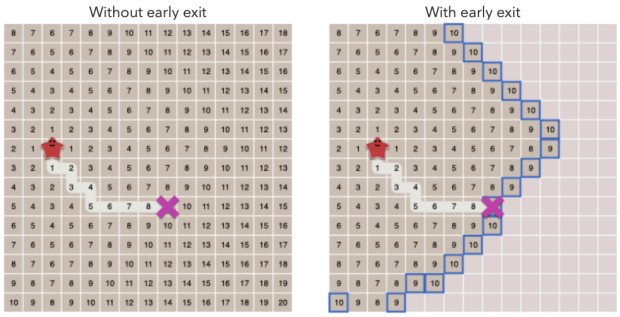


Figure 23 Early Exit

Lastly, we will evaluate the effect adjusting the heuristic has on the A* algorithm. Ideally, the heuristic scaler will be left as 1 (u=1). However, for more greedy operations the heuristic scaler can be increased. We will evaluate if increasing this scaler has any major effects on our A* algorithm.

$$d(i,j) = c(i,j) + \mu H(i,j)$$

Figure 24 A Node Cost Equation*

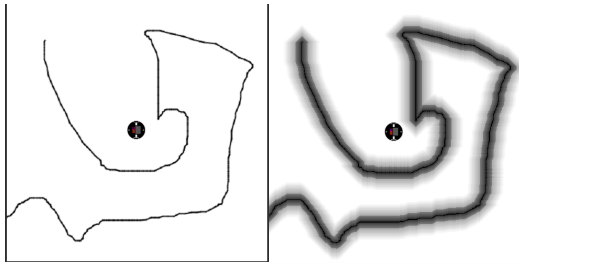
Using this process, we will evaluate the effectiveness of the algorithm to calculate a path planner from any given map, start position, and goal position.

V. EXPERIMENT RESULTS AND DISCUSSION

As mentioned, to determine the effectiveness of our A* algorithm we tested the program with three custom maps. Each test will require the program to successfully develop the individual grid cost, find possible path to the goal and finally select the best path with the lowest cost and export it to the path planner.

The first experiment uses the professor's provided test map. This map represents an irregular, unstructured obstacle. Figure 25 shows the given map, the cost map, and the robots' starting and goal position. We begin by evaluating the number of grides the A* inspected. Without early exit the algorithm calculated 803984 grids cost. With early exit enabled the number of grids the A* algorithm needed to inspect before finding the solution was 395907. This represents a 50% reduction of grid calculation the algorithm performed to solve the solution. Figure 26 shows the number of nodes searched.

Next, we evaluated the effect of adjusting the heuristic scalar. With a default of u=1 the algorithm solved the path planner without any issue. We then adjust the scaler to 10 (u=10). As shown in figure 27 the pathway is prioritizing getting to the goal position more directly. Overall, the program was successfully able to develop the pathway for safe robot travel. No visible errors appear in the final map.



```
# Specify Starting Location (0,0)
self.set_start(world_x = 0, world_y = 0)
#Set Goal Location in World coordinate ...  Location will be updated to
self.set_goal(world_x = -152.0, world_y = -73.0, world_theta = .0)
```

Figure 25 Experiment 1 Map

```
PATH TO GOAL FOUND
the number of nodes searched with Early Exit
395907
```

```
PATH TO GOAL FOUND
the number of nodes searched
803984
```

Figure 26 Experiment 1 Number of Grids Search

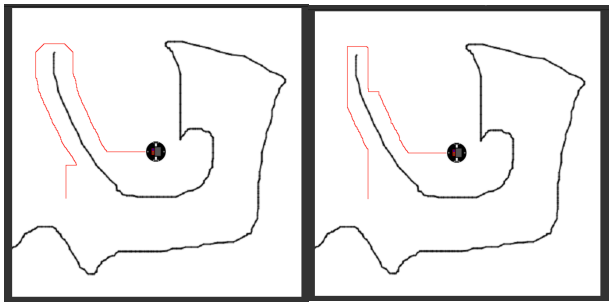
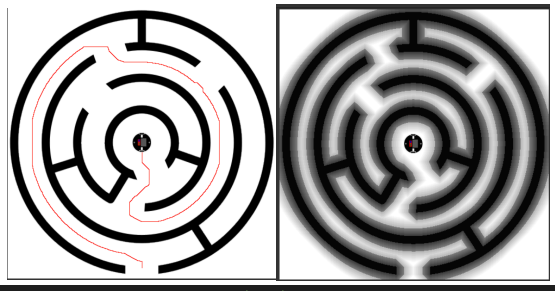


Figure 27 Experiment 1 U=1 vs U=10

The second experiment uses a custom maze map. This map represents complex obstacle with many false paths. Figure 28 shows the given map, the cost map, and the robots' starting and goal position. We begin by evaluating the number of grides the A* inspected. Without early exit the algorithm calculated 246130 grids cost. With early exit enabled the number of grids inspected was 234636. This represents only a 4% reduction of grid calculation the algorithm performed to solve the solution. This is likely due to the complexity of the map. With many false pathways the entire maze needs to be mostly solved before the exit is reached. Figure 29 shows the number of nodes searched

.

Next, we evaluated the effect of adjusting the heuristic scalar. With a default of u=1 the algorithm solved the path planner without any issue. We then adjust the scaler to 10 (U=10). As shown in figure 30 the pathway is prioritizing getting to the goal position more directly, however since the maze only allows for a narrow path to travel the change of u didn't not have much effect. Overall, the program was successfully able to develop the pathway for safe robot travel. No visible errors appear in the final map.



```
# Specify Starting Location (0,0)
self.set_start(world_x = 0, world_y = 0)
#Set Goal Location in World coordinate ...  Location will be updat
self.set_goal(world_x = 0.0, world_y = -230.0, world_theta = .0)
```

Figure 28

```
PATH TO GOAL FOUND
the number of nodes searched
246130
the number of nodes searched with Early Exit
234636
```

Figure 29 Experiment 2 Greyscale Map

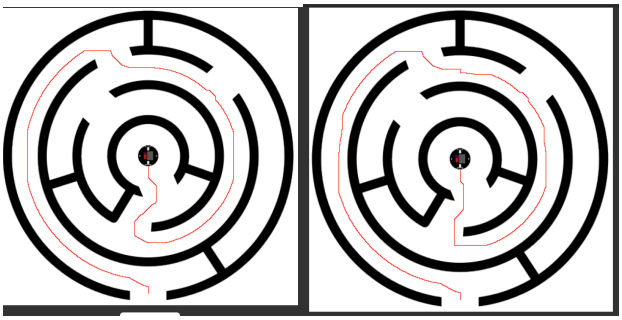


Figure 30 Experiment 2 Cost Map

The final experiment uses a custom map specifically deign to highlight the effect of the heuristic scaler. This map represents a structure whose shortest distance to the goal position has a higher cost compared to the safer path around the obstacle. Naturally the cost map will favor the safer path that is still relatively direct. But with a very high greedy factor the path planner should prioritize the shortest path. Figure 31 shows the robots' starting and goal position.

We begin by evaluating the number of grides the A* inspected. Without early exit the algorithm calculated 877022 grids cost. With early exit enabled the number of grids the A* algorithm needed to inspect before finding the solution was 388326. This represents a 55% reduction of grid calculation the algorithm performed to solve the solution. Figure 32 shows the number of nodes searched.

Next, we evaluated the effect of adjusting the heuristic scalar. With a default of u=1 the algorithm solved the path planner without any issue. As seen in figure 33 the path planner choose a path around the obstacle with the lowest cost and a higher prioritization to safety. We then adjust the heuristic scaler to 10 (U=10). As expected, the path planner now prioritizes a path most direct to the goal location. As shown in figure 33 the path selected goes between the two obstacles narrow channel. Overall, the program was successfully able to develop the pathway for safe robot travel. No visible errors appear in the final map.

```
# Specify Starting Location (0,0)
self.set_start(world_x = 0, world_y = 0)
#Set Goal Location in World coordinate ...  Location will be updated
self.set_goal(world_x = -190.0, world_y = 140.0, world_theta = .0)
```

Figure 31 Experiment 3 Cost Map

```
PATH TO GOAL FOUND
the number of nodes searched
877022
```

```
PATH TO GOAL FOUND
the number of nodes searched with Early Exit
388326
```

Figure 32 Experiment 3 Greyscale Map

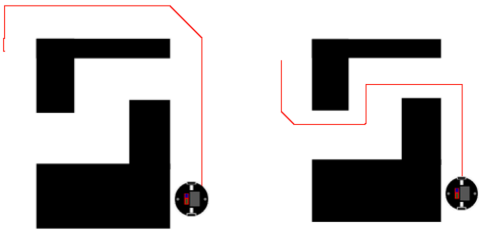


Figure 33 Experiment 3 Cost Map

VI. CONCLUSION

In modern robotics good path planning is critical for effective navigation. A robot following a poor path can cause unnecessary wasted travel time or even cause the robot to collide with its environment. The goal of this lab is to develop an algorithm capable of generating an optimal path to take the robot from point A to point B.

We implemented a custom A* algorithm to generate our paths. After the development of our custom algorithm, we then conducted three experiments to determine the effectiveness of the code. Each map tests the programs' with progressively more difficult environments. Each test was a success, and the algorithm was capable of correctly generate an effective pathway from start to goal that would allow the robot to transverse the space. Depending on the map we had up do a 50% reduction of the amount of nodes inspection compared to breath first search algorithm.

Overall, we are satisfied with the effectiveness and the level of versatility shown by our custom algorithm. The program has been tested and we believe it produces suitable data for use in future robot navigation tasks.

VII. ACKNOWLEDGEMENTS

None

VIII. REFERENCES

Jenkins, Jonathan and Erik Duque. "ME 5751 Module 1: Proportional Control for Point Tracking." 2022. *ME5751_Module1*. https://github.com/jonjenkins31/ME5751_Module1

Red Blob Games. *Introduction to the A* Algorithm*. 26 May 2014. 10 2022. https://www.redblobgames.com/pathfinding/a-star/introduction.html

Siegwart, Roland, Illah R Nourbakhsh and Davide Scaramuzza. *Introduction to Autonomous Mobile Robots*. Second Edition. Massachusetts Institute of Technology, 2011.