86739: Mobile Robots
Motion planning

Date: Mon 0:00, 30.05.16

Prof. G. Garcia, Prof. P. Martinet

Rabbia Asghar, BEng, Ernest Skrzypczyk, BSc

Contents

List of Figures

Chapter 1

Motion planning

Mapping

For mobile robots, especially autonomous mobile robots, the mapping problem is an essential one and a part of the motion planning process. In this aspect mapping is the process of acquiring spatial models from the environment using available sensors[1]. Information acquisition can be bound to the mobile platform or be an external system communicating with the robot, for example a drone equipped with computer vision processing subsystems flying over an autonomous car. Spatial models, ergo topological, geometric or geographic properties of entities[2], are in case of mobile robots basically geographic maps with the distinction of free space and obstacles. Motion planning of mobile robots deals with finding a trajectory in the workspace linking the starting and goal points contained in the free space, which is defined as a set of configurations avoiding collision with obstacles and is complementary to obstacle region.

Predefined map and Bug algorithms

Bug algorithms, as the name suggests inspired by insect behaviour, assume only local knowledge of the environment and a global goal[3]. Local information is usually attained by tactile sensing, which provides current position of the robot and detects obstacles. Bug algorithms mostly consist of two simple switchable behaviours: moving in a straight line toward goal and following an encountered obstacle edge. They can be implemented in a scenarios, where following requirements are met:

- the workspace area is finite,
- there are finite obstacles in it,
- a line will intersect obstacle finite number of times.

There are several versions of bug algorithms available starting from the simple prototype *Bug0* to *Bug1* and *Bug2* that include certain improvements, but also hold different requirements on the hardware. *Bug2* algorithm defines a line, so called *m-line*, going through the initial point S and the goal point G, which the robot follows heading toward the goal. If an obstacle is in the way, the robot follows its edge until it encounters the m-line again. If the current robot position is closer to the goal, it leaves the obstacle and continues towards the goal following the m-line. This behaviour requires memory, so that the comparison between obstacle and m-line contact coordinates can take place.

[1] Sebastian Thrun, Robotic Mapping: A Survey, 2002
[2] Wikipedia – Spatial analysis
[3] Howie Choset, Robotic Motion Planning: Bug Algorithms

Bug2 algorithm is simple to implement, greedy and complete, meaning it finds a solution if there exists one and returns failure if there is no solution. It takes the first route towards the goal that is closer, so it does not necessarily produce an optimal solution. Simplicity of the algorithm results in its limitations. *Bug2* usually outperforms the exhaustive search algorithm *Bug1*, the latter has a more predictable performance[4].

The implemented *Bug2* algorithm in the Matlab robot toolbox provided by Peter Corke is a more restricted version as explained in *"Bug3 algorithm"*. The toolbox also provides a default *map1* to evaluate planning algorithms, which is a binary image defined in a 2D matrix, where 0 represents free space and 1 obstacle occupied space. This implementation of the algorithm assumes that the robot is a point, thus unoccupied space within the map is considered free space. It does not take into account geometrical and mobility constraint of the robot. For a more realistic implementation of path planning, it is assumed that the robot is a disk of radius $3px$ or *cells*. The maps (1.1) are modified into C-obstacle maps (1.2) according to `Cobstaclemaps.m` so that the occupation of space by the robot is taken into account and effectively reducing the free space to accommodate robots dimensions. Mobility constraints are however not implemented.

Script (1.1) shows used procedure for evaluation of the bug algorithms using custom maps. The efficiency indicator is calculated as the relation between path steps taken and loops of the algorithm, essentially indicating processing power efficiency. Since there is no optimal solution reference provided as a reference for comparison, which would be an objective measurement option, it remains an indication of performance.

Script 1.1: Matlab script for evaluation of the bug algorithms.

```
%% L02p1 - Bug maps script
clear all; close all; clc;

% Setting basic script options
screensize = get(0, 'ScreenSize'); %Screen size
set(groot, 'defaultFigurePosition', [screensize(3)/6, screensize(4)/6, screensize(3)/1.5, screensize(4)/1.5]);
set(groot, 'defaultFigurePaperUnits', 'points', 'defaultFigurePaperSize', [1366 1024])
% Set to off for plot generation
set(groot, 'defaultFigureVisible', 'off');

% imageextension = 'png';
imageextension = 'pdf';

bt = 0.99; % Binarization threshold [0, 1]; More results in thicker objects

for extractpath = 0:1
  for i = 1:8
    % filename = ['Maps/', 'map', num2str(i, '%0.2i'), '.bmp'];
    filename = ['Maps/', 'cmap', num2str(i, '%0.2i'), '.bmp'];
    disp(['Processing ', filename]);
    tmpmap = imread(filename, 'bmp');
    % map = 1 - double(im2bw(tmpmap, bt));
    map = double(im2bw(tmpmap, bt));
    mapsize = size(map);

    figure;
    imshow(map);

    % Parameters
```

[4] Howie Choset, Robotic Motion Planning: Bug Algorithms

```
    sp = [10; 10];
    ep = [50; 35];
    % ep = [60; 45];

    % Initialization of the bug algorithm
    % bug = Bug2(map);
    bug = Bug3(map);
    % bug.verbose = 1;
    bug.verbose = 0;
    try
      bug.goal = ep;
    catch
      disp('Skipping map');
      continue
    end

    % Simulated navigation
    grid on;
    grid minor;
    try
      bug.path(sp); % Simulated navigation
    catch
      disp([filename, ' not solved! Skipping...'])
      continue
    end

    grid on; grid minor;
    title(['Map ', num2str(i)]);
    axis([0, mapsize(1), 0, mapsize(2)]);
    saveas(gcf, ['Bug/', 'Bug3_cmap', num2str(i), 'S'], imageextension);
    % saveas(gcf, ['Bug/', 'Bug3_map', num2str(i), 'S'], imageextension);
    % saveas(gcf, ['Bug/', 'Bug2_map', num2str(i), 'S'], imageextension);

    if (extractpath)
      % Path
      disp('Extracting path');
      figure;
      try
        bp = bug.path(sp); % Extracted path
        plot(bp(:, 1), bp(:, 2), 'g-'); % Plot path
      catch
        disp(['Extracting failed! Skipping...'])
        continue
      end

      grid on; grid minor;
      xlabel('x'); ylabel('y');
      axis([0, mapsize(1), 0, mapsize(2)]);
      saveas(gcf, ['Bug/', 'Bug3_cmap', num2str(i), 'P'], imageextension);
      % saveas(gcf, ['Bug/', 'Bug3_map', num2str(i), 'P'], imageextension);
      % saveas(gcf, ['Bug/', 'Bug2_map', num2str(i), 'P'], imageextension);

    % Print results
```

```
    disp(['Loops: ', num2str(bug.lc)]);
    disp(['Path steps: ', num2str(size(bp, 1))]);
    disp(['Efficiency: ', num2str(size(bp, 1) / bug.lc * 100), '%']);
    end

    % pause
    clear bug tmpmap map; % Clear temporary variables

  end
end
```

Custom maps

Custom maps have been developed using a combination of freely available applications Inkscape and ImageMagick. Maps shown in the figure (1.1) have been designed to provide a variety of possible scenarios that test used algorithms with focus on local minima ((1.1a), (1.1b)), encountering obstacles at different angles ((1.1e), (1.1f)), labyrinths ((1.1e), (1.1h)) and spirals ((1.1f), (1.1g)). The generated C-obstacles maps illustrated in figure (1.2) using the script `Cobstaclemaps.m` resulted in maps (1.2d), (1.2e), (1.2f) and (1.2h) being unusable.

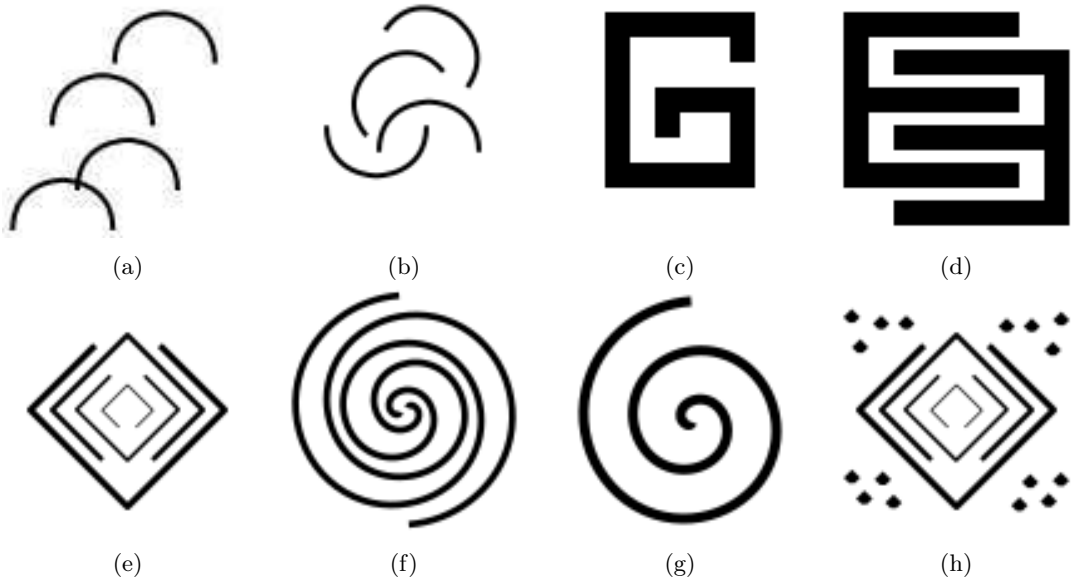
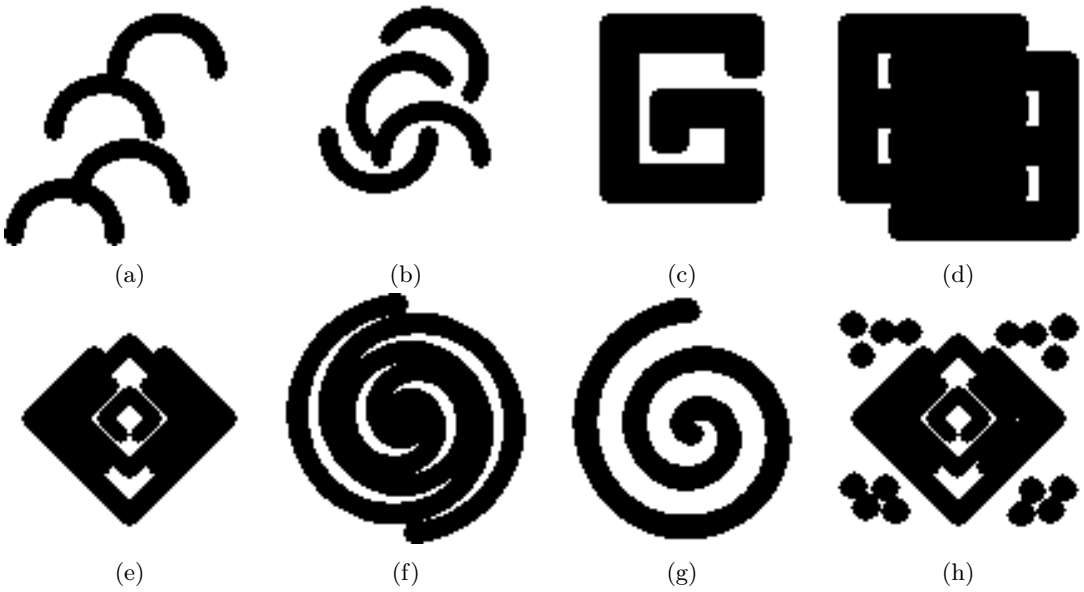


Figure 1.1: Bitmaps serving as custom maps developed to test planning algorithms.

For the purpose of using the maps with the provided toolbox, the vector graphics containing original maps needed converting and conditioning, which included flipping the maps upside down as seen in the used Bash script (1.2). ImageMagick's executable `convert` is required for this script.

Script 1.2: Bash script for converting all svg maps into the bmp format compatible with the toolbox.

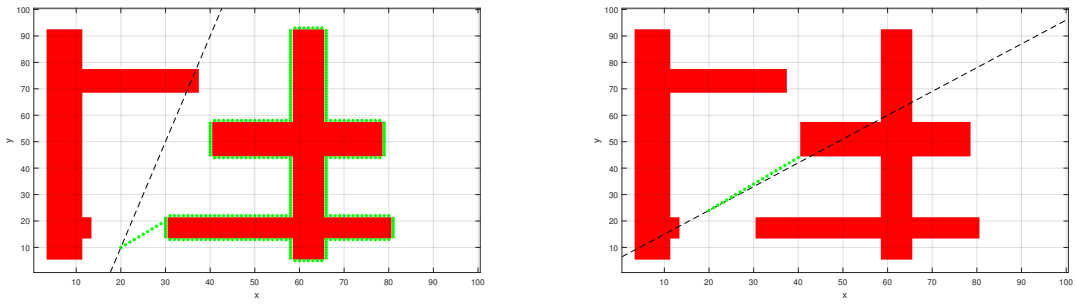
```
#!/bin/bash
for F in Maps/*.svg; do
  echo "Converting $F..."
# convert -colors 2 "$F" -flip -resize 100x100 "$(echo "$F"|sed 's/\.svg/\.bmp/')"
  convert "$F" -flip -resize 100x100 "$(echo "$F"|sed 's/\.svg/\.bmp/')"
done
```



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 1.2: Bitmaps serving as custom C-obstacle maps developed to test planning algorithms.



(a) Robot trapped in local minima (b) Robot corner contact

Figure 1.3: Limitations of *Bug2* algorithm.

Bug3 algorithm

Bug2 algorithm in the robot toolbox implements the same principle as the conventional version. However it only allows the robot to move in horizontal and vertical direction and at other $k \cdot 45°$ angles. Since it does not follow the m-line accurately, it can get trapped in a local minima as shown in figure (1.3a). This behaviour occurs, because the robot follows an obstacle that does not meet the m-line. The *Bug2* algorithm provided in the toolbox detects it arrived at the same position on the edge of the obstacle and returns the error *"robot is trapped"*.

Using the *Bug2* algorithm provided in the toolbox, if the robot approaches an obstacle at a pixel corner as illustrated in figure (1.3b), the vision toolbox package script `edgelist.m` fails with returned error: *"no neighbour outside the blob"*.

Bug3 algorithm was the written to overcome these shortcomings. For the condition when the robot follows m-line, robot does not move at $k \cdot 45°$ angles for any slope of the line other than 0 or infinity. The *Bug2* algorithm defines the m-line using homogeneous representation (1.1) (lines $106 \div 110$ of the script `Bug3.m`). *Bug3* algorithm uses this m-line representation and equation (1.2) to compute change in x and

y coordinates for an increment in the opposite coordinate. The results are then rounded, in the current implementation using the `floor` function, so that they can be converted into coordinates as seen in script excerpt (1.4).

$$mline = \begin{bmatrix} a & b & c \end{bmatrix} \tag{1.1}$$

where

$$ax + by + c = 0 \tag{1.2}$$

Script excerpt 1.3: *Bug3* algorithm – Next step computation from the neighbouring 8 cells.

```
140         d = bug.goal-robot;
            dx = sign(d(1));
            dy = sign(d(2));
```

Script excerpt 1.4: *Bug3* algorithm – Next step computation.

```
151         dxm = floor((bug.mline(3) + (robot(2) + dy) * bug.mline(2)) / (-bug.mline ▼151
    151▲ (1)));
            dym = floor((-bug.mline(3) - (bug.mline(1) * (robot(1) + dx) )) / bug. ▼152
    152▲ mline(2));
```

Excerpt (1.5) checks if all combinations of a possible next step with accordance to (1.4) result in obstacle contact. In case of contact direction of robot is determined. Random and fixed direction have been tested, however from observations made during the evaluation it has been determined that a calculated direction provides optimal solutions to that particular aspect. The trivial computations are based on excerpt (1.3), which results in table 1.1 used then in algorithm excerpt (1.6). Vision toolbox function `edgelist()` that returns a list of edge pixel for the region, is then called with the appropriate direction paremeter (line 190 in script excerpt (1.6)).

Table 1.1: Direction of movement on obstacle contact

Δy \ Δx	-	0	+
+	L	*	R
0	*	/	*
-	R	*	L

Results of equation $sign(sign(\Delta x) \cdot sign(\Delta y))$
$*$ – random direction, / – not available/no movement

Script excerpt 1.5: *Bug3* algorithm – Obstacle detection.

```
163         if bug.occgrid(dym, dxm) && bug.occgrid(robot(2) + dy, robot(1) + dx) &&  ▼163
    163▲ bug.occgrid(dym, robot(1) + dx) && bug.occgrid(robot(2) + dy, dxm)
```

Script excerpt 1.6: *Bug3* algorithm – Movement direction on obstacle contact.

```
181           switch sign(dx * dy)
                case -1 %CCW
                  rdir = 1;
                case 1 % CW
```

```
185             rdir = 0;
              case 0 % Random assignment
                rdir = sign(randn(1, 2) * randn(2, 1));
            end

190         bug.edge = edgelist(bug.occgrid==0, robot, rdir);

            if (rdir == 1)
              disp(['edge ', 'CCW']);
            elseif (rdir == 0)
195           disp(['edge ', 'CW']);
            else
              error('no direction')
            end
```

Script excerpt 1.7: *Bug3* algorithm – Computation algorithm selection and preference list.

```
204         % if and(not(bug.occgrid(dym, dxm)), max(robot(1) - dx, robot(2) - dy)  ▼204
    204▲ <= 1)
205         if not(bug.occgrid(dym, dxm))
              n = [dxm; dym];
              % disp('dxm, dym');
              mf = min(mf + 0.1, 1);
            % elseif and(not(bug.occgrid(dym, robot(1) + dx)), max(robot(1) - dx,  ▼209
    209▲ robot(2) - dy) <= 1)
210         elseif not(bug.occgrid(dym, robot(1) + dx))
              n = [robot(1) + dx; dym];
              disp('dx, dym');
              mf = min(mf + 0.1, 1);
            % elseif and(not(bug.occgrid(robot(2) + dy, dxm)), max(robot(1) - dx,  ▼214
    214▲ robot(2) - dy) <= 1)
215         elseif not(bug.occgrid(robot(2) + dy, dxm))
              n = [dxm, robot(2) + dy];
              disp('dxm, dy');
              mf = min(mf + 0.1, 1);
            % elseif and(not(bug.occgrid(robot(2) + dy, robot(1) + dx)), max(robot ▼219
    219▲ (1) - dx, robot(2) - dy) <= 1)
220         elseif not(bug.occgrid(robot(2) + dy, robot(1) + dx))
              n = robot + [dx; dy];
              disp('dx, dy');
              mf = min(mf + 0.1, 1);
            else
225           warning('Backtracking');
              n = robot - [dx, dy];
              mf = max(mf - 0.1, 0.4);
            end
```

Bug3 algorithm excerpt (1.7) illustrates obstacle tests for new coordinates with respect to the preference list, where the innovative coordinates approach *dxm* and *dym* are preferred. When none of the proposed solutions offers obstacle-free new coordinate pair for the generated path, then backtracking is initialized, however this scenario should never occur and has not been observed during the experimental phase of the evaluation.

Script excerpt 1.8: *Bug3* algorithm – Disable error when robot trapped.

```
239         if bug.k <= numrows(bug.edge)
240           n = bug.edge(bug.k,:)';  % next edge point
            else
              % we are at the end of the list of edge points, we
              % are back where we started.  Step 2.c test.
              % error('robot is trapped')
245           % return;
              warning('robot is trapped')
              mf = max(mf - 0.1, 0.5);
            end
```

Partial script (1.8) shows the changes to disable the error *"robot is trapped"* and introduces the variable mf. Even though it is not used in the current implementation, it is a possible improvement of a potentially failing scenario, where the robot seems trapped. The idea behind the mf parameter is to use it as a scaling factor determining how much the robot gets affected by the m-line at a given distance. This could be further developed by additionally scaling mf parameter proportionally to the distance from the goal pulling the robot towards the m-line as shown in script excerpt (1.9). A precise approach would be guaranteed near the goal.

Script excerpt 1.9: *Bug3* algorithm – M-line detection and follow condition.

```
252         if abs([robot' 1] * bug.mline') <= 0.5
            % if abs([robot' 1] * bug.mline') <= 0.75
            % if abs([robot' 1] * bug.mline') <= 1
255         % if abs([robot' 1] * bug.mline') <= mf
              bug.message('(%d,%d) moving along the M-line', n);
              % are closer than when we encountered the obstacle?
              %TODO refine condition
              % if colnorm(robot-bug.goal) < colnorm(bug.H(bug.j,:)'-bug.goal) || (( ▼259
    259▲ bug.elc > 100) && ((bug.goal - robot) > 10))
260           if colnorm(robot - bug.goal) < colnorm(bug.H(bug.j, :)' - bug.goal)
                % back to moving along the M-line
                bug.j = bug.j + 1;
                bug.step = 1;
                return;
265           end
```

Script excerpt 1.10: *Bug3* algorithm – Loop counters update.

```
269         bug.k = bug.k + 1;
270         bug.elc = bug.elc + 1;
          end % step 2

          % Calculate loops
          bug.lc = bug.lc + 1;
275       if (mfl ~= mf) disp(mf); end
          % disp(bug.lc);
          % disp(bug.elc);
```

Additional performance measuring variables have been added like loopcounters `elc` and `lc`, which get updated in (1.10). Performance of *Bug2* and *Bug3* algorithms has been summarized in the table 1.2. *Bug2* could solve only two of the conventional custom maps `map01` and `map04` and with slightly less steps than the *Bug3* algorithm, despite it's limitations. The computational efficiency of *Bug3* is constant at 50%,

meaning two computation loops are needed for one path step on average. Therefore current implementation of the *Bug3* algorithm does not seem to provide an efficient solution. Number of computation loops is proportional to the time needed to generate a motion plan. Because of limited measurement options this data is inconclusive.

Table 1.2: Comparison between *Bug2* and *Bug3* algorithms solving custom maps

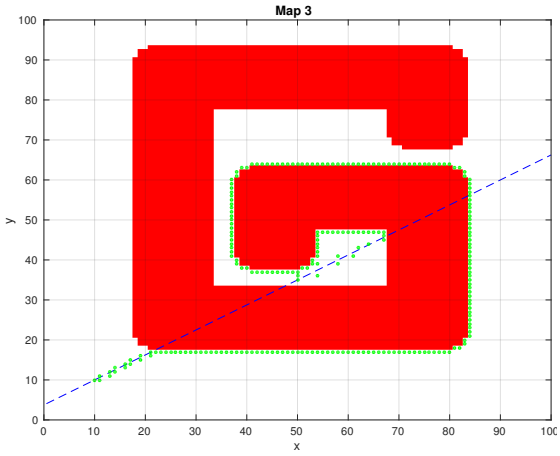
Map	1	2	3	4	5	6	7	8
Bug3								
Loops	184	166	450	238	566	632	/	560
Steps	93	85	226	120	285	318	/	283
Efficiency	50.5	51.2	50.2	50.4	50.4	50.3	/	50.5
Bug2								
Steps	79	/	/	115	/	/	/	/

From a general motion planning perspective *Bug3* is an improvement over the default *Bug2* algorithm from the robot toolbox, however further modifications can be implemented. It has been experimentally determined that tests on the distance between steps are not always satisfied. Depending on the application, precision requirements and control scheme this aspect can be irrelevant, however for completeness sake it should be solved. Another possible improvement would include regenerating the m-line or computing additional ones. One condition for this event could be regenerating the m-line when an angle threshold has been passed following a predefined angle pattern, such that the robot is located at the edge of the back wall of the obstacle with reference to the contact point. Patterns could differ, also the original m-line could still have more influence on the mobile platform than the newly computed one or vice versa. Under specific circumstances this could simulate the *Bug0* algorithm. Random direction of movement selection has been considered a possible improvement, however it does not provide consistency between runs on a map with same parameters.

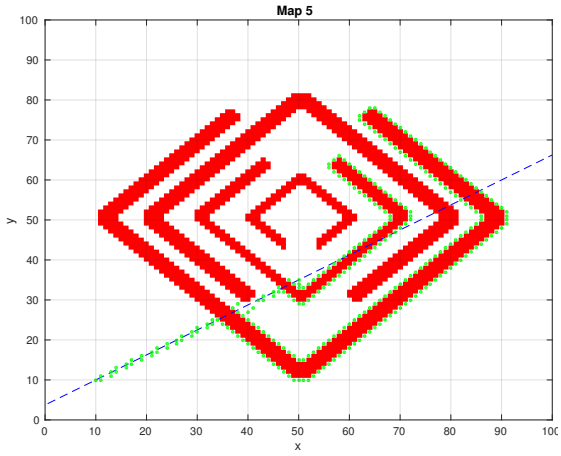
Figure (1.4) portrays *Bug3* solutions on different custom maps, including a C-obstacle map (1.4a). Plot (1.4b) portrays one of the current implementation *Bug3* algorithm limitations, where the solution path does not follow the m-line from the last obstacle on the way towards the goal point at $(50, 35)$. Also depending on the slope of the m-line there is a certain spread of the generated path, usually as a result of computed dxm and dym values. Both aspects should be rectified. The latter problem, should be solvable with distance tests that (1.7) already incorporates in the commented out parts of the code. However (1.4c) shows that the algorithm is capable of detaching from the obstacle, apparently computational aspects play a role in those scenarios, since the theoretical algorithm in (1.9) is sound. The file `Bug3.log` also shows that at each crossing of the m-line the correct decision for the direction of following the obstacle has been made. Figure (1.4d) shows the opposite behaviour, where the robot reverses after following the spirals edge and encountering the m-line, resulting in robot being trapped. This is the only custom map that has a possible solution for the defined start and goal point pair that fails with *Bug3* algorithm. Those inconsistencies are most probably result of computation restrictions on discrete systems and possibly random aspects specific to the used maps.

An interesting application for bug algorithms is maze solving. The current *Bug3* implementation should be able to solve the majority of maze problems, however further modifications to backtracking and the robot being trapped problem would be necessary to provide a practical algorithm. One possible theoretical model worth studying would be the Pledge algorithm[5]. Similarly to the angle pattern approach proposed, angle turns around an encountered obstacle are being counted. In case of facing the previous direction before obstacle contact and the angle sum equal to zero, the obstacle is left in the original direction. That way local minima as portrayed by figure (1.3a) for the *Bug2* algorithm can be overcome.

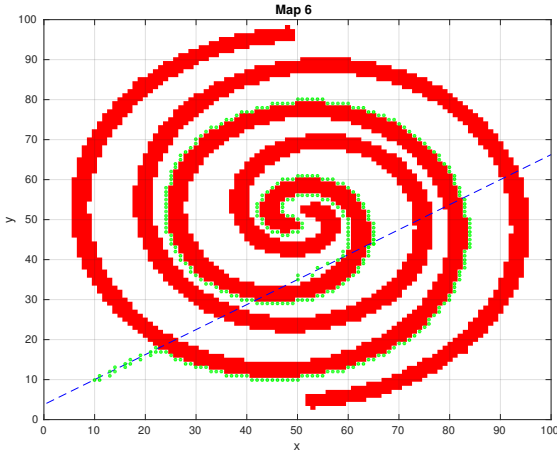
[5] Wikipedia – Pledge algorithm



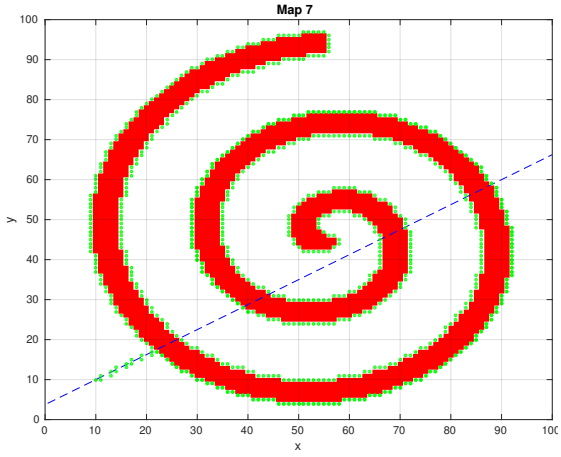
(a) *Bug3* solution of custom `cmap03`



(b) *Bug3* solution of custom `map05`



(c) *Bug3* solution of custom `map06`



(d) *Bug3* solution of custom `map07`

Figure 1.4: Limitations of *Bug3* algorithm.

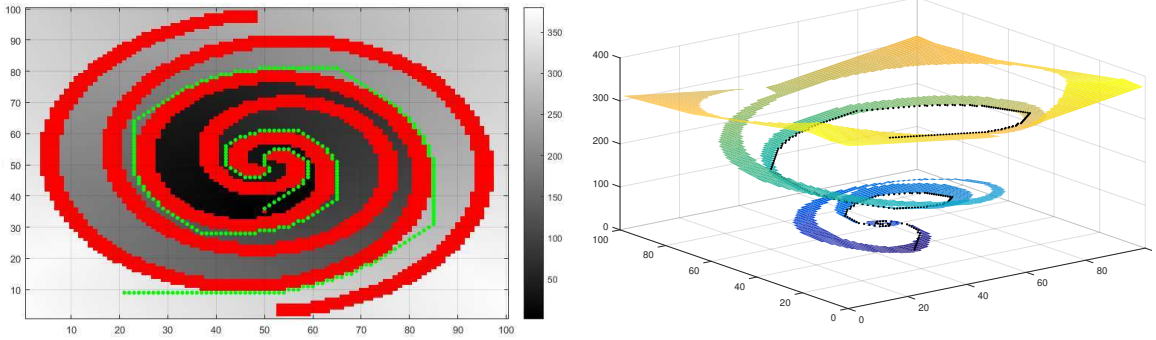
Distance based methods

Distance based methods solve the motion planning problem using a distance matrix computed with respect to the goal point, where the zero value is assigned to the goal and minimum distance from it to other points on the map. These methods use greedy search algorithms, such that the start point follows minimum distance path to reach the goal. They are applicable in environments where global knowledge is provided.

Distance transform method

Distance transform method (DTM) is used to compute minimum distance paths. DTM generates a matrix the same size of map that hold the value of the minimum distance it will take to reach the goal. `DXform.m` class from the robot toolbox is used to perform this technique. Euclidean distance is used for computations by default. Once the minimum distances from the goal towards each cell of the map are computed, trajectory path is generated by moving from the starting point to the neighbouring pixel with the smallest distance towards the goal.

In the figure (1.5a) it can be seen that region around the goal is dark representing small distance towards goal. Further a point is from the goal, brighter it gets, signifying higher distance from the goal. Figure (1.5b) shows a snapshot of 3D plot with the same path. The curve illustrates the minimum distance plane from the goal, which is situated at the base of the plot. The three dimensional curved plane provides analogy to a ball placed rolling down a spiral around obstacles towards the goal. Similarly, the algorithm follows the minimum distance path from start towards the goal.



(a) 2D path planning solution using DTM

(b) 3D plot of path planned using DTM

Figure 1.5: Path planning using distance transform method for custom `map06`.

Script 1.11: Matlab script for DTM and evaluation on toolbox and custom maps.

```
%% L02p2v1 - Distance based methods - Distance transform method (DTM)
close all; clear all; clc;

% Load toolbox map
load map1;
DXform_map(map);

% Load custom maps
for i = 1:8;
  load(['Maps/map0', num2str(i)]);
  DXform_map(map);
end
```

*D** method

*D** method, whose names derives from *Dynamic A** search algorithm, is an improvement of the default distance transform method and allows incremental planning, ergo the cost map can be modified online, which is the difference between *A** and *D**.

The *D** class `Dstar.m` available in the robot toolbox, begins the solution algorithm by generating the cost map. Free space cells are assigned the value 1 and obstacles *NaN* (not a number) as a reference to infinity. The algorithm then computes distances towards the goal based on cost. *D** class offers flexibility to change the cost for the same map.

Script excerpt 1.12: Script for modifying cost map in *D** algorithm.

```
% Change the cost map
for y = 78:85
    for x = 12:45
    ds.costmap_modify([x, y], 2);
    end
end
```

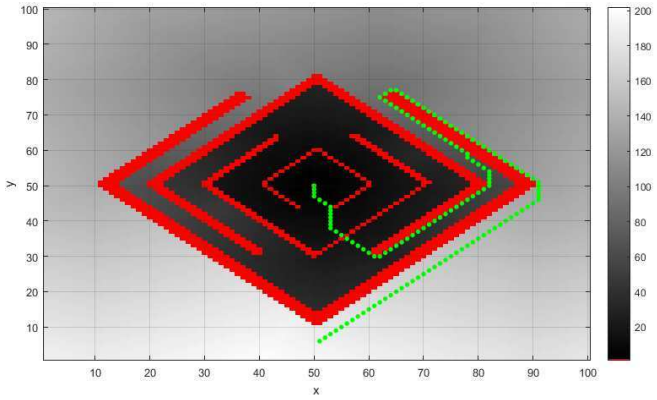
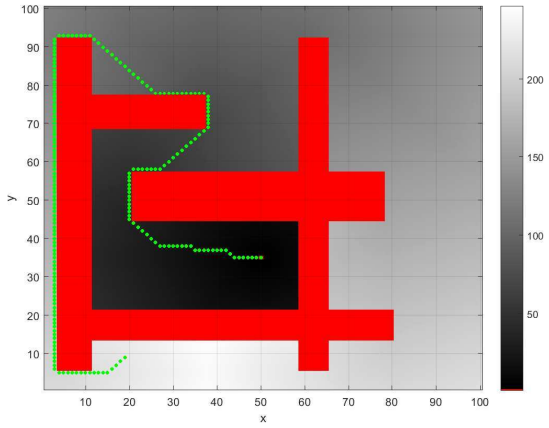


Figure 1.6: Bug in `DXform` and `Dstar` classes – Solution path passing through an obstacle.

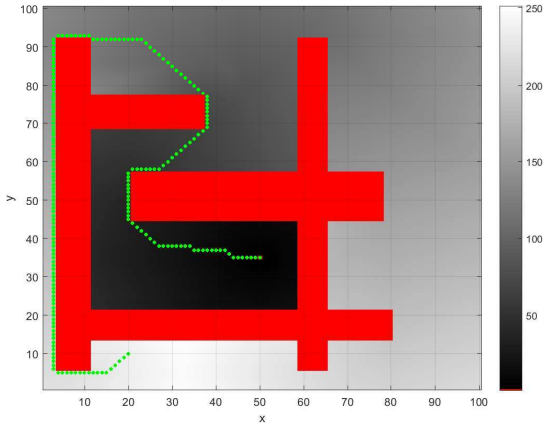
Figure (1.7a) shows the path planned using *D** algorithm with the original cost map. In the script excerpt (1.12) cost map is modified from value 1 to 2 for a small region the path crosses at $x = 12 \div 45$ and $y = 78 \div 85$. Figure (1.7b) illustrates that the path does not pass straight through that region like before. The modified region is avoided to the right and then the path runs diagonally through the range where it finds minimum distance to the goal. If a much larger value of cost, for example 5, is defined to that region, the path will altogether evade that region.

Both classes `DXform` and `Dstar` from the robot toolbox fail however, when the map holds an obstacle on a possible path defined as a diagonal $1px$ line, as shown in figure (1.6). A possible solution would include testing if the neighbouring cell adjacent to the obstacle pixel and the path pixel also holds an obstacle. The `Dstar` class holds another bug, where it is not verified if the start point is placed in an obstacle.

Figures (1.8) illustrate application of *D** algorithm in a dynamic environment. The obstacle located close to the center of figure (1.8a) simulates a dynamic obstacle. The cost map is modified for different positions of this obstacle for the very same map. Figures (1.8b), (1.8c) and (1.8d) show different positions of the same obstacle. The new obstacle position is superimposed on the original map. The new path generated avoids the new and is able to pass through old positions, which are now free space. Cost modification of a map provides a useful tool in dynamic environments or where local knowledge only is provided. Another possible application of this technique is discussed in the section *"Voronoi graph"*.

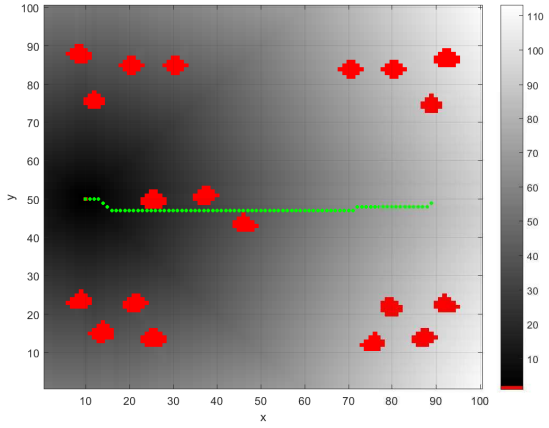


(a) Path planned using D^* algorithm

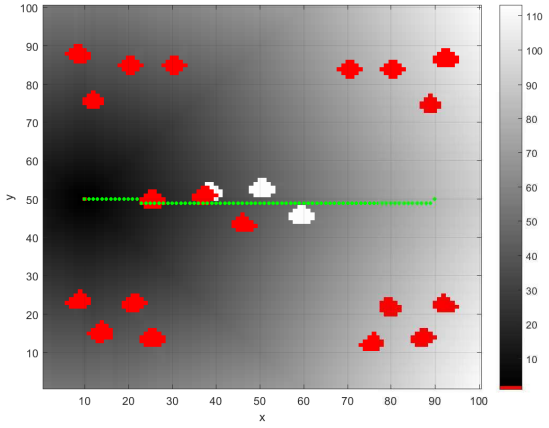


(b) Path planned using D^* algorithm with cost map modified in the region $x = 12 \div 45$ and $y = 78 \div 85$

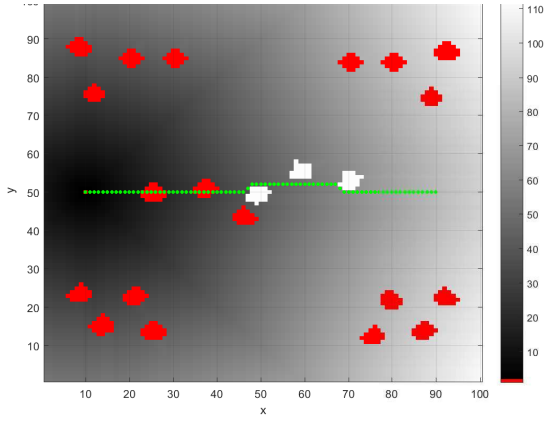
Figure 1.7: Results of modifying cost map in a region using D^* algorithm.



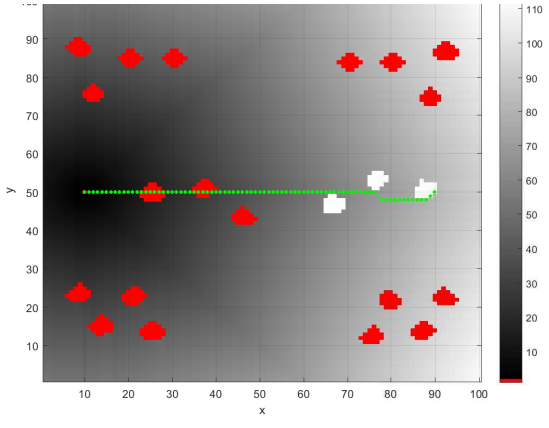
(a) D^* solution with the original cost map



(b) D^* solution with modified cost map simulating a dynamic obstacle



(c) D^* solution with modified cost map simulating a dynamic obstacle



(d) D^* solution with modified cost map simulating a dynamic obstacle

Figure 1.8: D^* solution for a dynamic obstacle.

Voronoi graph

Voronoi diagrams are another technique for path planning. The widely applied in online coverage path planning generalized Voronoi diagram (GVD) algorithm determines paths running at equal distance between obstacles[6]. The workspace area itself must be limited by a boundary. With the assumption that the robot is a point, ergo with no correlation to the size and dimensions of the mobile platform, the generated path between obstacles tends to be the safest path in a very simple safety approach.

Voronoi diagrams are generated by morphological skeletonization of the map using `ithin` function provided in the vision toolbox, which returns a binary skeleton for the provided binary map. In order to implement Voronoi diagrams, a boundary of the map must be defined, usually by either expanding the map by $1px$ with the value set to 0 or filling the outer pixels of the map with that value as in the case of script (1.14).

The script `Lab2p3v1algorithm.m` incorporates random generation of start and goal points. Different techniques can be used to plan a path once the skeleton has been generated. The skeleton is morphologically closed[7], ergo it is first dilated and then eroded using a disk shaped structuring element. This operation smooths out turns and junctions of the Voronoi graph, making them more feasible to perform for most non-holonomic mobile robots. D^* algorithm is then used to modify the map cost according to the skeleton graph as follows.

1. Cost map $c1$ and minimum distance path is generated for the original map using `Dstar.m` class provided in the toolbox. Free space is assigned cost value 1, while an obstacle is assigned infinity by default.
2. Voronoi diagram generated from the map is processed and smoothed out using morphological closing, ergo erosion of dilation of the skeleton.
3. Skeleton is then inverted to match the map format and used to create a new cost map $c2$ using `Dstar.m` class functions `costmap_get()` and `costmap_modify()`. The skeleton is assigned the cost value 1, while the rest of the map is assigned infinity.
4. Cost map $c1$ is modified so that
   - the obstacles cost remains unchanged at infinity,
   - the skeleton is assigned cost 1,
   - and the rest of the free space is assigned cost larger than 1.

The optimal candidate for cost value for remaining free space was experimentally determined to be 5. This ensures that the optimal route for the robot is via the Voronoi graph. If the start and the goal points are outside the skeleton, the computed path will progress towards goal while also inclined towards the skeleton. The map is modified so that the skeleton is superimposed on the original map.

Script 1.13: Matlab script to conduct path planning using Voronoi graphs.

```
%% L02p3v1algorithm - Voronoi graphs - Processing algorithm
close all; clear all; clc;

% Robot is assumed to be a point

% Load toolbox map
load map1;
```

[6] Ming-Lei Shao, Ji-Yeong Lee, Chang-Soo Han, Kyoo-Sik Shin, Sensor-based Path Planning: The Two-Identical-Link Hierarchical Generalized Voronoi Graph, 2015

[7] Mathematical morphology – Closing

```
% Extract skeleton of the map
skeleton = Voronoigraph(map);

% Set fixed start and goal points
start = [56 24];
goal = [79, 33];

% Randomly generate start and goal point
while(true)
  start = floor(1 + 99 * rand(1, 2));
  % Verify if start point is free space
  if (map(start(2), start(1)) == 0)
    break;
  end
end

while(true)
  goal = floor(1 + 99 * rand(1, 2));
  % Verify if goal point is free space
  if (map(goal(2), goal(1)) == 0)
    break;
  end
end

% Run Dstar algorithm on the map
figure;
ds = Dstar(map);
c1 = ds.costmap_get();
ds.plan(goal);
p = ds.path(start);
ds.plot(p);
figure;

%Close (dilate and then erode) the skeleton to create more feasable routes for the robot
se = strel('disk', 5); % Structuring element of shape disk and size 5
skeleton = imerode(imdilate(skeleton, se), se);
idisp(skeleton);
title('Skeleton');
skeleton = 1 - skeleton;

% Modify costmap using the generated skeleton in Dstar
figure;
dstemp = Dstar(skeleton);
c2 = dstemp.costmap_get();
for y  = 1:100
  for x = 1:100
    if (c2(y, x) == Inf && c1(y, x) ~= Inf)
      ds.costmap_modify([x, y], 5);
    elseif (c2(y, x) == 1 && c1(y, x) ~= Inf)
      ds.costmap_modify([x, y], c2(y, x));
    end
```

```
  end
end

% Execute the planner
ds.plan();
ds.path(start);
```

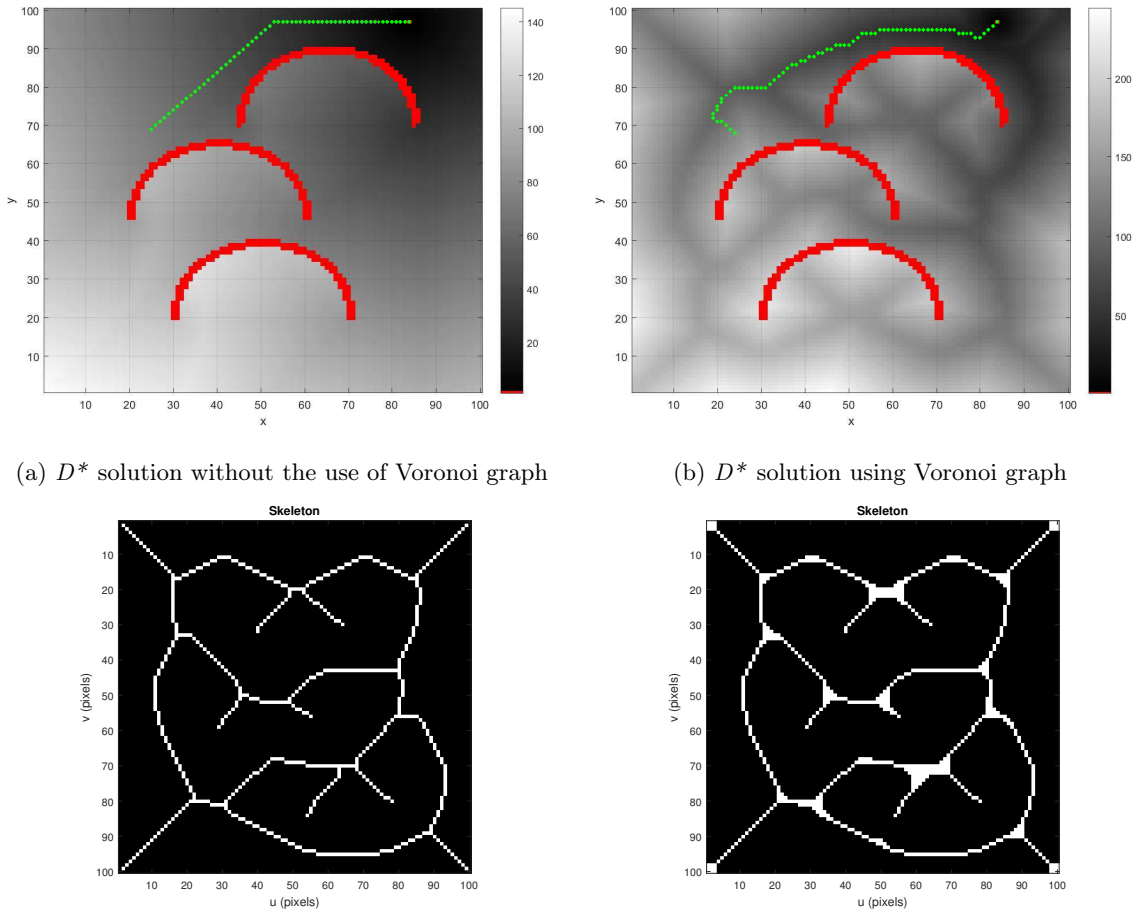
Script 1.14: Matlab function for generation of Voronoi graphs.

```
function [ skeleton ] = Voronoigraph( map )
%% Voronoi Graph
% Returns a skeleton holding the Voronoi graph for the map.

% Inverse and condition map
free = 1 - map;
lx = size(free, 1);
ly = size(free, 2);
% Set boundary on the limits of map
free(1, :) = 0;
free(lx, :) = 0;
free(:, 1) = 0;
free(:, ly) = 0;

% Generate skeleton
skeleton = ithin(free);

% Display free space
figure;
idisp(free);
title('Free space');

% Display skeleton
figure;
idisp(skeleton);
title('Skeleton');

end
```

Figures (1.9c) and (1.9d) show the extracted and the processed skeleton respectively. It can be seen that the processed skeleton fills up sharper turns and provides alternate routes to the robot. The skeletons displayed in the figure are flipped vertically for the corresponding map (1.10a), (1.10b). This is because the vision toolbox uses different y axis order for the skeleton representation only.

Figure (1.9a) shows the path from start to goal point using D^* algorithm alone, while figure (1.9b) shows the path solution using D^* algorithm and the Voronoi graph. The difference between the two planned paths is significant. Plotted path in figure (1.9a) reaches toward the goal following the minimum distance, where entire free space is assigned cost 1. This results in the planned route being very close to the obstacles. Path in figure (1.9b) on the other hand, follows, what can be considered a safer route, that is distant from both, the obstacles and map boundary. The cost of free space marked by the skeleton is lower than the cost of the rest of free space resulting in minimum distance path over the skeleton. This is visible in the graph as the Voronoi graph region is darker than the surrounding free space.

The lack of smoothness in the generated path is the result of computed Voronoi graphs on a low resolution map of size $100px \times 100px$. With a map of higher resolution, the path would seem more smooth. A robot programmed to *strictly* follow this path evolution would need to change its orientation very often. However



(a) D^* solution without the use of Voronoi graph

(b) D^* solution using Voronoi graph

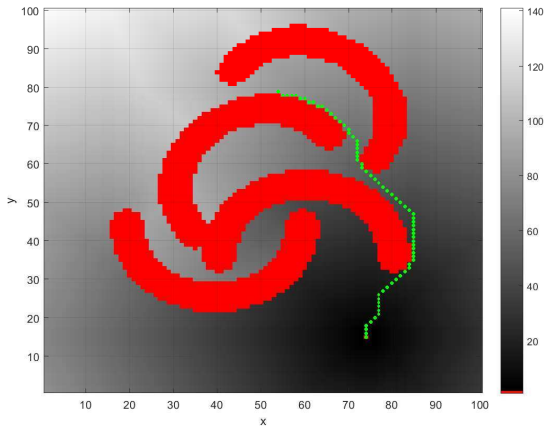
(c) General Voronoi diagram

(d) Voronoi diagram after morphological closing

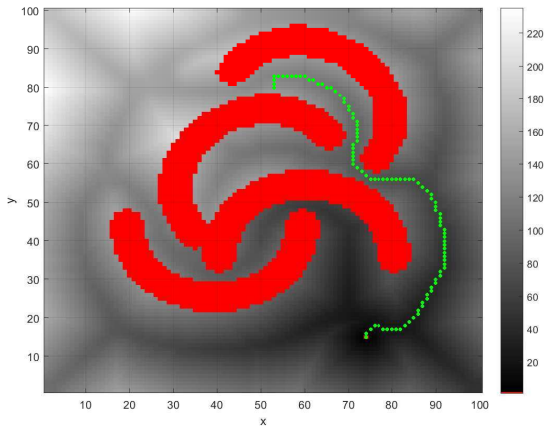
Figure 1.9: D^* solution using Voronoi graphs for custom `map01`.

from the control point of view this is not difficult to overcome. Approaches using specific Lyapunov control functions would allow to smooth out any irregularities in the path trajectory, making the presented solution suitable for majority of wheeled mobile robots.

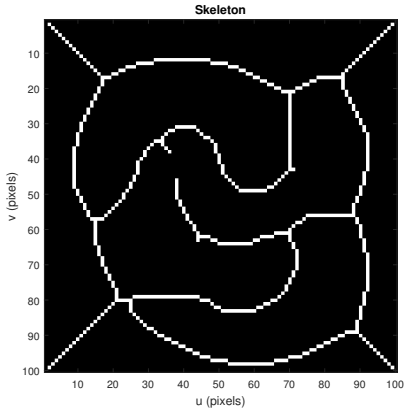
Figure (1.10) shows the result of implementation of Voronoi with the C-obstacle `cmap02`. Similar observations as with `map01` can be made, however in this case the map already accounts for the correlation to the size and dimensions of the mobile platform. Using C-obstacle maps with Voronoi graphs would be the first step towards a safe path planning.



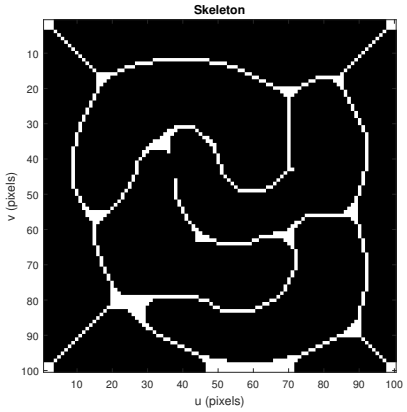
(a) D^* solution without the use of Voronoi maps



(b) D^* solution using Voronoi map



(c) General Voronoi diagram

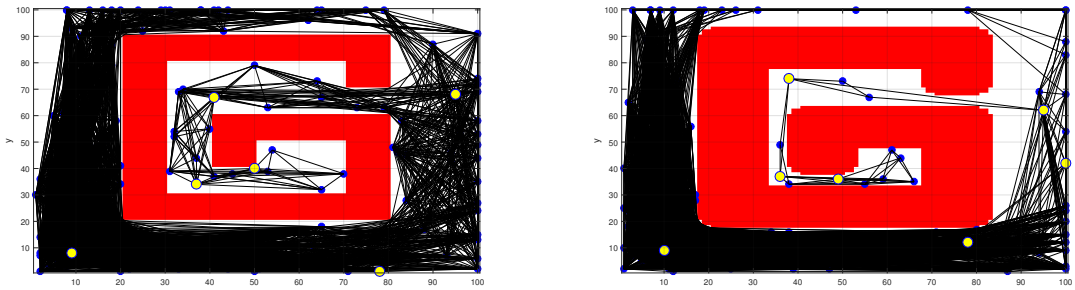


(d) Voronoi diagram after morphological closing

Figure 1.10: D^* solution using Voronoi graphs for custom C-obstacle `cmap02`.

Probabilistic roadmap mapping

Probabilistic roadmap mapping (PRM) method finds application in large environments where the global knowledge of the map is provided, but techniques like distance transform or Voronoi graphs take high computation time or are not applicable. PRM from the robot toolbox is used to simulate path planning in the maps studied in previous section. General PRM algorithm consists of two phases: a construction and a query phase. The construction phase first finds N random points that are located in the free space. By default the implemented algorithm uses 100 points, which are then connected to the nearest neighbours by straight line that does not cross any obstacles. In the query phase the start and goal points are connected via previously generated points. The distance threshold between two connected cells can be varied and is equal to 30 cells by default.



(a) Path planned using PRM method on `map03.bmp`

(b) Path planned using PRM method on C-obstacle `cmap03.bmp`

Figure 1.11: Probabilistic roadmap mapping on a custom map set `map03`, `cmap03` with $n = 150$ and $d = 75$.

Figure (1.11) shows PRM solutions for a custom `map03` and the corresponding custom C-obstacle `cmap03`. Based on experience gathered during evaluation of the PRM method, it can be stated that the C-obstacle maps are more difficult to solve than their conventional counterparts, since the obstacles occupy more space. This leads to a more concentrated distribution of random points, which reduces probability of finding a path between starting and goal points. However the requirement on the distance threshold parameter weakens at the same time. The parameter values had to be increased from the default values to find a solution.

Depending on the map's content and size, number of random points n and the distance threshold d, the computation time for a solution can be significantly high, making PRM unsuited for certain applications. Table 1.3 summarizes the performance of the PRM using different parameters for the `generic map` from the robot toolbox of size $100px \times 100px$. The measurements have been determined using `Lab2p4vtest.m` script , which consist of parametrization of n and d with values $10 \div 100$ and a statistically significant loop number k equal to 100 loops. As it can be seen there are no solutions for the tested map with a distance threshold d set to value 10, however 10 points are sufficient for a success rate around 5% with the distance threshold d above 40. It is assumed that practically usable results are indicated by a higher than 75% success rate, so that the path could be generated online by the mobile platform without the need to stop in order to repeat calculations until a solution has been found. This requirement is fulfilled by the parameter pair $n = 90$, $d = 40$ as well as from the other low end side by the parameter pair $n = 40$, $d = 100$ and by other cells marked *green*, which is less than half of the parametrization table. Interestingly $n = 90$ and $d = 90$ resulted in 100% success rate. Further increase in either of the parameters did not guarantee 100% success rate. It should be noted that both parameters are important for a successful path generation from start to goal point. However from the shape of the cells fulfilling the 75% requirement, it can be concluded that the number of points n is slightly more significant. Both paramters seem to result in a non-linear function for

success rate. The small number of loops $k = 100$ taken from the test script `Lab2p4vtest.m` might not have provided statistically significant measurements, however a second run with $k = 1000$ was conducted providing similar results.

Table 1.3 is missing the computation time for all of the solutions and failed attempts. However it has been observed that the computation time is exponentially proportional to the number of generated points and to the distance threshold, where the former is of more influence than the latter. Based on the conducted experiments, it can be speculated that PRM is probabilistically complete, ergo with the increase of number of sampled points, the probability that the algorithm will find a solution path, if one exists, approaches 1. Further it can be stated, that the convergence time is inversely proportional to the threshold distance and number of points, assuming a solution is possible. The planner finds a solution faster, given the set of points covers a large fraction of space with other parts of space covered in their subsets, ergo when tree of points to a certain level covers the majority of map.

Since the roadmap points generation relies on a randomness, whose implementation on discrete systems is non-trivial, there is no *hard guarantee* on finding a solution, even after many solution attempts or with increase of relevant parameters. The relatively long computation time makes PRM a weak candidate for real-time applications, where aspects like safety, precision and reaction time of the system, especially if the work environment is dynamic, are of high priority. However this reduces the probability of finding a solution, since the ratio between obstacles and free space raises.

Table 1.3: Success rate of PRM with different parameters expressed in percent

n \ d	10	20	30	40	50	60	70	80	90	100
10	0.0e+0	0.0e+0	0.0e+0	2.0e+0	4.0e+0	4.0e+0	5.0e+0	9.0e+0	6.0e+0	5.0e+0
20	0.0e+0	0.0e+0	1.0e+0	1.0e+0	5.0e+0	1.4e+1	2.0e+1	2.5e+1	1.7e+1	2.1e+1
30	0.0e+0	0.0e+0	0.0e+0	5.0e+0	1.3e+1	3.2e+1	3.4e+1	4.7e+1	4.4e+1	4.8e+1
40	0.0e+0	0.0e+0	1.0e+0	6.0e+0	3.5e+1	4.7e+1	5.6e+1	6.2e+1	6.8e+1	7.9e+1
50	0.0e+0	0.0e+0	9.0e+0	2.9e+1	5.3e+1	7.4e+1	7.2e+1	7.7e+1	7.5e+1	7.7e+1
60	0.0e+0	0.0e+0	9.0e+0	3.8e+1	7.2e+1	8.2e+1	8.2e+1	8.3e+1	9.3e+1	9.0e+1
70	0.0e+0	0.0e+0	2.3e+1	5.3e+1	7.6e+1	9.1e+1	9.2e+1	9.4e+1	8.7e+1	9.3e+1
80	0.0e+0	1.0e+0	3.4e+1	6.2e+1	8.1e+1	9.0e+1	9.6e+1	9.7e+1	9.6e+1	9.8e+1
90	0.0e+0	4.0e+0	5.2e+1	8.0e+1	9.0e+1	9.2e+1	9.6e+1	9.7e+1	1.0e+2	9.5e+1
100	0.0e+0	3.0e+0	5.0e+1	8.7e+1	8.9e+1	9.7e+1	9.9e+1	9.8e+1	9.9e+1	9.8e+1

n – number of points, d – distance threshold, ▭ – requirement fulfilled

Conclusions

From the validation and simulation parts in each section for it can be stated that the problem of motion planning is a complex one. Evaluated were the bug algorithms, distance based methods, including D^*, Voronoi graphs and finally the probabilistic roadmap mapping.

Path planning algorithms like the bug algorithms can be implemented when no global knowledge of the environment is given, only the goal. In general bug algorithms are complete, but not optimal. Mobile platforms using bug algorithms for motion planning, depending on the version used, require memory to store the m-line and contact points with obstacles, but a very large memory or high processing power is not required. In real-time applications and where only local knowledge is provided through sensors, *Bug3* algorithm is simple and efficient to implement and use, although the current limitation would have to be overcome. Several improvements have been proposed.

Distance based methods are complete and optimal. They take the advantage of the global knowledge of the environment. Since minimum distance to goal is computed for all points in the environment, hardware memory proportional to the size of used maps is required. For static environments the distance map needs to computed only once. In case of dynamic environments it is possible to modify the cost map only for the region of interest using the D^* method, however the distance map will need to be recomputed from the first encountered dynamic obstacle onward. Online computation conducted on parts of maps, makes D^* a real-time application candidate. The cost maps can include other properties than just the default cost of 1 for horizontal and vertical movement as well as the $\sqrt{1}$ cost for diagonal movement. One possibility are mobility constraints of the robotic platform. It is suggested however to deal with other aspects of mobile robotics in motion planning and use control for mobility constraints, since these can be crucial and mapping might not always be guaranteed or precise. Also such modifications make further processing of maps more complex.

Voronoi graphs can be used for robot motion planning when global knowledge is provided and when they are combined with other algorithms like the D^* search algorithm. Morphological skeletonization of the map resulting in equidistant points from obstacles and the maps boundary can be combined with other techniques for further improvement and capabilities in motion planning. The combination of Voronoi graphs with C-obstacle maps and D^* path search algorithm could be first step towards a safety concerned solution, since the correlation to the size and dimensions of the mobile platform would be provided. The D^* algorithm could take care of unforeseen changes in the environment. However, after evaluation it can stated that Voronoi diagrams do not offer shortest path to the goal and are therefore not optimal. In case of real-time applications with dynamic obstacles, the Voronoi graphs would have to regenerated, possibly periodically. Since the algorithm has a computation time proportional to the size of the map and the computation occurs globally, on the whole map, it is a weak candidate for real-time application scenarios.

Probabilistic roadmap mapping is neither complete nor optimal. This particular technique is applicable in environments, where the ratio between obstacle occupied and free space is low. A ratio too high makes PRM practically unusable. Reliance on randomness and difficulty of its implementation make the algorithm not consistent across runs and maps. PRM is however probabilistically complete, when the number of random points is high enough for a given map. High computation time, dependent on the map size and exponentially dependent on the number of points and distance threshold, as well as lack of predictability make PRM not applicable in real-time scenarios.

Different improvements have been proposed throughout all evaluated algorithms. Another possible optimization technique includes a technique known from optical flow generation from the domain of image processing in computer vision. In order to determine a global as well as local optical flow, the image is resized creating pyramids then used by hierarchical Lucas-Kanade algorithm. Similarly, in order to deal with higher resolution maps, a sub-sampling technique could be used to solve low resolution map first, where the robot would start moving. Then solve the motion planning problem for a medium resolution and finally original, high resolution map. The speed of movement should be proportional to the resolution, ergo low for

low resolution maps and high for the original, respecting the application specification.

Concluding presented results, it can be stated that the motion planning problem can be solved using a multiple of techniques. As it is with most complex problems, the most efficient approaches are often a combination of chosen positive aspects from different techniques in such a manner that the task is completed fulfilling it's requirements.