Autonomous Robots
Potential Functions - Wavefront planner

Pramita Winata
Master VIBOT

***Abstract*—Many path planning algortihms have been invented nowadays. In this lab, one of the potential function algorithm, the wavefront planner, will be implemented.**

***Index Terms*—Potential functions, Wavefront Planner, Path Planning, Autonomous Robots**

I. Introduction

Potential function is used to move the robot from high potential (starting point) to a lower potential point (goal point). In general case, it is built using:

$$U(q) = U_{att}(q) + U_{rep}(q) \tag{1}$$

The gradient is used to know which direction to follow.

II. Wavefront planner

In this lab, we will implement a planner based on the brushfire algorithm, wavefront planner. Gradiend descent indicates the direction to go. It will be computed by choosing the neighbour with the highest difference.

$$\Delta U = max(center.weight - neighboors.weight,' first') \tag{2}$$

The neighborhood adjency that will be used is the 8-point connectivity. Since we know that the diagonal movement will be longer, the 4-point-connectivity (up, down, left, right) will be prioritized than the diagonal connectivity.

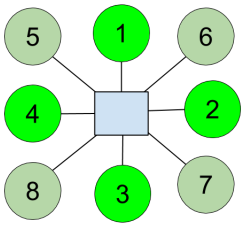

Fig. 1: Neighborhood exploration

$MATLAB$ function implemented for this can be found in:

```
function neighbors = get_neighboors(r, c)
```

Implementation of the wavefront planner will be divided into 4 main sections. The following sections will explain them in details.

A. Construction of weighted map

Constructing the weighted map will work similiar with region growing algorithm. We will start from the goal position and then evaluate the neighborhoods points. The order of the evaluation needs to be remembered as the next exploration will follow the same sequence of the evaluation.
To implement this $Queue$ concept is utilized. There will be 2 pointers used in the $Queue$. The first pointer will keep track the evaluation sequence and the second one will be used to keep track the last element in the queue. Then the $Queue$ will be evaluated with *First In First Out* rule to preserve the evaluation sequence.
New weight that will be assigned to the corresponds neighboors is the by one increment of the center node. This process will be iterated until the goal is found.
Algorithm 1 shows the pseudocode described earlier.

Algorithm 1 Constructing weighted map

```
1:  procedure CONSTRUCT_WEIGHTED_MAP(map, start)
2:      initate flag matrix for weighted map
3:      queue ← [goal]
4:      while not empty queue do
5:          point ← queue.pop
6:          neighboors ← 8 − connected − points
7:          center.weight ← map(point).weight
8:          for all neighboors do
9:              if neighboor NOT evaluated then
10:                 queue.stack ← neighboor
11:                 neighboor.weight ← center.weight + 1
12:             end if
13:         end for
14:     end while
15: end procedure
```

$MATLAB$ function implemented for this can be found in:

```
function map = build_new_map(map, goal_point )
```

B. Construction of trajectory

For obtaining the trajectory, we are going to move from the starting point to the lower weight between the center point and

it's neighborhors; gradient descent. Gradient descent indicates the direction to go. An illustration in Figure 2 shows the flow of calculating the gradient descent for this implementation.

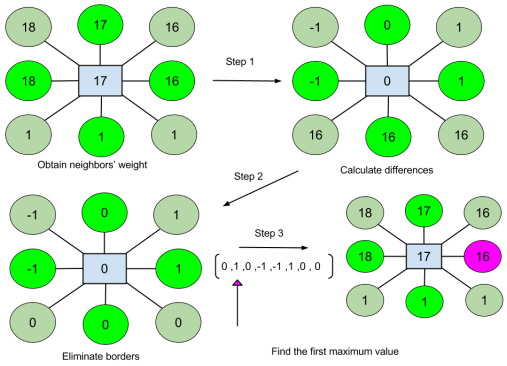

Fig. 2: Constructing trajectory

We will store every direction sequentialy to form a path. This path will be the trajectory to reach the goal. Algorithm 2 shows the pseudocode described earlier.

Algorithm 2 Constructing trajectory

procedure CONSTRUCT_TRAJECTORY($map, start$)
  $initate\ path\ matrix$
  $path \leftarrow [start]$
  $current.weight = start.weight$
  while $goal$**NOT** $found$ **do**
    $neighboors \leftarrow 8 - connected - points$
    for all $neighboors$ **do**
      $storeneighbors'weight$
    end for
    $gradient\_descent = current.weight - neighbors.weight$
    $excludethebordersfromthedifference$
    $next\_path = max(gradient_descent,' first')$
    $path.stack \leftarrow next\_path$
  end while
end procedure

$MATLAB$ function implemented for this can be found in:

```
function trajectory = compute_trajectory( new_map, start_row, start_column)
```

C. No solution environment

We need to consider the possibility of the no solution environments when implementing path planning algorithm. Several cases where the algorithm will produce no solutions outputs are:

1) Specifying the start point to be outside the map
   Error message will be displayed if user gives invalid parameters.

```
Start point must be inside the map.
Please specify a point within [14..20].'].
```

2) Goal is not specified This is to catter the case of invalid map. The map needs to have goal; point with value is 2. Error message will be displayed if user gives invalid parameters.

```
Goal cannot be found. Please specify
a point with value 2 in the map.
```

3) Multiple goal are detected Error message will be displayed if user gives invalid parameters.

```
There are multiple goals. Please specify
only one point with value 2 in the map.
```

4) Closed room case This is the last case checked after all of the other cases are validated. It is the case where the goal is in a different closed space than the starting point, Figure

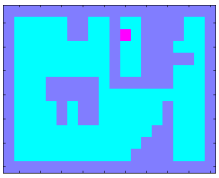

Fig. 3: Example of closed room case

After constructing the weighted map, we can observe the value in the starting point. If the starting point is not processed (weight = 0), then there is not needed to construct the trajectory for the map since it is not reached by the wave.

D. Plotting the map and trajectory

In this step, we mainly utilized $MATLAB$ default functions. Functions are $imagesc$, $colormap$, and $plot$. $MATLAB$ function implemented for this can be found in:

```
function plot_map(map, trajectory )
```

III. Result

- Small map

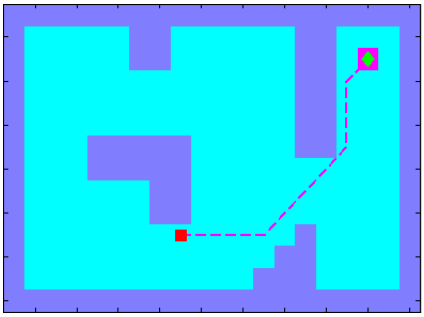

Fig. 4: Result of small map

- Big Map

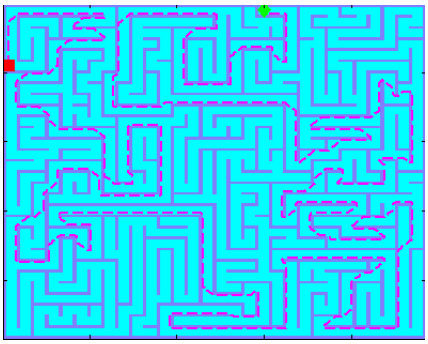

Fig. 5: Result of big map

- Other maze

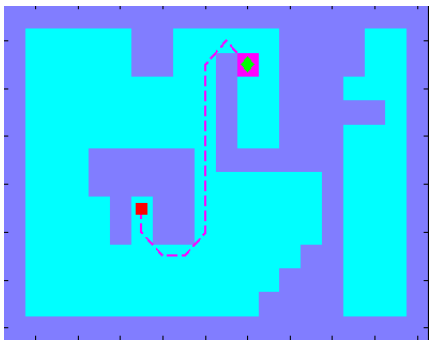

Fig. 6: Result of other maze

- No solution

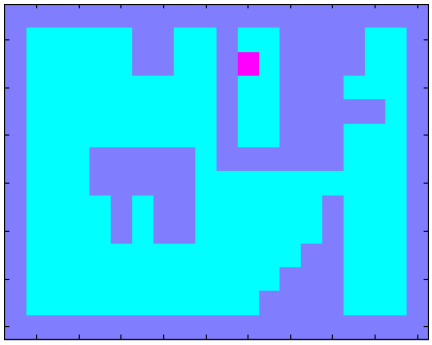

Fig. 7: Result of no solution maze

Computation time is shown in Table I.

Maze	Time(seconds)
Small maze	0.045875
Big maze	0.490855
Other maze	0.039864

TABLE I: Computation time

IV. Conclusions

In this lab, wavefront planner has been implemented and tested in several environments. The implementation can be considered simple and very straight-forward. In terms of execution time, it provides a good result which makes this algorithm favorable in the future.x