

Autonomous Robotics

Lab Report 1 - Potential Functions

Anirudh Puligandla

March 5, 2017

1 Introduction

Autonomous Robotics is increasingly being used in many robotics applications. Path Planning algorithms are the backbone for autonomous navigation. The goal of this lab is to design, implement and test two path planning algorithms, namely, 'brushfire' and 'wavefront algorithms. The goal of each of the algorithms is to generate a potential map from the given maps containing 1s representing obstacles and 0s representing free space. The algorithm is supposed to assign potential values to all the zeroes in increasing order from the obstacles or the goal position depending on the algorithm. The following sections briefly describe the working of each of the algorithm, implementation and the results obtained in MATLAB.

2 Brushfire Algorithm

The aim of this section is to implement the Brushfire algorithm on MATLAB and observe the results. Brushfire algorithm generates repulsive potential from the obstacles, using 8-neighbourhood connectivity. The algorithm was written as a matlab function which be called in the following way, where the function returns the output *value_map* containing the final potential map:

$$\text{function}[\text{value_map}] = \text{brushfire}(\text{map})$$

The function when called generates the following shown map, where the values represent the values in the matrix *value_map* for each index (i, j) . The algorithm starts iterations from the obstacles and assinging the next potential value to the immediate 8-neighbour connectivity pixels. usually, for 14x20 map, the algorithm terminates within 4 iterations

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	2	2	2	2	2	1	1	2	2	2	2	2	2	1	1	1	2	2	2
1	2	3	3	3	2	1	1	2	3	3	3	3	2	1	1	2	3	2	1
1	2	3	4	3	2	2	2	2	3	4	4	3	2	1	1	2	3	2	1
1	2	3	3	3	3	3	3	3	3	4	3	2	1	1	1	2	3	2	1
1	2	3	2	2	2	2	2	2	2	3	4	3	2	1	1	2	3	2	1
1	2	3	2	1	1	1	1	1	2	3	4	3	2	1	1	1	2	3	2
1	2	3	2	1	1	1	1	1	2	3	4	3	2	2	2	2	3	2	1
1	2	3	2	2	2	2	1	1	2	3	4	3	3	3	3	3	3	2	1
1	2	3	3	3	2	1	1	2	3	3	3	2	2	2	2	3	3	2	1
1	2	3	4	4	3	2	2	2	2	3	3	2	2	1	1	2	3	3	2
1	2	3	3	3	3	3	3	3	3	3	2	2	1	1	1	2	3	3	2
1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	2	2	2	2
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 1: *value_map* generated by brushfire algorithm

and within a second of computational time. The program iterates over the map until all the values in the map are non-zero. This is achieved by using the matlab function *nnr()* that returns the number of non-zero elements in the matrix.

3 Wavefront Planner Algorithm

The wavefront planner algorithm on the contrary starts generating potentials from the goal position instead of from the obstacles. In this way the goal position has always the least possible potential value of 2. This makes the trajectory generation from the start to the goal position, much easier as the path is planned starting from the start position, with very high potential value towards the least potential value connecting all the intermediate values in the neighbourhood until the goal. This algorithm comparatively takes more number of iterations than the brushfire algorithm as the operation starts with only one index in the beginning while expanding the queue containing the unprocessed indices with each iteration. For the typical 14x20 map, the wavefront planner converges within about 20 iterations.

The goal of this section was to implement the wavefront planner algorithm as a MATLAB function in the following way:

```
function [value_map, trajectory] =
wavefront(map, [start_row, start_column], [goal_row, goal_column])
```

The variable names used for the function clearly indicate their usage. The code has been implemented using a while loop which executes until all the elements in the map

matrix have non zero values, i.e, all the elements are assigned a potential value, the same way as mentioned above. The working of the algorithm is shown in the following steps:

- start.
- get the dimensions $[r, c]$ of the matrix map and assign $value_map([goal_x, goal_y]) = 2$.
- iterate on the condition $nnz(map) = r \times c$ i.e, until all the values are non-zero.
- find the *indices* of elements associated with *value* which is 2 for the first iteration.
- get all the neighbours for each entry of *indices* and assign the zero values with the updated potential *value*.
- increment *value* and continue step 3.
- end.

Finding the indices of elements associated with the current potential value is done by using the *find()* function in MATLAB. The results for this implementation for all the given maps are shown below.

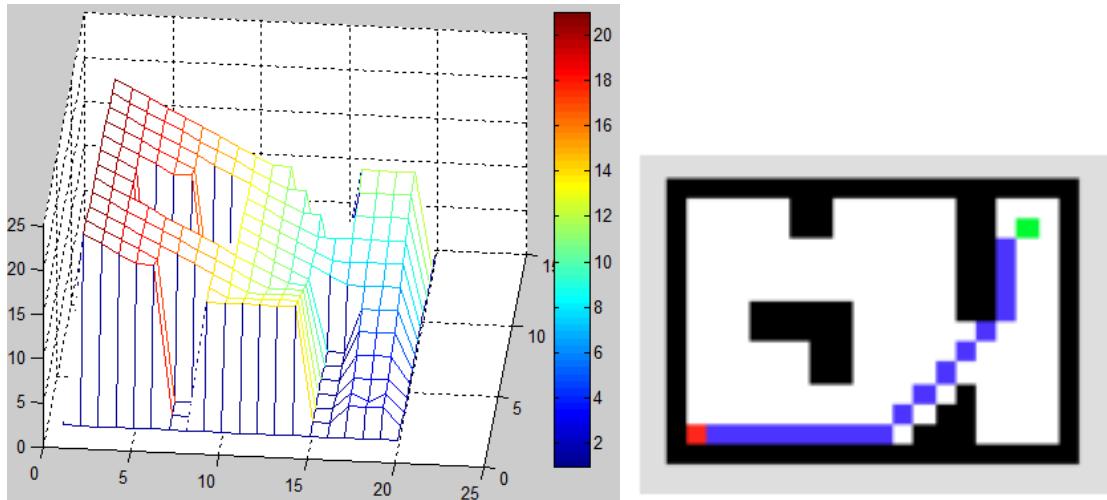


Figure 2: From Left: (a)3D potential map; (b)Trajectory of small map with start=[13,2] and goal=[3,18]

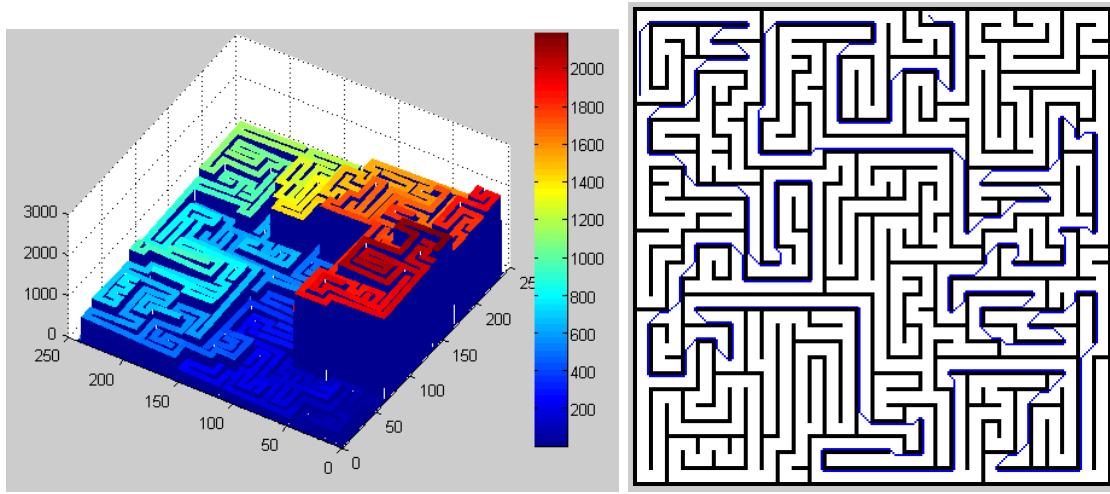


Figure 3: From Left: (a)3D potential map for maze; (b)Trajectory of for maze with start=[45,4] and goal=[5,150]

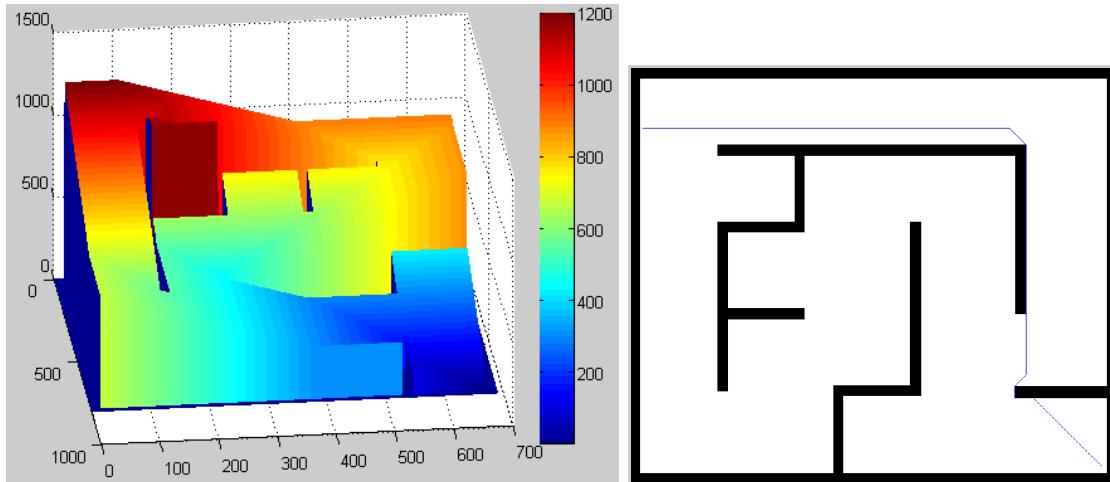


Figure 4: From Left: (a)3D potential map for maze_big; (b)Trajectory of for maze_big with start=[100,20] and goal=[660,780]

As seen from the results, the algorithm provides the potential map and the trajectory for the given initial and final positions. It can be argued that the algorithm is well optimized as it works fine for the bigger maps also.

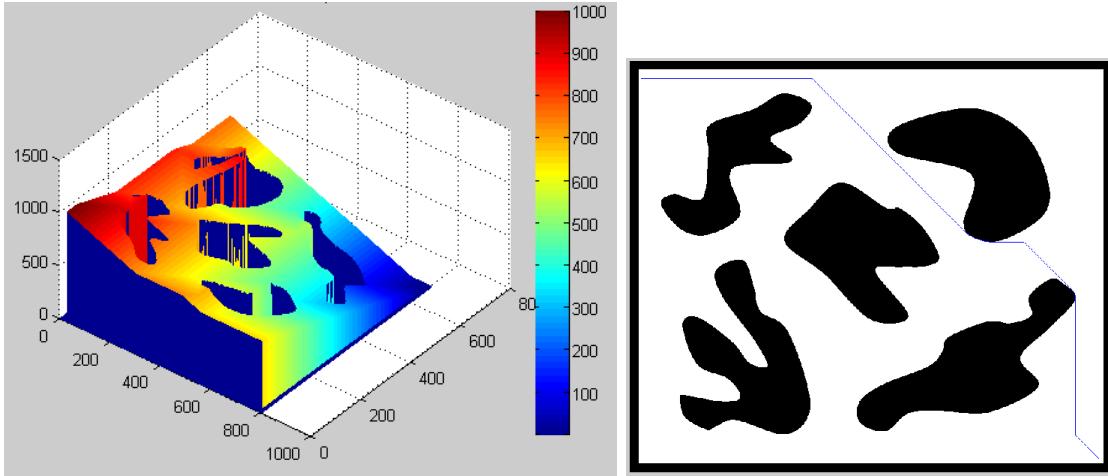


Figure 5: From Left: (a)3D potential map for obstacles_big; (b)Trajectory of for obstacles_big with start=[30,20] and goal=[660,780]

4 Problems Faced

There were a few issues while implementing the algorithms:

1. Firstly, there was one zero in the maze_big map in the first index from top left. Since the algorithm is designed in such a way that it doesn't iterate over the elements with potential value 1 and also, it iterates over the map until there are no zeros left in the map, this resulted in an infinite loop. The zero had to be changed to 1. But this approach also has an advantage that the program does not iterate over the obstacles, thus, saving time.
2. secondly, the *find()* function used in the code iterates over the entire map for each iteration of the while loop, thus increasing the computational time. The time consumption increases multi-fold for large maps where the no. of iterations could go upto a few thousands.

This can be avoided by using queue like data structures where we would save all the unprocessed elements with the potential value 0 and iterate over the queue until it is not empty. This approach has been implemented in the *wavefront1* function provided in the .zip file.

5 Conclusion

Both the *Brushfire* and *wavefront* algorithms were correctly implemented on MATLAB and the code was optimized to make it work on bigger maps as well. All the results are clearly shown. Lastly, a few of the problems were discussed and an alternative solution is proposed for faced problems.