ME5751 Lab 2 – Brushfire Algorithm for Costmap Generation for a Mobile Robot

Ryan Chao
Cal Poly Pomona
Pomona, CA
rchao@cpp.edu

Cameron R. Weigel
Cal Poly Pomona
Pomona, CA
crweigel@cpp.edu

***Abstract*— In this lab, potential maps of various, unique occupancy grids were generated via application of inflation, breadth-first search, and a brushfire algorithm. Several issues faced throughout the module were infinite looping and nonconforming costmapping resulting in jagged potential maps. However, the biggest concern was the program's efficiency for computing high resolution maps which ended up taking at least 30 seconds for maps of resolution 500x500. Nonetheless, our program was capable of producing smooth potential maps within reasonable times.**

Keywords— costmap; mapping; mobile robotics; robot motion planning; brushfire; breadth first search

I. INTRODUCTION

Costmap generation is the construction of a grid map where each cell is assigned a cost which is a value specifying the cell's distance from an obstacle. The higher the cost, the smaller the distance between the cell and an obstacle cell. In robotics motion planning, the cell represents the potential positions that can be traversed without any interference. The robot will act as a single point and the obstacles will be expanded or inflated to account for a robot's body. Since the obstacle cells already account for the robot's footprint, there must be a method for calculating or detecting traversable cells. A brushfire algorithm assigns cost values to each cell as a representation of their distance from the nearest obstacle cell. An 8-connection breadth-first search (BFS) function is utilized to assign costs to neighboring cells. The result of costmap generation via a brushfire algorithm is a map that looks similar to a physical elevation map. The costmap will be in grayscale with inflated obstacle cells in the darkest tone and any traversable cell in lighter tones. From this, the path or motion planning can be optimized

II. BACKGROUND

One critical component of robot motion planning is effectively mapping the robot's workspace. Robot motion planning requires a map that is divided into cells with cost values associated with each of those cells. Based on the cost of each cell, the robot determines the most efficient available path to take. The generation of these maps is a wide area of research and there are many methods to employ. In this paper, we have employed the brushfire algorithm method to generate our costmap based on the distance of cells to the distance of obstacles. We added onto the code from Lab 1 as this was the next step in building a complete mobile robot motion simulation. The costmap to the right in Fig 1 demonstrates a proper application of costmapping via brushfire algorithm. An important note to make is in the original GUI, the "Toggle Map" colors pixels opposite to the colors expected, for this reason, our function colors the costmap to comply with this inversion. There is a class file "cost_map_white.py" that will color the map in the opposite colors.

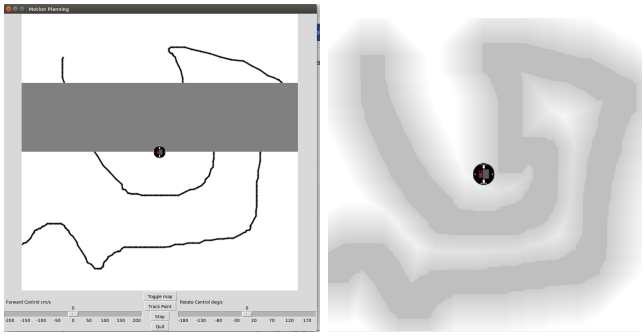

Fig 1. A wrong potential map (left) and a correct potential map (right).

III. ALGORITHMS

Utilizing a breadth first search on a 2D grid that represents the map of a robot's environment, we apply an inflation method to obstacle pixels in the first step. Pseudocode for the inflation method is listed below:

```
FOR Pixels in Map
        IF Pixel is black
                inflate pixel based on radius of robot
        ENDIF
ENDFOR
```

After inflation, we apply a brushfire algorithm on the edges of the obstacles. We use an 8-connection neighbor check to find the pixel's neighbors, if these pixels satisfy the conditions of being traversable pixels with a cost value associated with the closest obstacle pixel; we append them to the queue.

```
FOR black pixels in map
        check neighbors
        IF neighbor satisfies brushfire conditions:
                enqueue pixel
                color costmap pixel
        ENDIF
        WHILE queue is not empty:
                dequeue element
                check element's neighbors
                IF neighbor satisfies brushfire cond.:
                        color costmap pixel
                        add neighbor to queue
                ENDIF
        ENDWHILE
```

The distance map of the given occupancy grid is then established using the Manhattan distance as follows:

$$d = |y1 - y0| + |x1 - x0|$$

This method solves for distance as the sum of the absolute difference of two points in Cartesian coordinates.

IV. METHODS

Given a map of size *m x n*, we employ a costmap generation algorithm based on the brushfire algorithm to label each cell with a cost value and to determine collision locations (i.e. walls) based on pixel coloring. Using a grayscale map, we determine black or near-black pixels to be considered as obstacles and all else considered to be traversable areas. Based on a mobile robot's footprint, we inflate the obstacles by the robot's minimum circumscribed circle radius to avoid any possibility of collision. After inflation, we apply a brushfire algorithm to calculate cost values of each cell based on their distance to the nearest obstacle cell. Utilizing the breadth-first search with an 8-connection neighbor function shown in Fig 3 and Fig 4. We use the queue data structure to minimize operations on the costmap in order to reduce the number of operations to fully compute the costmap. Although computation time is reduced, it still requires an extensive amount of time for the program to solve for a very large potential map.

```
# Use black pixels as a start point
for x in range(m-1):
    for y in range(n-1):
        if self.costmap[x,y] == 255:
            black_pix.append([x,y])

# Perform brushfire from unfilled pixels, starting at a black pixel
for pix in black_pix:
    pix_neighbors = self.get_neighbors(pix, m, n)

    for pix_neighbor in pix_neighbors:
        if self.costmap[pix_neighbor[0], pix_neighbor[1]] != 255:
            q.append(pix_neighbor)
    while len(q)>0:
        node = q.pop(0)
        dist = abs(pix[0]-node[0]) + abs(pix[1]-node[1])
        # Cost is our potential function, in this case it is a simple linear potential
        cost = 200- cost_const*dist
        if cost < 0:
            cost = 0

        if self.costmap[node[0],node[1]] != 255 and self.costmap[node[0],node[1]] < cost:
            self.costmap[node[0],node[1]] = cost


        neighbors = self.get_neighbors(node, m, n)

        for neighbor in neighbors:
            dist = abs(pix[0]-neighbor[0]) + abs(pix[1]-neighbor[1])
            cost = 200- cost_const*dist
            if cost < 0:
                cost = 0

            # Only assign cost if the pixel isn't black and if its cost value corresponds to the minimum distance obstacle pixel
            if self.costmap[neighbor[0],neighbor[1]] != 255 and self.costmap[neighbor[0],neighbor[1]] < cost:
                self.costmap[neighbor[0],neighbor[1]] = cost
                if neighbor not in q:
                    q.append(neighbor)
```

Fig 3. Breadth-first search brushfire algorithm

```
# Get neighbors, including diagonals
def get_neighbors(self,tile, m,n):
    #m, n = np.shape(grid)
    #print("tile ", tile)
    neighbors = []
    # Left
    if (tile[0]-1 >= 0):
        neighbors.append([tile[0]-1,tile[1]])
    # Bottom Left
    if (tile[0]+1 <= m-1 and tile[1]-1 >=0):
        neighbors.append([tile[0]+1, tile[1]-1])
    # Down
    if (tile[1]+1<=n-1):
        neighbors.append([tile[0],tile[1]+1])
    # Bottom Right
    if (tile[0]+1  <=m-1 and tile[1]+1 <=n-1):
        neighbors.append([tile[0]+1, tile[1]+1])
    # Right
    if (tile[0]+1<=m-1):
        neighbors.append([tile[0]+1,tile[1]])
    # Top Right
    if (tile[0]-1 >=0 and tile[1]+1 <=n-1):
        neighbors.append([tile[0]-1, tile[1]+1])
    # Up
    if (tile[1]-1 >= 0):
        neighbors.append([tile[0],tile[1]-1])
    # Top Left
    if (tile[0]-1 >=0 and tile[1]-1 >=0):
        neighbors.append([tile[0]-1, tile[1]-1])

    return neighbors
```

Fig 4. 8-Connection Neighbor Search function.

V. RESULTS AND DISCUSSION

The resultant potential map of three unique occupancy grids will be discussed regarding any overlap errors, computation time, and the robot's ideal path. It should be noted there is an emphasis on the obstacle inflation. The immediate, neighboring traversable cells start at a cost value of 200 simply for visual means. The cost constant used by default is 3, so each cost has a difference of 3 between one another. This was chosen due as 500x500 resolution maps were computed within reasonable time.

A. Map Comparison

In Fig 5, the occupancy map and its associated costmap shown below were generated using the algorithm described in the paper. This map is 150x150 pixels in resolution and took approximately 4.72 seconds to generate. Test map 1 serves as a sample to test the program's performance and whether it outputs as expected which it did. This can be seen as cells further away from the obstacle are lighter than those closer. In terms of motion planning, the robot would simply be stuck if spawned inside the maze since the gaps between each obstacle wall is too small for the robot to pass through.

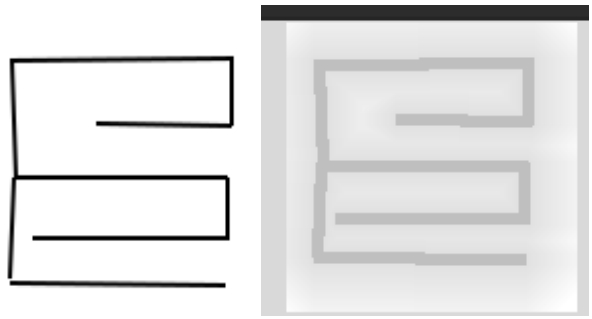

Fig 5. Test Map 1, 150x150, occupancy grid (left) and costmap (right).

In Fig 6, a 500x500 resolution occupancy map is translated to its potential map. The costmap was generated in roughly 32.19 seconds which fares well when compared to that of Test Map 1. The expected time to generate a map of this resolution was to be roughly 11 times of Test Map 1's time which is the ratio of the Test Map 2's size relative to Test Map 1's . However, this proved to be wrong as the program finished within 7 times of Test Map 1's time. The creation of this map was to observe whether or not there would be any computational errors when two walls collide. As shown in the

potential map, there are no issues when it comes to obstacles being too close to one another. This is due to our algorithm accounting for any cost values already present in nearby cells. In this scenario, the robot should ideally follow the path to the right though in a radical case, it may go through the slit in the path to the left.

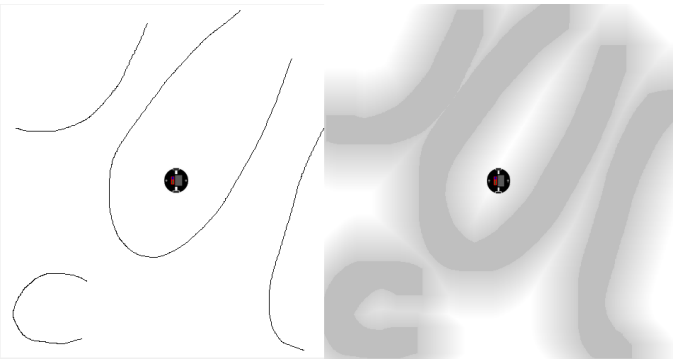

Fig 6. Test Map 2,500x500, occupancy grid (left) and costmap (right).

In Fig 7, another 500x500 resolution occupancy map is used to compute its potential map. This costmap was generated in roughly 37.80 seconds which is close to that of Test Map 2. This is as expected since this grid contains more black pixels, it would require a higher computation time. There are no overlaps in this trial, but there are very thin gaps that offer the robot some movement options. The purpose of this map is to evaluate the robot's ability to either perform a long, yet safe run or a short, risky run. This test would be suitable in real life applications due to basic robots experiencing difficulty in navigating completely straight due to mechanical deviations such as wheel angle and rotation. However, for a robot that could be considered as a single point and in simulation, it would easily maneuver through the slits since the robot's footprint is already considered and any obstacle collision would be avoided.

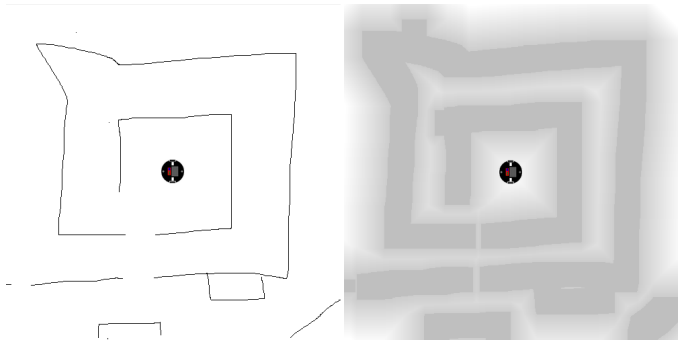

Fig 7. Test Map 3,500x500, occupancy grid (left) and costmap (right).

Fig 8 displays how a proper potential map should look with no gradient gaps between traversable and obstacle cells. It is visually difficult to identify where the inflated obstacle cells are which is why we ran test maps with a gradient gap.

Fig 8. Test Map 3, 500x500, costmap with higher accuracy.

B. Observations

Comparing the results of Test Map 1 and Test Map 2, we can quickly identify computation time is dependent on resolution. However, this time does not have a linear relationship with the resolution size of an occupancy grid. Computation time is affected by the amount of black pixels or obstacles in a grid. The more obstacles there are, the more time is needed for the program to solve for the potential map. When testing the original occupancy map, a cost constant of 1 required 2 minutes and 23 seconds to load while a cost constant of 3 required

35.61 seconds. The cost constant is shown to have an inverse relationship with time since time increases as cost constant decreases. This is due to the distance map requiring more calculations, but yields a more accurate potential map.

VI. CONCLUSION

From this module, we investigated costmap generation through the use of a brushfire algorithm. The program successfully implemented said method without any computational issues. For smaller map resolutions like a 150x150 map, the computation time was 4.72 seconds while for 500x500 maps, it required up to 37.80 seconds to load based on the self made occupancy grids. Adjusting the cost constant to 3 from 1 reduced the computation time by at least 70% as it reduced the number of cells to be evaluated.

Due to difficulty in differentiating single tone changes with the naked eye, a gradient gap was used to identify the boundaries of inflated obstacles. Ideal settings consist of a cost constant of 1 and initializing cost values at 254 to construct a smoother costmap.

In order to improve costmap generation, there may exist better methods to reduce computation time either through removing unnecessary checks or utilizing better coding commands. Potential functions of higher order may also be utilized to make use of things such as gradient descent for improved robot path planning efficiency.

VII. REFERENCES

[1] R. Siegwart, I. Nourbakhsh, D. Scaramuzza, Introduction to Autonomous Mobile Robots, 2nd ed., The MIT Press : Cambridge, 2011.

[2] Y Chang, California Polytechnic University, ME5751 Lecture Slides, Fall 2022