Implementation and analysis of the main path planning techniques

Alessandro Zini, Filippo Morselli, Carlo Stomeo
Dipartimento di Informatica - Scienza e Ingegneria, University of Bologna, Italy
Emails: alessandro.zini3@studio.unibo.it, filippo.morselli@studio.unibo.it, carlo.stomeo@studio.unibo.it

***Abstract*—This report describes the final project for the Mobile Systems course. The work that has been done represents the first part of a larger project, which aims at implementing different mobility algorithms for a rover equipped with sensors in a real-world environment with obstacles.**

In this first phase a JavaScript application, Path Planner [1], has been developed, which provides a generic 'playground'-platform for positioning and moving objects in a bi-dimensional grid-based environment. Within such environment, the most popular path planning approaches studied in the Mobile Systems course have been implemented, tested and compared.

The final product of this phase consists of an application that implements a user customizable environment in respect to size, obstacle positioning, as well as start and destination point. Using the software it is possible to obtain a path from start towards the destination, with the possibility of varying the path planning algorithm.

The main purpose of this work is to put into practice the path planning algorithms seen during the lessons and study them up to the implementation details. Another goal is to have an easily customizable and extensible software to use as a starting point for the second part of the project, that is the actual mobility of an agent in the real world.

I. Introduction

Path planning is the process of determining the path an agent has to follow in order to reach a given destination. It's a problem with lot of variants, depending on the characteristics of the environment in which the entity has to move and also depending on the capabilities and knowledge of the entity itself.

The context that has been assumed during the making of the project is a bi-dimensional environment partitioned in a grid map. Each cell of the grid can be free or occupied by an obstacle object. Two of the free cells are marked as starting point and destination. The problem consist in finding the best possible path to reach the destination cell from the start position.

It is assumed that the dimension of the moving agent are always smaller that the one of the unit cell, therefore the path is defined as a sequence of free adjacent cells that lead the agent to the goal. Diagonal movements are allowed, provided that the agent does not go through two diagonally adjacent obstacles. Also, the environment is assumed to be static, so there can't be any modification to the displacement of obstacles once the path computation starts.

In such environment, two different versions of the problem have been analyzed: the *global* or *offline path planning* in which there is a full *a priori* knowledge of the map configuration, and the *local* or *online path planning* where the only inputs are the starting position and the destination position and the agent discovers the presence of obstacles only when they are within a certain range, so to simulate proximity sensors.

For both kind of problems, the most popular techniques are implemented in order to study their behavior in different environment configurations and to compare their performances.

The implemented techniques are the following:

- Cellular/Grid Decomposition
- Visibility Graph
- Probabilistic Roadmap
- Potential Field
- Bug (version 1 and version 2)
- Tangent Bug

The paper is organized as follows. In Section 2 there is a brief survey of related works and similar applications. Section 3 describes the architecture of the web application and the module used to implement it. Section 4 describes the implementation details of each path planning method adopted. In Section 5 a performance evaluation for the various algorithms is presented. Last, Section 6 illustrates the conclusions.

II. Related works

The first step we took in our work has been to look for projects similar to our concept. We found two projects ([2] and [3]) from which we borrowed some ideas regarding the layout of the project. Path Planning has been described and studied in depth in the state of the art. Warren et al. [4] proposed to use potential fields for robotic manipulators and mobile robots in the presence of stationary obstacles. In [5] Sariff et al. presented an overview and discusses the strength and weakness of many path planning algorithms including Visibility Graph and Potential Fields. In [6] Yoon et al. discuss in details the Probabilistic Roadmap approach: Geraerts et al. in [7] analyze and compare many variants of this approach. Regarding the implementation of Bug approaches, [8] has been discovered to be a very useful resource; in addition, various different variation of Bug algorithms, including Bug v1, Bug v2 and Tangent Bug, are described, tested and compared on different environments by James et al. in [9].

III. Architecture

While investigating the architecture for the software, the idea of developing a standalone web page has appeared to emerge among other choices. The primary reason for this choice was portability, followed by the advantage of previous experience by the members of the team in term of web development, and the general ease of programming of user interfaces in the web environment.

In order to provide a smooth and, when possible, lightweight experience, the use of external libraries has been reduced to the minimum; also, the 'playground'-grid has been maximized in space, being the real focus of the software. The user interface has been realized in HTML5 using Materialize CSS, a front-end framework based on Material Design. A floating menu has been introduced to let the user have the control over which planning algorithm to use, and to have some other handy functions available, like obstacle clearance, ...

The grid and all its graphic functionalities have been realized in pure HTML5 using the `canvas` element. To ease the development, we wrote a small library which takes responsibility for handling all the functionalities related to drawing both the grid and its elements (squares, circles, lines, ...). Lastly, all the algorithms have been implemented in JavaScript, subdivided by their family of belonging. When required, an external library has been used to compute the Dijkstra shortest path algorithm over a graph.

IV. Implementation

A. *Cellular/Grid Decomposition*

A decomposition algorithm, as the name suggests, works by decomposing a continuous environment into a discrete grid, which is then properly translated into a graph where a shortest path algorithm (*e.g.* Dijkstra) is applied. Such algorithms belong to the family of offline algorithms. In this work, both Cellular and Grid Decomposition approaches have been grouped into a single approach, since the initial assumptions of the environment caused the two to be equivalent. A brief explanation is given below.

A *Cellular Decomposition* algorithm goes through the following steps:

- subdivide the free space F into connected regions (cells);
- determine which cells are adjacent and construct an adjacency graph, where vertices corresponds to cells and edges join cells that have a common boundary;
- search for a path in the adjacency graph between the cells corresponding to start and end points;
- from the path resulted from the previous step, compute a path connecting certain points of the cells such as their midpoints (centroids).

While such approach is very simplistic, it is not usable for 8-directional motion planning, since it requires extra movements other than the canonical N-NE-E-SE-S-SW-W-NW (Fig. 1).

To allow movements in 8 directions, the environment should be decomposed into squared tiles, which would cause the approach to behave identically as if the environment would have been decomposed into a grid (Fig. 2), therefore causing it to degenerate into a Grid Decomposition method.

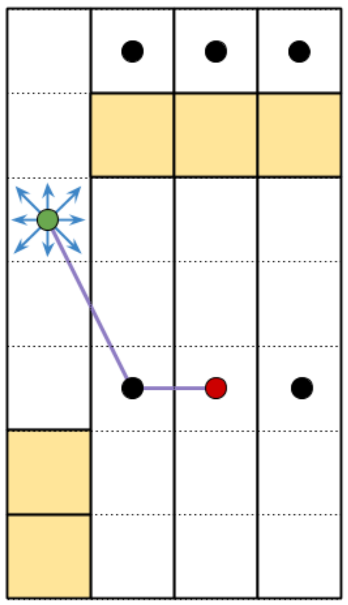
Fig. 1. The blue arrows show the 8 (theoretical) allowed directions, while the purple line shows the path from source to destination obtained with the algorithm. The discrete grid environment assumed in this work is dotted in the background, while the resulting Cellular Decomposition cells are outlined in bold, with their centroid denoted by a black dot.

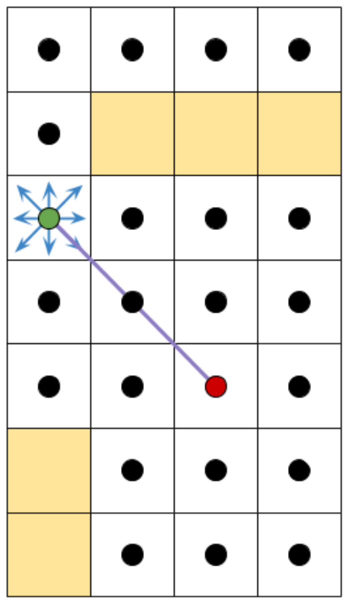
Fig. 2. A Cellular Decomposition approach degenerated into a Grid Decomposition method.

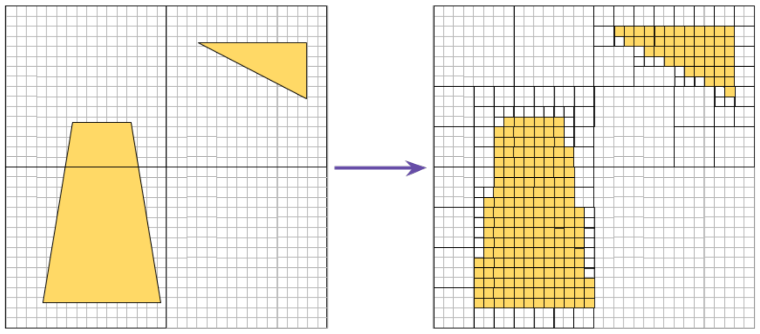
Fig. 3. A Grid Decomposition scenario.

A simple *Grid Decomposition* approach differs from the Cellular counterpart by how it subdivides the free space. It goes through the following steps:

- decompose the space into a grid of cells of equal size;
- mark every cell either as *free* or *busy* depending on the presence of an obstacle inside.
- determine which free cells are adjacent and construct an adjacency graph, where vertices corresponds to free cells and edges join free cells that have a common boundary;
- search for a path in the adjacency graph between the cells corresponding to start and end points.

Since the naive approach suffers from resolution and memory occupation problems, there are many variants, such as the *Recursive Grid Decomposition* method which recursively decomposes cells that are found to be busy; moreover, using a quad-tree structure it is also possible to further distinguish between fully and partially occupied cells. Anyway, given the assumptions stated at the beginning of the report, any variant of the approach would always end up in the same conditions, as shown in Fig. 3: since any obstacle can only be square-shaped, any complex obstacle is already subdivided in elementary squared tiles, causing the decomposition to be pointless.

For these reasons, both approaches have been grouped into a single one, which:

- builds a map of the environment, excluding any obstacle;
- convert such map into a graph, where vertices corresponds to cells and edges join cells that have a common boundary;
- compute Dijkstra's shortest path over the graph obtained in the previous step.

B. Visibility Graph

This is another approach belonging to the offline path planning algorithm family. In particular, this approach aims at discovering the shortest obstacle-free path by applying a shortest path algorithm (*e.g.* Dijkstra) on a graph representing a map of obstacle positions and distances in the environment, called the *Visibility Graph*. Such graph is created as follows:

- **nodes:** vertex points of every obstacle in the environment, in addition to start and destination points;

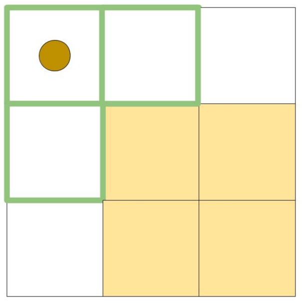
Fig. 4. Example of obstacle vertex detection method. The considered obstacle is the one in the central cell. Adjacent cells in N, W and NW directions are empty. Therefore diagonal cell NW is considered as an obstacle vertex. Of course the algorithms has to check also the other three diagonals.

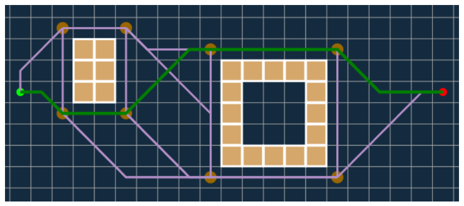
Fig. 5. Example of a visibility graph and the best path.

- **edges:** segments between each pair of nodes that does not intersect an obstacle;
- **weights:** euclidean distances between vertices in the environment;

The implementation of the method for this project is slightly different from its original definition due to the assumption of a grid-partitioned environment. Therefore, the following adjustments have been applied:

- each node of the Visibility Graph is a cell of the grid
- the considered nodes are the the start cell, the destination cell and each obstacle vertex cell
- obstacle vertex cells are detected by checking the surrounding configuration of the environment of each obstacle cell as described in Fig. 4;
- edges are obtained by creating *dummy paths* between each possible pair of node cells; a dummy path is defined as the simplest path between two cells, ignoring obstacles; only dummy paths without obstacles are considered edges for the Visibility Graph;
- the weight for each edge is the number of cell its dummy path is composed of.

The shortest path algorithm is then executed on the graph obtained as described above, in order to find the best available path from the start node to the destination node.

C. Probabilistic Roadmap

This method follows the same general principles of the Visibility Graph. It represents the environment as a graph and then apply a shortest path algorithm. The main difference is in the procedure that builds the graph, which is highly simplified.

The nodes of the graph are a set of randomly selected free cells in the grid, plus the start and the destination cells. The cardinality of the nodes set is the number of cells in the x axis of the map grid. Edges and weight are obtained in the same way as in the Visibility Graph.

D. Potential Fields

The main idea on which this method is based is to replicate the natural potential field, like the ones generated by gravity or charged particles. The agent should be attracted by the goal and repelled by obstacles: the combination of attractive and repulsive forces will move the agent in the space. The classic definition of this method has been modified to allow its usage in a grid-based discrete environment.

For each empty cell of the grid, the attractive potential field is calculated with the formula:

$$Attr_{x,y} = (x - x_{goal})^2 + (y - y_{goal})^2$$

Therefore, the destination corresponds to the global minimum, having his attractive potential field set to zero. On the other hand, every obstacle cell has a potential value set to infinity.

The repulsive field is characterized by two parameters: the distance of influence and the repulsive value. Each obstacle cell create a repulsive force to each empty cell (except for the goal cell) within its distance of influence. The magnitude of this force is given by the repulsive value divided by the distance.

$$Rep_{x,y} = \frac{RepulsiveValue}{\sqrt{(x - x_{ClosestObst})^2 + (y - y_{ClosestObst})^2}}$$

The potential field is the sum, for each cell, of the attractive and repulsive fields.

$$Pot_{x,y} = Attr_{x,y} + Rep_{x,y}$$

To compute the path to the goal the agent iteratively selects, at each step, the adjacent cell with the lowest potential field value. In this project this method has been referred to as *Simple Potential Fields*.

As described, such approach is too simplistic. In fact, the agent ends up in local minimums too often, even with simple configurations of obstacles (see Fig. 6). In order to mitigate these situations, an improved version of the algorithm has been implemented, a variant called *Potential Fields with Memory*, where the agent is able to remember the path that it took to get to its current position. At each step the agent is forced to move to the adjacent cell that has the lowest value between the ones not already in its path.

E. Bug Algorithms

Unlike other path planning algorithms, a Bug approach only assumes to have a local knowledge of the environment, plus destination it must reach. Therefore, it is not possible to establish an *a priori* path, but instead the path will be defined in real time, *i.e. online*. Since the family of Bug algorithms is vast, for the purpose of this work the three most common approaches have been taken into account: Bug v1, Bug v2 and Tangent Bug. Given that at a certain point all the three algorithms assume that the agent follows the perimeter around the obstacle, a key function of the algorithms resulted to be the one that describes how the agent can achieve such behavior, independently from the shape of the obstacle. Such function, called *circumnavigate()* for further reference, is a recursive function whose core behavior is common for all the three algorithms, while the termination conditions changes depending on the case.

As shown in Fig. 8, the function *circumnavigate()* computes the next step based on the direction in which the obstacle is placed with respect to the bug position. The figure represents the simplest case, in which the obstacle consists of a single cell; in case of more complex scenarios the function will

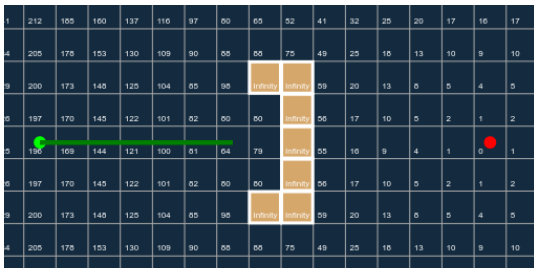
Fig. 6. Example of a Potential Fields algorithm stuck into a local minimum.

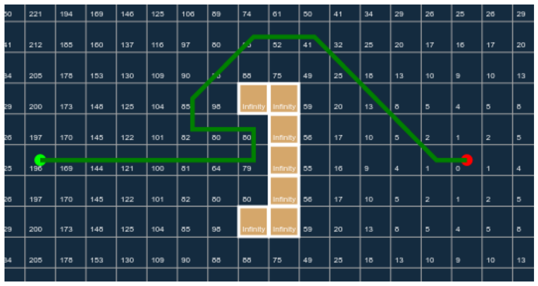
Fig. 7. Potential Fields with Memory can reach the destination in a scenario where Simple Potential Fields would fail (see Fig. 6).

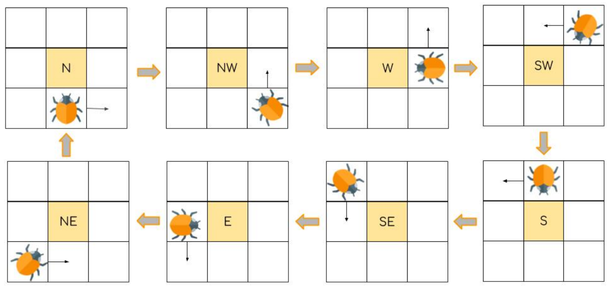

Fig. 8. Agent behavior when meeting an obstacle: depending on his current position and the position of the obstacle, the function defines the steps to cover in order to move along the obstacle perimeter. Note that the following figure shows a clockwise movement, but the procedure can be adapted to move in the opposite way without effort.

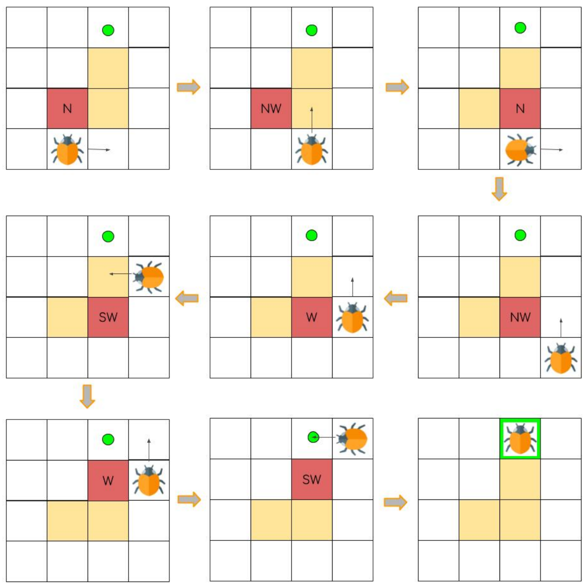

Fig. 9. Bug behavior when meeting a complex obstacle. The red cell indicates the obstacle cell considered at the current time step.

be called recursively, following the path along the obstacle perimeter, as represented in Fig. 9.

The three considered approaches are now described in details.

1) Bug v1: The algorithms can be summarized as follows:

- move by following the simplest path toward the goal (*i.e.* the *dummy path* in the code);
- if an obstacle is encountered, move around it computing at each step the distance to the goal;
- when the obstacle perimeter has been completely explored, return the to nearest point to the goal around the obstacle;
- repeat from step 1 until the goal is reached.

As described above, the termination conditions of the *circumnavigate* function (*circumnavigate1* in the code) are the following:

- the next step corresponds to the goal;
- the next step corresponds to the first that was already covered along the circumnavigation path;

The recursive calls are two, addressing the two remaining cases:

- the next step corresponds to a free cell, therefore the next call of the function will consider as current position of the bug the new step (because he can move) and as obstacle the same of the current recursion step;
- the next step corresponds to an obstacle cell, therefore the next call of the function will consider as current position the last one (because he cannot actually move), and as obstacle the just found next step (because it is a wall).

Algorithm 1 shows the pseudo-code of *circumnavigate1*.

Since an agent can encounter more obstacles along the way, *circumnavigate1* is called at every encounter, and when

Algorithm 1: Circumnavigate1(lastStep, obstacle, end, res)

```
compute newStep along the obstacle;
compute the distance to the goal;
add the newStep and the distance to the result;
if newStep is the goal or the circumnavigation is complete then
    return res;
else
    if newStep is not an obstacle then
        return circumnavigate1(newStep, obstacle, end, res);
    else
        remove the last res entry;
return circumnavigate1(lastStep, newStep, end, res);
```

it returns the *dummy path* from the point in which the bug stops to follow the obstacle to the destination is computed and followed until a new obstacle or the destination itself is reached.

2) Bug v2: This algorithm is very similar to the previous one, but it has some improvement in the length of the goal path. It can be summarized as follows:

- step forward, following the simplest path toward the goal (*dummy path*;
- if an obstacle is encountered, move along its perimeter until the *dummy path* is reached again;
- repeat from the first step until the goal is reached.

This means that the termination conditions of the *circumnavigate* function (*circumnavigate2* in the code) in this case are following:

- the next step corresponds the goal;
- the next step is included in the dummy path;

While the recursive calls are the same of Bug v1.

Algorithm 2 shows the pseudo-code of *circumnavigate2*, which is easier than the previous one. In the pseudo-code, *oldPath* represents the *dummyPath*.

Similarly to v1, Bug v2 may call *circumnavigate2* many times, as there can be many obstacles along the path. Unlike v1, it is now easier to restart moving over the *dummyPath*, since the path will not mutate after the circumnavigation.

3) Tangent Bug: This is the most complex approach among this family of algorithms. What really distinguishes it from the others is the assumption that the bug is capable of sensing the area around it within a given range. Starting from the assumption that the environment is defined by a grid, the sensed area will be defined by a square around the bug as represented in Fig. 10. In the image it is possible to notice that all the cells within the sensing range have a distance from the bug $<= r$ (2 in this case).

The behavior of this algorithm can be summarized as follows:

- follow the *dummy path* until an obstacle is sensed;

Algorithm 2: Circumnavigate1(lastStep, obstacle, newPath, oldPath)

```
compute newStep along the obstacle;
add the newStep to the newPath // newPath is the path
  around the obstacle;
if newStep is in oldPath then
    return newPath;
else
    if newStep is not an obstacle then
        return circumnavigate2(newStep, obstacle,
          newPath, oldPath);
    else
        remove the last entry from newPath;
return circumnavigate1(lastStep, newStep, newPath,
  oldPath);
```

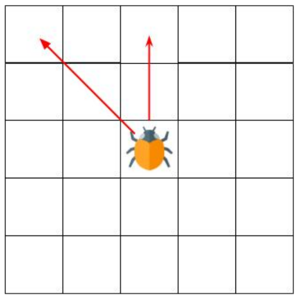

Fig. 10. The range area sensed by the agent, with r = 2.

- detect the point(s) of discontinuity;
- compute the heuristic distance to the goal for each discontinuity;
- move to the discontinuity point identified by the minimum heuristic distance;
- when the discontinuity is reached, move along the obstacle boundary;
- while following the obstacle perimeter, compute the shortest distance from the points in the range of the agent to the destination (*dReach*), and also the shortest distance from the points on the perimeter to the destination (*dFollow*);
- when *dReach* becomes lower than *dFollow*, compute the new *dummy path* to the destination and start over.

The heuristic distance is computed as the sum between the distances from the bug to the discontinuity points and from the discontinuity points to the destination.

Since the grid assumption for our environment causes a non optimal behavior, the procedure has been slightly modified, making *dReach* being computed as the distance between the last step and the destination and *dFollow* as the distance between the last considered obstacle cell and the destination. The perimeter is followed until *dReach* $<$ *dFollow* and there are no obstacles along the *dummyPath* from the actual position to the goal is free for at least one step. Algorithms 3 and 4 show the pseudo-code of tangent bug. The function *boundaryFollow* is equivalent to the *circumnavigate*, but adapted for this approach.

Algorithm 3: tangentBug(dummyPath)

```
pathToDest = [];
while not reached dst do
    compute range area;
    compute discontinuities;
    if disconties are blocking the dummyPath then
        find the discontinuity at minimum heuristic
          distance;
        go straith to that discontinuity;
        compute the best direction to follow the
          obstacle;
        boundaryFollow(lastStepBeforDiscontinuity,
          minDiscontinuity, direction, pathToDest);
        dummyPath = findDummyPath(lastStep, dest)
          //last step is the last step along the boundary;
    else
        pathToDest.push(currentStep);
return pathToDest
```

Algorithm 4: boundaryFollow(lastStep, obstacle, destination, dir, path)

```
compute newStep along the perimeter obstacle;
if newStep is not an obstacle then
    path.push(newStep);
    if newStep = destination then
        return path;
    dummy = findDummyPath(newStep, destination);
    range = rageArea(newStep, 1) // 1 is r;
    dFollow = pathCost(obstacle,destination);
    dReach = pathCost(newStep, destination);
    if not obstacleInPath(dummy, range) AND dReach
      ¡ dFollow then
        return path
    return boundaryFollow(newStep, obstacle,
      destination, dir, path);
else
    return boundaryFollow(lastStep, newStep,
      destination, dire, path);
```

V. Performance evaluation

A. Offline Algorithms

The *Decomposition* methods, as implemented in this work, are obviously complete, since they are always able to find the best path to reach the destination if there is one.

The *Visibility Graph* method is also complete. The downside of this approach is its elevated computational cost. In fact, it is necessary to scan every obstacle cell in order to discover the

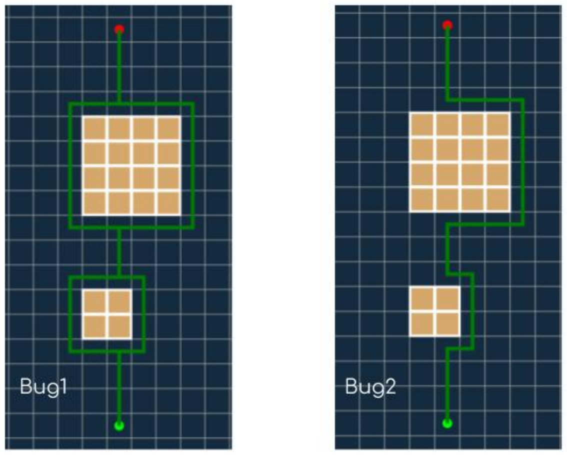

Fig. 11. Bug v1 and Bug v2, compared in the same environment.

obstacle vertex points, then build the *dummy path* between each pair of vertices and finally apply a shortest path algorithm.

The *Probabilistic Roadmap* approach tries to mitigate this problem by randomly selecting nodes of the graph. However, it still has to create all the *dummy paths* that are used as the edges of the graph. Obviously, the path obtained using this algorithm is not guaranteed to be the best possible; in fact, it may even not find a path even if there is one.

There is a tradeoff between the time needed to compute and the number of random samples in the map: increasing the number of nodes increases the chance to find a shortest path, but requires more computational time to create the edges.

Simple Potential Fields has proven to be too weak to be used without any adjustment. The agent gets stuck in local minimums even in scenarios with just a few obstacles.

The reason above is exactly why *Potential Fields with Memory* has been implemented. It is just a simple addition to the Potential Fields approach, but shows significant improvements. Strong and weak scenarios can also heavily depend on repulsive values and distance of influence of obstacles.

B. Online Algorithms

Bug v1, despite using a naive approach and therefore being non optimal, is complete; compared to Bug v2, the latter is able to find shorter paths, as shown in Fig. 11. Nevertheless, this performance improvement has a cost, since in some configuration Bug v2 can get stuck in loop while searching for path to the destination; an example is shown in Fig. 12.

The implementation of Tangent Bug can find even shorter paths compared to Bug v2 (as shown in Fig. 13), but similarly it can sometimes get stuck in loops. Also, it can happen that the goal path found by Tangent Bug results longer than the one found by Bug v2, as shown in Fig. 14.

VI. Conclusions

In this report we described our implementation and analysis of the most popular path planning algorithms and approaches.

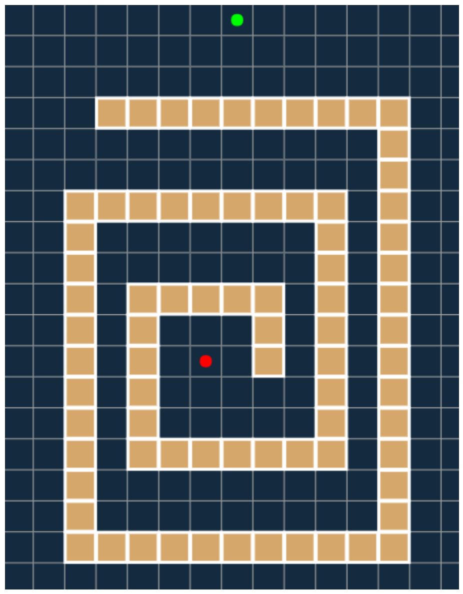

Fig. 12. An example of configuration in which Bug v1 can find a path, while Bug v2 can't.

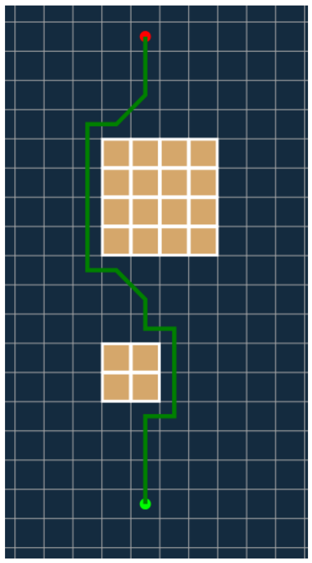

Fig. 13. Tangent Bug, applied in the same configuration shown in Fig. 11, results in a shorter path.

The output of our work resulted to be **Path Planner** [1], a JavaScript application which allows to experiment and tests the different approaches in a practical way over bi-dimensional grid-based environment.

On the other hand, our work allowed us to gain knowledge and to better understand and test different path planning approaches. Both the software produced and this knowledge will be used to accomplish the second part of our project, which aims at implementing a path planning methodology in a rover equipped with sensors, moving in a real-world environment with obstacles.

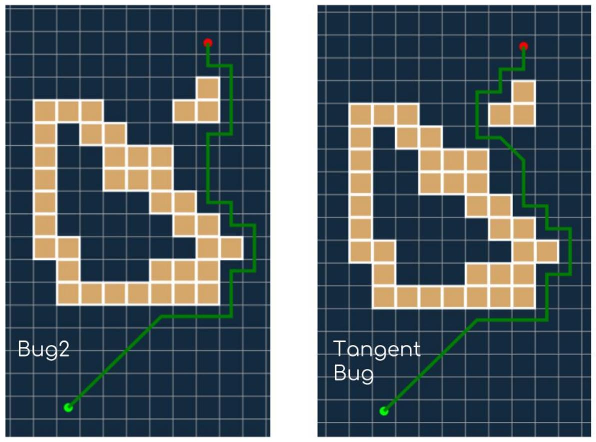

Fig. 14. Bug v2 and Tangent Bug in the same configuration.

References

[1] Carlo Stoemo Alessandro Zini, Filippo Morselli. Path planner. https://github.com/aleeraser/Path-planner.

[2] Mikola Lysenko. L1 path finder. http://mikolalysenko.github.io/l1-path-finder/www/.

[3] Xueqiao Xu. Pathfinding.js. https://qiao.github.io/PathFinding.js/visual/.

[4] C. W. Warren. Global path planning using artificial potential fields. In *Proceedings, 1989 International Conference on Robotics and Automation*, pages 316–321 vol.1, May 1989.

[5] N. Sariff and N. Buniyamin. An overview of autonomous mobile robot path planning algorithms. In *2006 4th Student Conference on Research and Development*, pages 183–188, June 2006.

[6] Han Ul Yoon. Probabilistic roadmap method. Technical report, University of Illinois Urbana-Champaign.

[7] Roland Geraerts and Mark H. Overmars. *A Comparative Study of Probabilistic Roadmap Planners*, pages 43–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[8] Z. Dodds H. Choset, G.D. Hager. Robotic motion planning: Bug algorithms. https://www.cs.cmu.edu/ motionplanning/lecture/Chap2Bug-Alg_howie.pdf.

[9] James Ng and Thomas Braeunl. Performance comparison of bug navigation algorithms. *Journal of Intelligent and Robotic Systems*, 50(1):73–84, Sep 2007.