Implementation notes about A*, Incremental GVD, Boustrophedon and RRT

Arthur H. D. Nunes[1], and Álvaro R. Araújo[1]

***Abstract*— In this work we implemented four robot motion planning algorithms to a differential drive robot: (i) A*; (ii) Incremental GVD; (iii) Boustrophedon; (iv) RRT. The implementations were made in StageROS ROS simulator with a ROS in Docker architecture. We described the implementation guidelines and present the results.**

***Index Terms*— Implementantion notes, ROS in Docker, tangent bug, potential fields, wave front.**

I. INTRODUCTION

In this work we implemented four robot motion planning algorithms:

- A*;
- Incremental GVD;
- Boustrophedon;
- RRT.

We considered a differential drive robot that can move in a two-dimensional workspace with obstacles. The goal of the first and the last algorithm is to guide the robot from a initial configuration to an goal configuration. For the second algorithm the goal is to map the workspace by constructing the incremental GVD. Finally, the goal of the third algorithm is to cover the entire workspace.

The implementations were made in StageROS ROS simulator with a ROS in Docker architecture.

In the following section, II, we describe the simulation architecture and the four motion planning problems definitions. Then, in Section III we describe the implementation guidelines and strategies that we used to achieve the completion of them. Next, in Section IV we show the simulation results. Finally we conclude in Section V.

II. SETUP

A. Simulation Architecture

The simulations presented in this work was based in Robot Operating System (ROS). Inside ROS, we used StageROS simulator which allows us to control a differential drive robot in a planar space with obstacles. This simulator is good for multiple robots, beginners at ROS and for simpler simulation scenarios. As we are not interested in complex physic interactions, irregular workspaces, non-euclidian spaces or tri-dimensional simulations, our case fits in the third selection, where we want a basic planar simulator to test the four mentioned motion planning algorithms.

Furthermore, we also used Docker Containers as part of the simulation architecture. Docker is a software that performs Operating System level virtualization to deliver software in packages. The intention to use ROS inside Docker is to minimize the effort of setting up the simulation environment by creating a container model, called container image. By creating our custom ROS image, one can assure that the software will incorporate its dependencies in it and therefore installations and configuration are no longer required, making a *ready-to-use* environment.

In order to access GUI provided by the container, we use a noVNC approach that is already configured inside the container image. With the noVNC running, just opening the web browse is enough to access the container GUI. Figure 1 shows the basic background for a ROS in docker simulation. Another key point to mention is that Docker containers can also be used to deploy applications, using the same container images and therefore the same software version.

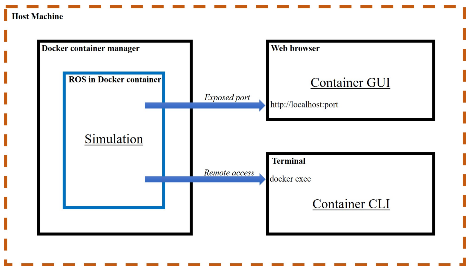

Fig. 1. Diagram to show ROS in Docker simulation architecture.

B. Problem Definitions

1) A:* The A* (*A star*) algorithm is a path planner that computes a route between a given start configuration and a given goal configuration in a any-dimensional problem using a graph search in discrete representation of the environment.

2) GVD: The GVD (*Generalized Voronoi Diagram*) is a roadmap algorithm that consists of the set of points where the distance to the two closest obstacles is the same. Path planning is achieved by moving away from the current position until reaching the GVD, then along the double equidistant GVD to the vicinity of the goal, and then from the GVD to the goal. In this work we address the problem of incrementally constructing the GVD using a range sensor.

3) Boustrophedon: The Boustrophedon algorithm aims to solve a covering task, where the robot aims to cover all the free workspace area. With this algorithm, the goal is to decompose the environment in cells and to drive the robot up and down in each cell.

4) RRT: The RRT (*Rapidly-exploring Random Trees*) algorithm consists of a sampling-based planner, or probabilistic roadmap. This kind of algorithm relies on a procedure that can decide whether a given configuration of the robot is in collision with the obstacles or not. In some sense, sampling-based planners have very limited access to the configuration space, also efficient collision detection procedures ease their implementation and increase the range of their applicability.

III. METHODOLOGY

A. A*

The methodology used in this work first consider the bitmap image as the discrete representation of the environment, in which each pixel is assigned to one node in an graph. If one pixel is at top, bottom, right or left from another pixel and none of them is an obstacle pixel, a connection between the nodes representing those pixels is created in the graph.

We considered the 4-neighbors (up, down, right, left) as oppose to 8-neighbors (with diagonals too) to avoid making the robot hit the edges of the obstacles.

When the algorithm is launch, it first read the image and store in a matrix data structure. To find the path, the planner starts in the pixel that corresponds to the robot initial position. Then, it iterative searches and constructs the graph by searching each pixel neighbor.

While iterating in the search, the planner assigns a cost to each node (pixel). The cost is composed by two terms. The first is a value that starts from 0 and is incremented in each neighbor search. The second is an heuristic cost defined as the norm of the distance between the pixel and the goal pixel.

The path is found by searching the combinations of pixels that goes from the initial pixel to the goal pixel with the lowest cost.

If we consider just the first term of the cost function, we achieve the Dijkstra algorithm. The second term is what differs Dijkstra from A* and it serves to speed up the algorithm search as it doesn't need to search the entire graph anymore.

To drive the robot for the given path, we use a waypoint strategy, where we compute a sequence of waypoints corresponding to the pixels in the path.

B. GVD

To incrementally construct the GVD the first step is to move away from the closest obstacle to the robot until it is equidistant to two different obstacles, then the goal is to move the robot in a path perpendicular to the line segments that correspond to the minimum distance measurements of the range sensor equipped to the robot.

In order to identify a minimum measurement we compare every sensor measurement i with sensor measurements $i-1$ and $i+1$ in the array returned by the sensor, if distance measurement i is smaller than the other two it is a minimum measurement. Figure 2 from [1] illustrates that logic. Also an additional step was take to compute if each minimum found is at least 25 samples away in the range sensor for it to be considered in a different obstacle.

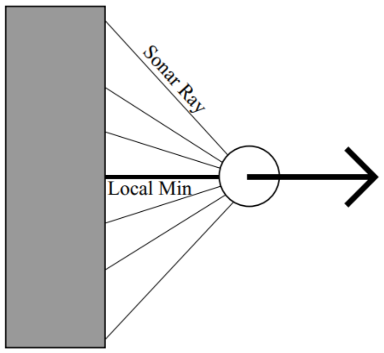

Fig. 2. Illustration for minimum measurement computation.

Once the robot moves away from the closest obstacle and accesses the GVD, it must incrementally trace the GVD by moving in a line orthogonal to $\nabla d_i(q) - \nabla d_j(q)$. When the robot encounters a meet point, a point where it is simultaneously close to more than two obstacles, it marks off the direction from which it came as explored, and then identifies all new GVD edges that emanate from it. From the meet point, the robot explores a new GVD edge until it detects either another meet point or a boundary point. In the case that it detects another new meet point, the above branching process recursively repeats.

C. Boustrophedon

The algorithm receives as input the environment map as a bitmap image, as in Figure 3, and the initial robot position.

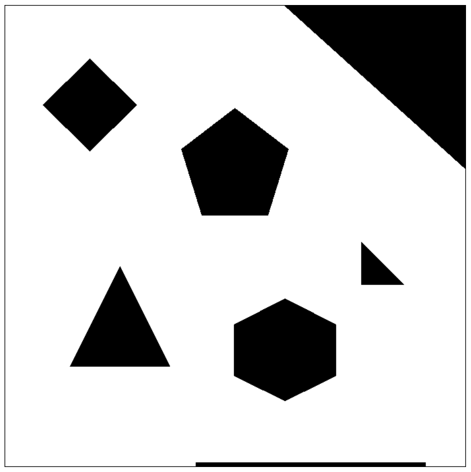

Fig. 3. Map used for Boustrophedon simulation.

The first step is to decompose the image in cells. Each cell is composed of one or more trapezoids. THe cell decomposition is achieved by tracing an upward and an downward line in each obstacle vertex that allows it. In order to do that in a computational way, we iterative analyze the image by its columns. In a given column, each sequence of white pixels corresponds to a cell. When comparing to the previous column cells the algorithm can determine the current cell. The result of this first step is shown in Figure 4 where each cell is colored by a random color. The cells are also numbered starting from the left and going to the right, giving a lower number to a top cell when two cells divide the same start column.

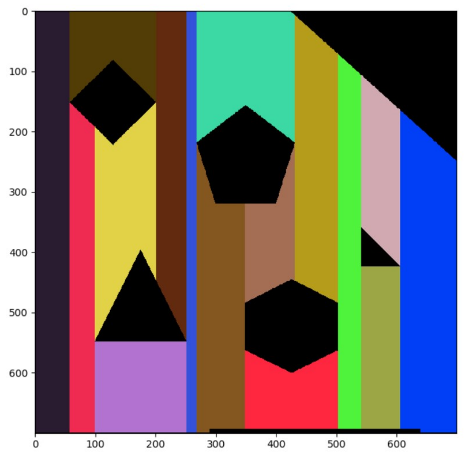

Fig. 4. Cell decomposed map for Boustrophedon simulation.

The second step is to find the cells cover order. In order to do this, we perform and recursive search starting with the cell that the robot starts. Then, the recursive method adds the current cell at the plan and for each neighbor cell not at the plan yet the method is called again. A draft of the cell order when the robot spawns at the middle of the map is shown in Figure 5. It is important to notice that the cell order will not necessarily ever point to neighbors cells. Therefore a cell-to-cell planner is needed.

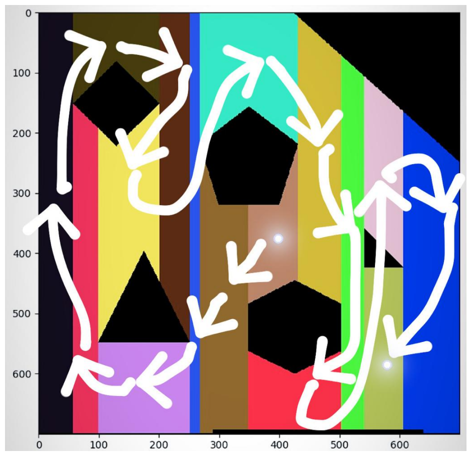

Fig. 5. Cell decomposed order for Boustrophedon simulation.

The third step is to plan the cell-to-cell motion, as the global plan may not include neighbors cells and the direction may vary to right or left. This is achieved by a BFS search to find the shortest path between two cells and in each cell waypoints at the middle of the cells are computed to achieve the motion.

Finally, the last step is to compute the Boustrophedon-like movement inside each cell. This is achieved also by using a waypoint strategy.

Another strategy used was to add a repulsive vector that acts very closer to the obstacles to avoid hitting obstacle edges that may appear at cell transisition.

D. RRT

The RRT grows a tree rooted at the starting configuration of the robot by using random samples from the work space. As each sample is drawn, a connection is attempted between it and the nearest state in the tree, if the connection is possible, hence it is collision free, a new state is added to the tree. The length of the connections between each node of the tree is determined by a growth factor, and a uniform distribution is used in order to achieve new samples of the space.

The environment map is received as an input in the format of a bitmap image. For the implementation it was chosen to use Dubins curves in order to draw the edges of the RRT, since this type of curve provides a smooth path from each node of the tree to the others. Also, this kind of path complies with the restrictions of the differential drive robot.

For the random expansion of the tree, just the x and y coordinates of the nodes are sampled from an uniform distribution , the orientation of the robot is always set to 0 degrees in every configuration. Before adding each node to the tree the algorithm finds the nearest existing node, checks if it is within the growth factor range of the tree, and then checks if it is collision free.

Once the desired configuration is added to the tree, the path is computed starting in the goal configuration and going backward in the tree finding the next node that most decreases the distance between the position currently being analyzed and the robot's starting position, this node is added to a list and that process repeats until reaching starting position. The list generated is reversed, and now it contains the path found by the RRT planner. This path is passed to the local control algorithm in order to drive the robot from the initial configuration to the goal.

IV. RESULTS

A. A*

The A* algorithm successfully navigate the robot from the initial position to the goal in the shortest path in the graph. Figure 6 shows the path of the robot when navigated by A* in a given map.

B. GVD

The GVD algorithm successfully managed to incrementally construct the roadmap while the robot was navigating through the workspace. Figure 7 shows the path of the robot during a test in a certain map, and Figure 8 shows the GVD for that particular map generated analyzing the bitmap image for it.

C. Boustrophedon

This algorithm could drive the robot to cover the environment are with success. Figure 9 shows the result of the algorithm in the map used in the methodology section. This

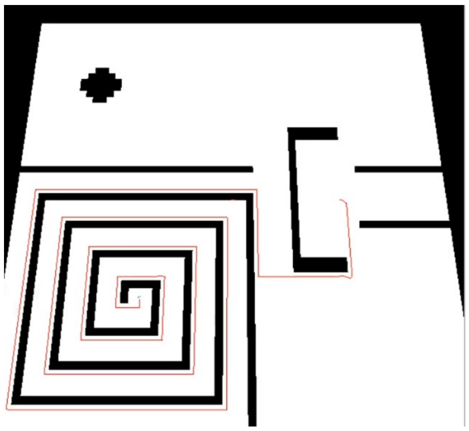
Fig. 6. A* result.

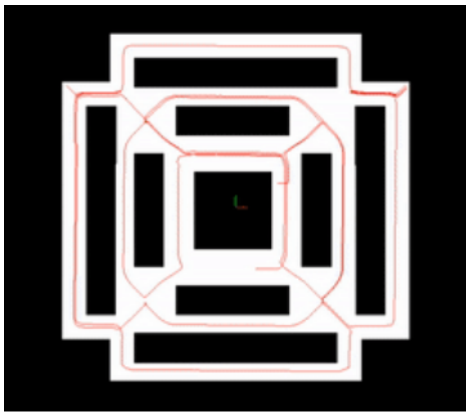
Fig. 7. GVD result.

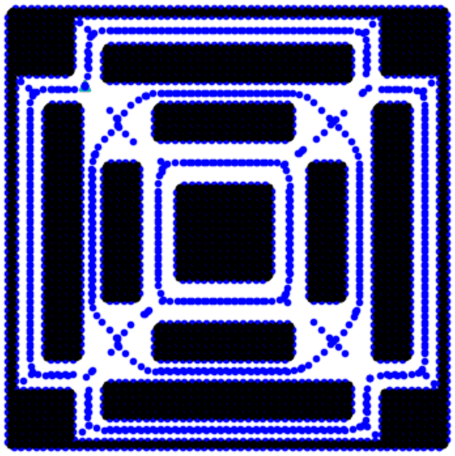
Fig. 8. GVD for the test map.

was the slowest simulation, what is expected because the robot needs to travel along the entire map.

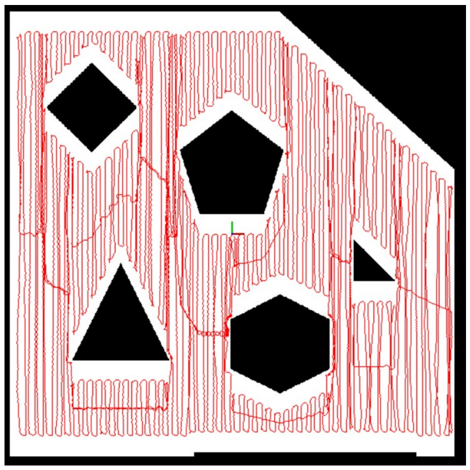
Fig. 9. Boustrophedon result.

D. RRT

The RRT algorithm could plan the path from starting configuration to goal configuration successfully, and the local feedback linearization control drove the robot through the path smoothly. Since this algorithm relies on the random sampling of an uniform distribution, sometimes the planner took more iterations, and therefore more time, in order to find a path. Figure 10 shows the result obtained by the planner in approximately 700 iterations, and Figure 11 the result for approximately 2000 iterations.

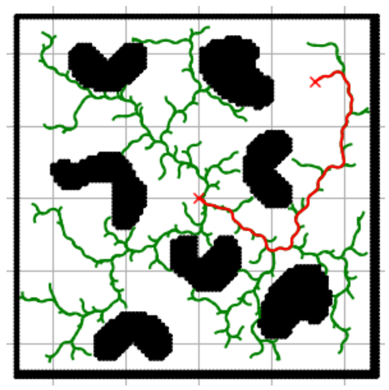
Fig. 10. RRT result achieved in 700 iterations.

V. CONCLUSIONS

For the A* algorithm, it accomplished success in all simulations. Although, when compared to our previous works that implemented the Wave Front algorithm, A* didn't show real improvements. In fact, when implemented Wave Front, the technique to navigate the robot in the generated Vector Field was to assign the velocity vector depending on what pixel the robot was currently at. This is an advantage from the waypoint strategy used in A* because the former can drive the robot from whatever pixel, even though the robot

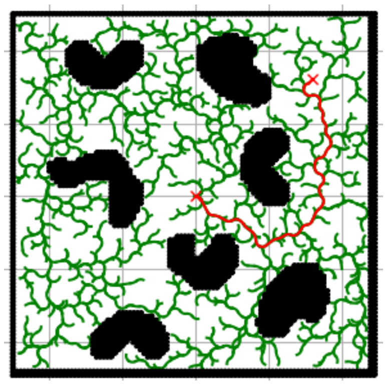

Fig. 11. RRT result achieved in 2000 iterations.

is kidnapped or has moved with pixel precision error, when the latter, in the other hand, can only take account of the found waypoints.

The GVD algorithm was one of the hardest to implement in this work, since it required some sophisticated abstractions in order to drive a differential robot using a range sensor in such a restrictive path. Although it made some good results, managing to navigate the robot inside the GVD all the time, sometimes some number of edges were not navigated by the robot during the tests. In that sense, the logic implemented to store the meetpoints data could be improved for a more robust implementation.

For the Boustrophedon algorithm, it could cover the entire map. Although, it was the hardest implementation to debug in this work due the time required to simulate in comparison to other implementations. Also, the given strategies to avoid colliding with the obstacles edge was to avoid reaching close to obstacles while doing the Boustrophedon-like motion and the vector fields. But those strategies took time to calibrate and better strategies may be developed for future implementations, for example using tangent bug approach.

Finally, for RRT algorithm the method achieved good results, but it's true efficiency in planning is very much tied to it's random nature and the path may take many iterations to be found. But for the implementation done in this work, the time spent to find a path could be greatly shortened using methods to bias the sampling towards the goal or adjust the step size of the tree.

REFERENCES

[1] Howie M Choset, Seth Hutchinson, Kevin M Lynch, George Kantor, Wolfram Burgard, Lydia E Kavraki, and Sebastian Thrun. *Principles of robot motion: theory, algorithms, and implementation*. MIT press, 2005.