ME 5751 Module 2: Cost Map with Brush Fire Algorithm

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

***Abstract*— In this project we developed a custom python algorithm to take a user suppled grayscale map and generate an occupancy grid map, a brushfire map, and a cost map. The final cost map is then exported for later use in robot motion planning. 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 algorithm effectiveness in generating different grid maps and exporting a final cost map from a user supplied image. Each test of our algorithm was a success, and the algorithm was capable of correctly identify occupied spaces, develop a distance map, and output a cost map that would provide the needed information for future robot motion planning algorithms.**

***Keywords*—Motion Planning, Grid Map, Brushfire map, Cost Map**

I. INTRODUCTION

What does an autonomous car, Boston Dynamic's robot, and your rumba have in common? They all are robots capable of navigating in 3D environments. Navigation is a major focus in modern robotic. The ability for a robot to transverse in an unknown environment begins in its ability to generate reliable environmental maps.

The goal of this lab is to develop an algorithm capable of generating a variety of different maps. Our algorithm must generate an occupancy grid map, a brushfire distance map, and a potential cost map. Each type of map supplies the robot and user with specific data of the environment. After the development of our custom algorithm, we then conducted three experiments to determine the effectiveness of the system.

This paper will go into depth on different elements of this project including control theory, different map types, python programing, and evaluation of the final algorithm's ability to develop maps from user supplied environmental image.

II. BACKGROUND

Just like how maps are used to assist human navigation, maps are also used to assist robotic navigation. In robotics a map is a symbolic representation of selected characteristics of an environment. A wide variety of different maps are used to accumulate statistical evidence about the occupancy of a 3-D environment. As shown in figure 1 a map can be used to represent space.

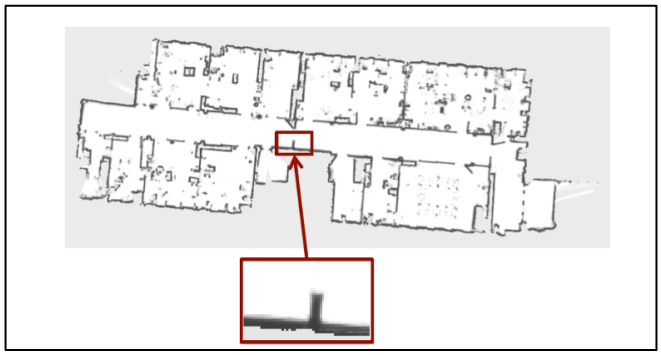

Figure 1 Maps

These volumetric maps are often referred to as grid maps because they are created from a series of pixels or grids. A map's grid size directly represents the map's resolution. With a larger pixel density, the map will contain a higher resolution. A general map consists of two types of pixels. A pixel that represents an occupied space often shown in black. And a pixel that represents a free space often represented in white, as shown in figure 2.

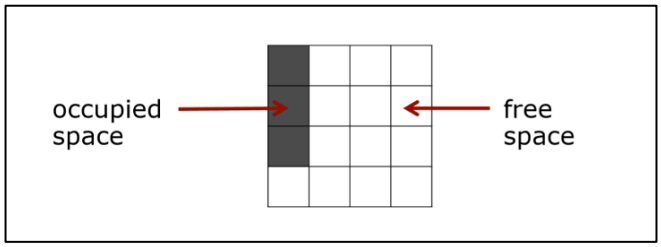

Figure 2 Position Map

As mentioned, a map can be used to represent a variety of information. In this project we are developing position maps, brushfire maps, and cost maps. A brushfire map uses an algorithm to approximate distance between occupied and unoccupied spaces. This algorithm produces a map whose unoccupied space are given a distance value to the nearest obstacle as shown in figure 3.

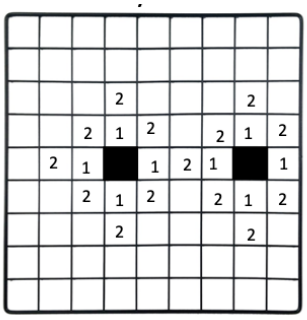

Figure 3 Distance map

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

$$d = |y_1 - y_0| + |x_1 - x_0| \tag{2.1}$$

The next map used in this project is the cost map or also known as a potential map. These 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 cost map is shown in figure 4.

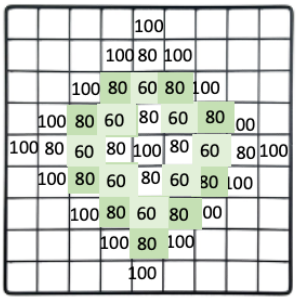

Figure 4 Cost Map

The cost map can determine the "risk" of a cell by a variety of methods. An exponential equation such as coulomb's law can be used to generate a cost map value, as shown in equation 2.2. Also, a linear equation as shown in 2.3 can be used to create a cost map value. The specific method depends on the users desired specification.

$$P = \frac{Kq_1}{r^2} \tag{2.2}$$

$$y = mx + b \quad m = \frac{y_2 - y_1}{x_2 - x_1} \tag{2.3}$$

III CONTROL DESIGN

The objective of our algorithm is to generate and export 3 types of grid maps, including an occupied map, distance map, and finally a cost map. To implement our custom algorithm onto our virtual robot we developed custom python code to derive each map in four steps.

We begin at step one which converts the given grey scale image to position map. This position map shows the location of occupied spaces and unoccupied spaces. Occupied spaces are assigned a value of 1, while unoccupied spaces are assigned a value of 0. To perform this operation the python scrip will store the greyscale image into a matrix called "grid". White pixel has a greyscale value of 255 and visually represents an unoccupied space. A black pixel has a greyscale value of 0 and visually represents an occupied space. The python scrip will index to each grid space and convert the greyscale value to a 0 or 1 to represent the grid occupancy. We implement a slight tolerance to allow grey spaces that isn't true black (0) to be considered an occupied space. The code used for step one can be seen in figure 5.

```
#SET 1
# Store costmap as grid
# Grey Scale  0 = black 255 = white
# Convert greyscale to object present/ nonpresent 1/0
# filter the grid and make islands a value of 1 and wa
grid=np.copy(self.costmap)              # Copy costmap
greyscale_tollerance = 50               # pixels might
grid[grid < greyscale_tollerance] = 1   # convert grid
grid[grid > greyscale_tollerance] = 0   # convert grid
#self.costmap=grid                      # CHECK OUTPUT
#print(grid)                            # CHECK OUTPUT
# #np.savetxt('Log/test.csv',grid, fmt='%-1d', delimit
```

Figure 5

Step 2 takes the information provided by the position map to create the brushfire map. The brushfire map, also known as the distance map, assign a value representing the distance that grid is to the nearest occupied space. To perform this task the python scrip begins by scanning the "grid" matrix to identify occupied spaces and stores the list into an array called "self.land". During this process the program also checks if the map contains only occupied or unoccupied spaces. If it does only contain one type the

program will output an error message informing the user. The code for this step is shown in figure 6.

```
#SET 2 A
# Using code created for hw2 . Run bushfire algorithun
# Make an array for Occupied and Unoccupied spaces
# # Initializing a array (queue)
self.land = []                                          #
self.water = []                                         #
# Identify Grid size
# A(i,j) = A(r,c)
self.row_length = len(grid)
self.col_length = len(grid[0])
 # Scan GRID Top Down -- LEFT to RIGHT
for r in range(self.row_length):
    for c in range(self.col_length):
        if grid[r,c] == 1:
            self.land.append((r,c))
        else:
            self.water.append((r,c))
 # Check to see if the grid contains atleast 1 land or
if  not self.land or not self.water:
    if  not self.land:
        Explination = "All cells are empty"
        print(Explination)                              #
    if  not self.water:
        Explination = "All cells are occupied"
        print(Explination)                              #
```

Figure 6

After scanning each grid space and forming a list of occupied spaces the program then uses then Manhattan equation to assign each unoccupied grid a distance value to the nearest occupied space. The program begins by indexing to the nearest unoccupied neighbor grid for each occupied space and assigning them a distance value of 1. This first layer of neighbors are then stored into a queue. During the next loop the queue of neighbors will be indexed one at a time and their unoccupied neighbors will be assign a distance value and be added to the next queue list. This process will continue till all unoccupied grid is assign a distance value. The code of this loop is shown in figure 7.

```
#SET 2 B
# Bushfire with 4 connection method.
#BUSHFIRE : INFLATION METHOD.... INDEX ONE NEIGHBOR AT A TIME AROUND THE ENTIRE LAND .
Bushfire=np.zeros((self.row_length,self.col_length), dtype=int)            # Bushfire
direction = ((-1, 0), (1, 0), (0, -1), (0, 1))                             # Left, To
distance = 1                                                               # Starting
bushfire_current_queue =self.land                                          # Take the
while bushfire_current_queue:                          # loop as long as a list of neighb
    bushfire_next_queue = []                           # clear the new found neighbors li
    while bushfire_current_queue :                     # FIND ALL THEN NEIGHBORS AND  SAV
        (r,c) = bushfire_current_queue.pop()           # get the next neighbor position
        for i ,j in direction:                         # INDEX TO ALL NEIGHBORS AROUND TH
            current_r=  r + i
            current_c=  c + j
            if 0 <= current_r < self.row_length and 0 <= current_c < self.col_length a
                #Bushfire[current_r][current_c] = distance
                Bushfire[current_r][current_c] = round(distance, 1)
                bushfire_next_queue.append((current_r, current_c))
    distance = distance+1                  # index the distance value for the next rou
    bushfire_current_queue  = bushfire_next_queue            # Save the list of new r
#self.costmap=Bushfire                      # CHECK OUTPUT DURING TESTING
#print(Bushfire)                            # CHECK OUTPUT DURING TESTING
#np.savetxt('Log/test.csv',Bushfire, fmt='%-1d', delimiter=',')     # CHECK OUTPUT DUR
```

Figure 7

The third step in the algorithm consist of converting the brushfire map into a cost map. We decided to make a two-zone algorithm to assign a gradient of risk to spaces in each zone. For both zones. the “risk” value of each grid decreases the farther the grid gets from the occupied space until it has the lowest risk value of 0. We used a linear equation as shown in figure 2.3 to assign the risk value to each cell. We wanted to keep a high-risk buffer around the occupied space where all the grids in this zone will be kept at a minimum high level. This zone is called our safety zone. The safety zone uses a linear equation with a specific slope to keep all the grids in the zone within the user defined parameters. These parameters include the minimum and maximum desired risk value, and the distance from the occupied space you want the safety zone to extend too.

The second zone uses a different linear equation to gradually decrease the risk starting at the edge of the safety zone down to zero. The rate of decrease can be specified by user parameter “cost_rate”. Once the cost is zero the robot will be far enough away from the wall that it unlikely to collide with the robot. The code used for this section is shown in figure 8.

```
#SET 3
# Convert the Bushfire map to a potential map / Cost map
# SPACES CLOSER TO AN OBJECT IS AT A HIGHER COST/VALUE
bushfire_costmap=Bushfire        # cost map ~~~~ THIS MATRIX REPRESENTS THE COST MAP ~~~ closer to object the
max_cost = 1000                                 # Specify the desired "cost" for an occupied space
max_cost_saftey_zone = 800                      # Specify the desired starting "cost" for any open space withi
min_cost_saftey_zone = 500                      # Specify the desired MINIMUM "cost" for any open space within
saftey_zone_distance = 12                       # Specify the desired saftey zone distance from known occupied
cost_rate_safty=(min_cost_saftey_zone-max_cost_saftey_zone)/(saftey_zone_distance)          # Specify the rate
cost_rate = 15                                                                             # Specify the rate
# Scan costmap Top Down -- LEFT to RIGHT
for r in range(self.row_length):
    for c in range(self.col_length):
        if bushfire_costmap[r,c] == 0:
            bushfire_costmap[r][c] = max_cost
        elif bushfire_costmap[r,c] < saftey_zone_distance :
            bushfire_costmap[r][c] = max_cost_saftey_zone+(cost_rate_safty)*bushfire_costmap[r,c]
        else:
            bushfire_costmap[r][c] = min_cost_saftey_zone+(-cost_rate)*bushfire_costmap[r,c]
            if bushfire_costmap[r][c]  < 0 :
                bushfire_costmap[r][c] = 0

# COSTMAP COMPLETED EXPORT RESULTS
self.costmap=bushfire_costmap          # Set the costmap pixel to be equal to our bushfire costmap
# self.costmap[200:250][0:-1]=0        #Set costmap pizel to a specific value Black
# self.costmap[300:350][0:-1]=250   #Set costmap pizel to a specific value white
# print(bushfire_costmap)              # CHECK OUTPUT DURING TESTING
# np.savetxt('Log/test.csv',bushfire_costmap, fmt='%-1d', delimiter=',')       # CHECK OUTPUT DURING TESTING
```

Figure 8

Finally, the program will export the calculated cost map and the program will display a visual representation of the cost map in the user interface. This code is shown in figure 9.

```
# COSTMAP COMPLETED EXPORT RESULTS
self.costmap=np.copy(bushfire_costmap)
# self.costmap[200:250][0:-1]=0          #Set costmap pizel to a specific value Black
# self.costmap[300:350][0:-1]=250   #Set costmap pizel to a specific value white
#self.costmap=bushfire_costmap          # Set the costmap pixel to be equal to our bushfi
#print(bushfire_costmap)                # CHECK OUTPUT DURING TESTING
np.savetxt('Log/test.csv',bushfire_costmap, fmt='%-1d', delimiter=',')      # CHECK OUTPU
```

Figure 9

IV. METHOD

After completing the development of our brushfire algorithm, we then moved to develop suitable challenges to test the program. These tests will require the program to calculate and export 3 types of grid maps. First, the user will supply the robot with a grayscale map as in figure 10. The python script with then convert the grayscale map to a position map. In the position map occupied spaces will be represented by a 1 and unoccupied spaces will be represented by a 0.

Figure 10 Initial Map

After the completion of the first step the program would use the information provided by the position map to create the brushfire map. Using the Manhattan equation each grid will be assign a value representing the distance that grid is to the nearest occupied space. An example of this step is shown in figure 11.

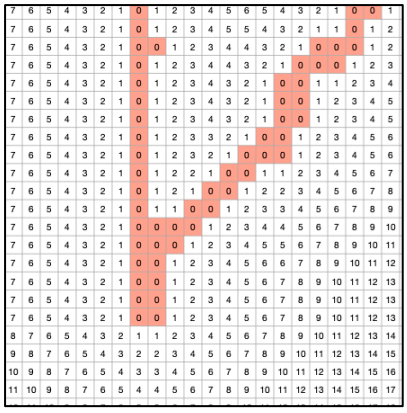

Figure 11 Brushfire Map

The third step in the experiment consist of converting the brushfire map into a cost map. As mentioned, before we have developed our own requirement to assign a cost to each grid depending on what zone the grid falls under and the distance that grid is from the nearest occupied space. The "risk" value of each grid decreases the farther the grid gets from the occupied space until it has the lowest risk value of 0. An example of this step is shown in figure 12.

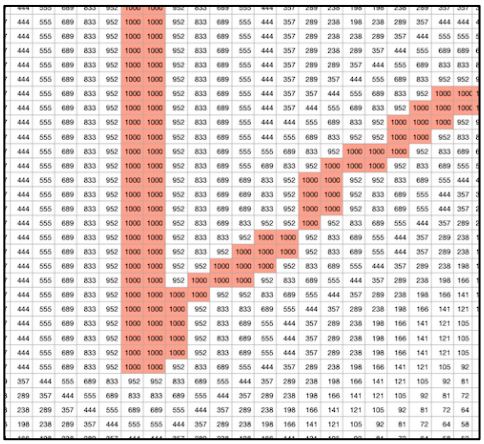

Figure 12 Cost Map

Finally, the program will export the calculated cost map and the program will display a visual representation of the zones in the user interface. An example of this visual overlay is shown in figure 13.

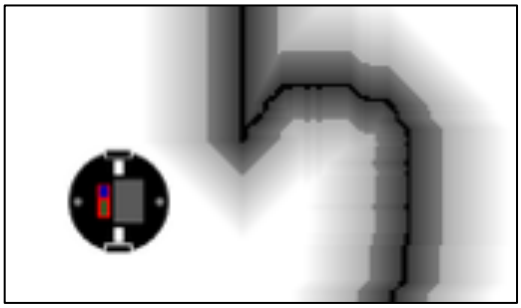

Figure 13 Exported Cost Map Overlay

Using this process, we will evaluate the effectiveness of the algorithm to calculate a cost map from a given greyscale map. The effectiveness will be dependent on how well the algorithm created this map without any signs of non-continuality or other visual defects.

V. EXPERIMENT RESULTS AND DISCUSSION

As mentioned, to determine the effectiveness of our brushfire algorithm we tested the program with three custom maps. Each test will require the program to successfully develop the occupancy map, brushfire map, and finally the cost map. Each test will use the same user parameters and are listed in figure 14.

```
greyscale_tollerance = 50
bushfire_costmap=Bushfire        # cost map ---- THIS MATRIX REPRESENTS THE COST MAP
max_cost = 1000                             # Specify the desired "cost" for an
max_cost_saftey_zone = 800                  # Specify the desired starting "cost
min_cost_saftey_zone = 500                  # Specify the desired MINIMUM "cost"
saftey_zone_distance = 12                   # Specify the desired saftey zone di
cost_rate_safty=(min_cost_saftey_zone-max_cost_saftey_zone)/(saftey_zone_distance)
cost_rate = 15
```

Figure 14 Program Constants for Experiment

The first experiment uses the professor's provided test map. This map represents an irregular, unstructured obstacle. The program algorithm will need to be able to detect the thin walls and extrapolate the inflation zone and the develop the cost map. Figure 15 shows the initial greyscale map and figure 16 shows the final cost map created by the software. The program was successfully able to develop all maps and identify the safe areas for robot travel. No visible errors appear in the final cost maps.

Figure 15 Experiment 1 Greyscale Map

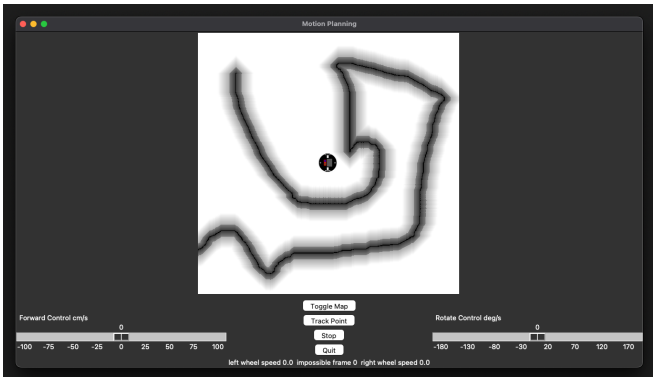

Figure 16 Experiment 1 Cost Map

The second experiment uses a custom map of two rooms. This map represents a simple wall dividing two rooms or two areas within a room. This scenario would often be found in vacuum robot applications. The program algorithm will need to be able to detect the thin walls and extrapolate the inflation zone and the develop the cost map. Figure 17 shows the initial greyscale map and figure 18 shows the final cost map created by the software. The robot successfully developed all required maps and identify the areas of travel to avoid while correctly keeping the open spaces of the room open for travel. No visible errors appear in the final cost maps.

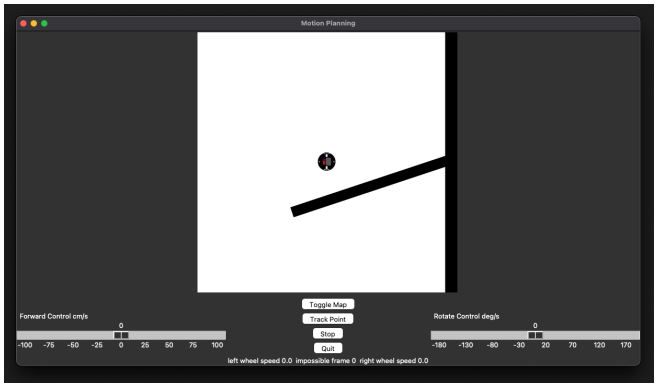

Figure 17 Experiment 2 Greyscale Map

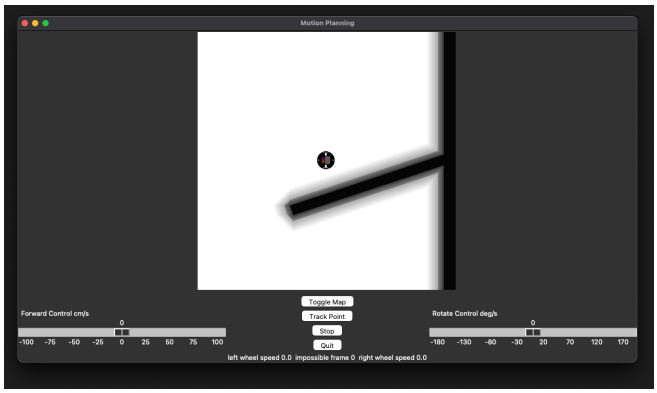

Figure 18 Experiment 2 Cost Map

The final experiment uses a custom maze map. This map will test the algorithm ability to solve a more complex and denser map. The algorithm will need to be able to detect the walls, calculate the inflation zone, develop the cost map while being conservative enough to not blackout potential robot pathways. Figure 19 shows the initial greyscale map and figure 20 shows the final cost map. The software was successfully able to create all maps while not over saturating the area with restricted zones. No visible errors appear in the final cost maps.

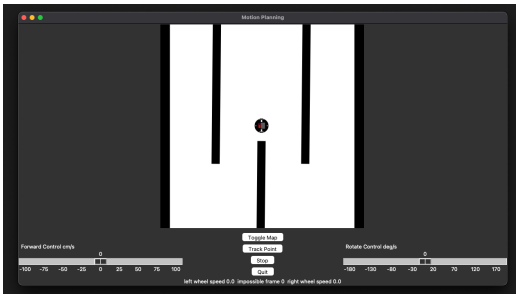

Figure 19 Experiment 3 Greyscale Map

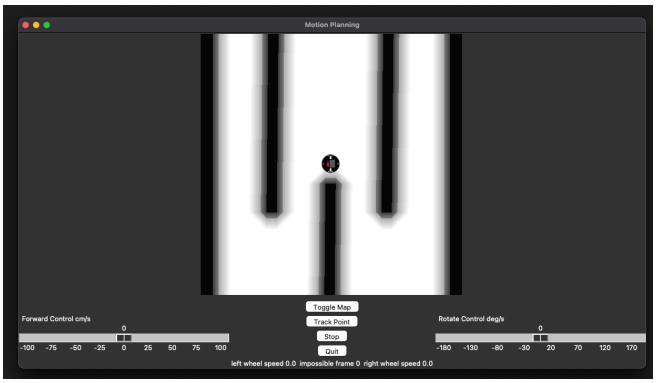

Figure 20 Experiment 3 Cost Map

VI. CONCLUSION

In modern robotics navigation is a major area of focus and the algorithms used to map environments are critical to robot's motion planning. The goal of this project was to develop an algorithm to that can generate brushfire and cost maps. Our custom brushfire algorithm uses 4-connection method to develop the distance map. We also implemented a two-zone linear equation to the cost map.

After the development of our algorithm, we then conducted three experiments to determine the effectiveness of the system. Each map tests the programs' with progressively more difficult environments. Each test was a success, and the algorithm was capable of correctly identify occupied spaces, develop a distance map, and output a cost map that would allow the robot to transverse the space.

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. "ME 5751 Module: Motion Planning." *ME5751_Module*, GitHub, 2022, https://github.com/jonjenkins31/ME5751_Module

Siegwart, Roland, et al. *Introduction to Autonomous Mobile Robots*. Second Edition ed., Massachusetts Institute of Technology, 2011.