Autonomous Robotics Lab 1 Brushfire and Wavefront

Rodrigo Caye Daudt

I. Introduction

THIS report describes implementations in MATLAB for the Brushfire and Wavefront algorithms developed for the Autonomous Robotics module. These algorithms are used for path planning based on a given map. Sections II and III describe the implementations and results of the Brushfire and Wavefront algorithms, respectively, Section IV contains an analysis of the obtained results, and Section V concludes this work.

II. Brushfire

The Brushfire algorithm serves to create a potential function that guides a robot away from the obstacles. Starting from the walls and objects, which are initialised with value 1, the algorithm spreads from these starting points (in this case using 8-neighbourhood) in generations until all of the map has been covered. In other words, the Brushfire algorithm calculates the small 8-neighbourhood distance to an obstacle for every point in the map. A parallel can be made to the way fire spreads in the real world. It is important to note that this algorithm assign high values to points distant to obstacles, not low values.

The Brushfire algorithm was implemented to work in generations of points. Instead of keeping one single queue of points to be analysed next, the function keeps two queues: one for the points in the current generation (all of which have the same distance from obstacles), and one for the points that were added during the current generation and that will be processed on the next batch. This procedure leads to a resizing of smaller MATLAB arrays, which is faster since it requires smaller memory allocations. The algorithm stops when a generation ends and the list of points to be processed is empty. The code for *brushfire.m* can be found in Appendix A.

Figures II-II contain the results of the Brushfire algorithm over the four provided maps displayed using a jet colour map (low values in blue, high values in dark red). It can be observed that the walls and obstacles are displayed in blue and the temperature of the color increases as the distance from the walls and object increases, which was expected since the smallest labels (1) are assigned to the fixed objects and progressively larger are assigned to points farther from these objects. The computational times will be discussed in section IV.

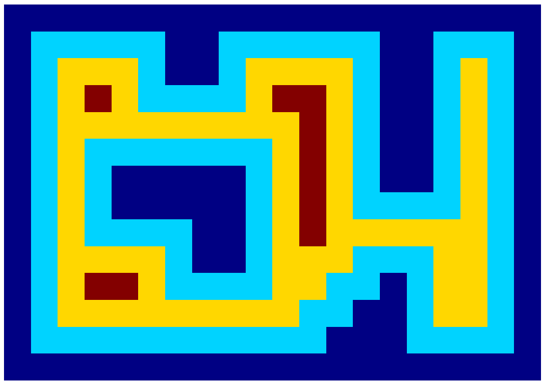
Fig. 1. Brushfire - basic.mat

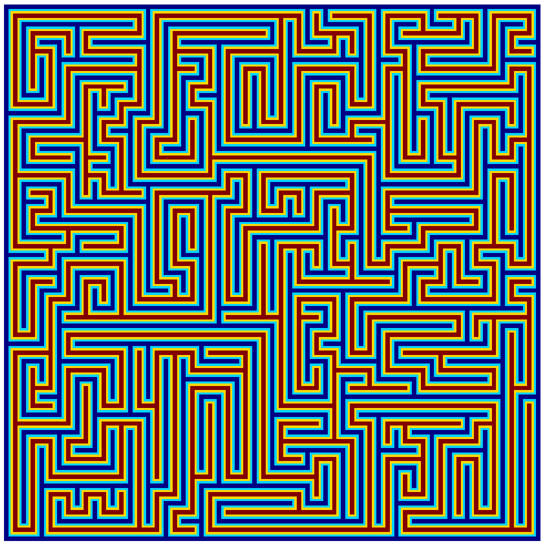
Fig. 2. Brushfire - maze.mat

III. Wavefront

The Wavefront algorithm's purpose is to find a path from a given starting point in the map to a given goal. It works in a similar way to the Brushfire algorithm, but instead of expanding from the walls and obstacles, it starts from the goal point and expands using the 8-neighbourhood of each pixel until it reaches every point in the map. It is important to notice that even though the starting point is not part of an obstacle, the waves cannot pass through the fixed objects defined in the map.

Given the similarities between the Wavefront and the Brushfire algorithms, their implementations were very similar. That means the Wavefront was implemented using the same generational principle and for the same reasons as described in Section II.

It was also required to use the *wavefront* function to calculate one of the possible good paths from the chosen

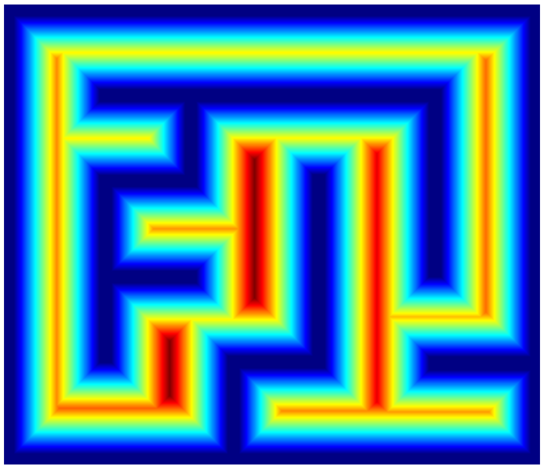

Fig. 3. Brushfire - mazeBig.mat

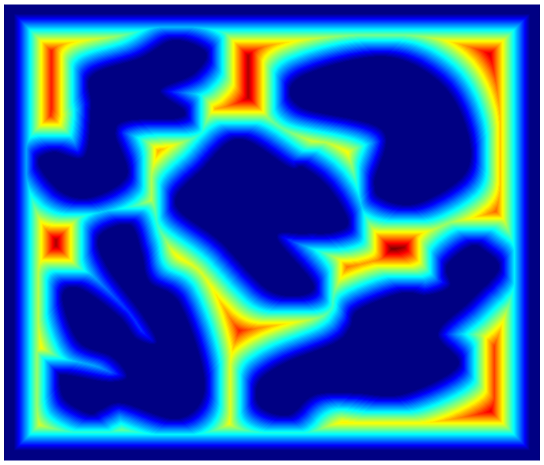

Fig. 4. Brushfire - obstaclesBig.mat

starting point to the goal point. This was made by creating a sequential list of points where each point had an associated value smaller by 1 relative to the previous entry. Priority was given to the cross-neighbours (4-neighbourhood), since they yield a smaller distance than the diagonal neighbours. This was done by checking the cross-neighbours before the diagonals, and picking the first point with an associated wavefront value smaller than the current value. The code for *wavefront.m* can be found in Appendix B.

Figures III-III show the results of the *wavefront* function over the four provided maps, with an overlaid trajectory for example starting points and goals. The maps are displayed with a parula colour map (blue for low values, yellow for high values), the calculated trajectory is displayed in red, a black dot marks the starting point and a yellow dot marks the goal. We can see this algorithm allows us to find a smart trajectory from the starting points to the goals as long as the two points can be connected. This algorithm is very robust and works in any type of environment, independently of the shapes of the objects in the map. It can also be observed that horizontal and vertical directions are prioritized, and diagonal directions are chosen only when the Carthesian directions don't provide any steps that decrease the distance to the goal.

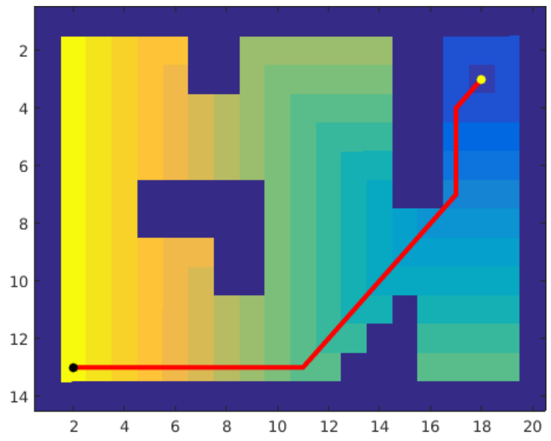

Fig. 5. Wavefront - basic.mat

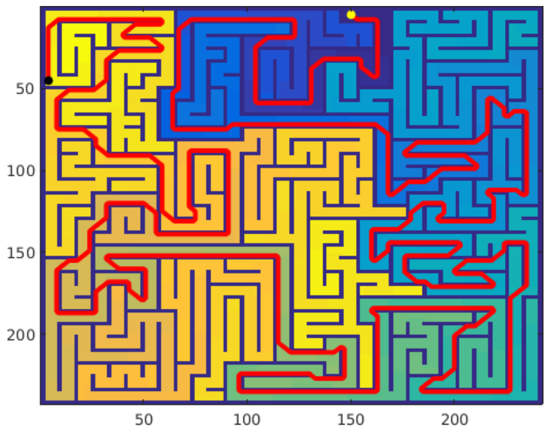

Fig. 6. Wavefront - maze.mat

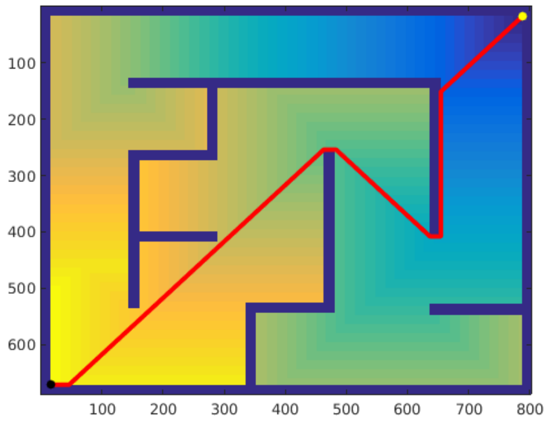

Fig. 7. Wavefront - mazeBig.mat

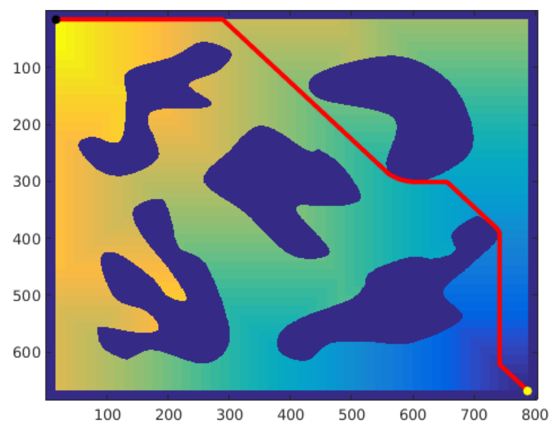

Fig. 8. Wavefront - obstaclesBig.mat

IV. Discussion

From the results shown in Sections II and III we can observe that both algorithms are very stable. The Brushfire algorithm can be calculated for all points in the map, while the Wavefront algorithm can be calculated for all points that can be connected to the goal, and, if possible, a trajectory between the starting point and the goal will be found. Together, these algorithms are very powerful for path planning of mobile robots.

One disadvantage of these algorithms is that they assume that the map is completely known, which in only true for a limited range of real world applications. The usage of these algorithms with maps that are updated in real time (using SLAM for example) would require frequent recalculation of the potential maps and trajectories, and it could be very demanding or impossible for a simple microprocessor to do it in real time for medium and large maps.

The table in Fig. IV shows the calculation times for the application of the Brushfire and the Wavefront algorithms to the four provided maps. These times were obtained using MATLAB running on one core of an Intel^{R} Core^{TM} i7-5500U CPU @ 2.40GHz. It is important to point out that even when running on a strong modern processor the computing times for large maps are not fit for real time applications if it is recomputed at every cycle.

Map	Num. of pixels	Brushfire	Wavefront
basic.m	280	0.013045	0.007583
maze.m	58564	0.786043	0.169837
mazeBig.m	550974	4.674438	1.861244
obstaclesBig.m	548449	4.005892	1.345025

Fig. 9. Execution times, in seconds, of the Brushfire and Wavefront algorithms over the four provided maps

V. Conclusion

We described and analysed implementations of the Brushfire and the Wavefront algorithms. Both algorithms proved to be very robust, but may not be fit for real time recalculations.

We also observed that the Wavefront algorithm prioritizes a minimum distance to the goal in the computed trajectory, while the Brushfire algorithm is not affected by starting points or goals, since it only guides a robot to stay away from obstacles. This indicates that these two algorithms could be used in conjunction to calculate a trajectory which may not me the shortest possible, but balances between distance and safety by staying far from the obstacles throughout the trajectory.

APPENDIX A
BRUSHFIRE.M

```
function [value_map] = brushfire(map)
% Brushfire algorithm for repulsive potential
% Rodrigo Daudt

    % Initialise value_map
    value_map = map;

    % Find initial values
    s = size(map);
    initials = find(map==1);
    [queue(:,1) queue(:,2)] = ind2sub(s,initials);

    % Offsets for neighbours
    N = [1 0;-1 0;0 1;0 -1;1 1;1 -1;-1 1;-1 -1];

    cl = 2; % Current label
    next_queue = [];

    while size(queue,1) > 0
        for i = 1:size(queue,1)
            for neigh = 1:size(N,1)
                v = queue(i,1) + N(neigh,1);
                h = queue(i,2) + N(neigh,2);
                if v>=1 && v<= s(1) && h>=1 && h<=s(2)
                    if value_map(v,h)==0
                        value_map(v,h) = cl;
                        next_queue = [next_queue;v h];
                    end
                end
            end
        end
        cl = cl + 1;
        queue = next_queue;
        next_queue = [];
    end
end
```

Appendix B

WAVEFRONT.M

```
function [value_map, trajectory] = wavefront(map, start, goal)
% Wavefront path planning
% Rodrigo Daudt

    % Initialise value_map
    value_map = map;
    value_map(goal(1),goal(2)) = 2;

    % Find map size
    s = size(map);

    % Offsets for neighbours
    N = [1 0;-1 0;0 1;0 -1;1 1;1 -1;-1 1;-1 -1];

    cl = 3; % Current label
    queue = [goal(1) goal(2)]; % Current level queue
    next_queue = []; % Next level queue

    while size(queue,1) > 0
        for i = 1:size(queue,1)
            for neigh = 1:size(N,1)
                v = queue(i,1) + N(neigh,1);
                h = queue(i,2) + N(neigh,2);
                if v>=1 && v<= s(1) && h>=1 && h<=s(2)
                    if value_map(v,h)==0
                        value_map(v,h) = cl;
                        next_queue = [next_queue;v h];
                    end
                end
            end
        end
        cl = cl + 1;
        queue = next_queue;
        next_queue = [];
    end

    % Find trajectory
    cp = start; % current point
    trajectory = [start(1) start(2)];

    while value_map(cp(1),cp(2)) ~= 2
        for i = 1:size(N,1)
            % Neighbour coordinates
            v = cp(1) + N(i,1);
            h = cp(2) + N(i,2);
            if v>=1 && v<= s(1) && h>=1 && h<=s(2)
                if value_map(v,h) == (value_map(cp(1),cp(2))-1)
                    cp = [v h];
                    trajectory = [trajectory;cp];
                    break;
                end
            end
        end
    end
end
```