Assignment 2
Map Maker

Artificial Intelligence - Methods and Applications, 5DV122

Summer 2020

Authors:
Jakub Łukowiak
Username: mrc16jlk

Teacher:
Ola Ringdahl

Content:

1. Problem Specification

The task is to create a program for a hybrid architecture robot in the simulation environment Microsoft Robotics Developer Studio 4. The robot's task is to map the surroundings using the laser scanner and save it to an image. The robot has to map the whole available environment, move within it and avoid any obstacles. The map at the end should contain no undiscovered areas, it has to be complete. Also a path planning algorithm is required so the robot can move around the environment to not leave any undiscovered areas. Basically the robot has to create a map totally on its own, without any human interference. The hybrid architecture applied in this assignment is described in details in the following section below.

2. Usage and user's guide

There are six different python files and one shell script. Below the short description of each is provided.

- **Mapper** - the shell script for running the program
- **main.py** - the main program file responsible for running the main_loop function and initializing the program in _init function. The main loop function runs all the other classes, finds the closest frontier to discover and feeds the path obtained from A* algorithm to the navigator and closes the program when there are no other frontiers to discover.
- **Planner.py** - PathPlanner class, responsible for creating a path from the robot to the target (for example a frontier)
- **Navigator.py** - the file with the class Navigator, responsible for following the path fed by main.py
- **Cartographer.py** - Cartographer class, responsible for scanning, creating the grid, enlarging the obstacles, finding frontiers. Additionally it also includes some helper functions for calculating a point position in different coordinate systems (robot, grid and world coordinate systems respectively)
- **Communicator_robot.py** - Communicator_robot class responsible for communication with the MRDS environment.
- **help_functions.py** - a collection of useful functions such as calculating the distance between some points, adding neighbouring points within a certain range to the banned targets list and so on.

The program can be run on itchy (from itchy:~/edu/lab2_final>) using the following Linux terminal command:

```
./mapper.sh url x1 y1 x2 y2 enableGUI
```

url, x1, x2, y1, y2 are the input parameters. They stand for:

- url - the address and port to the machine running MRDS
- x1, x2, y1, y2 - coordinates of the lower left and upper right corners of the area the robot should explore and map in the Factory environment.

To run the program the Python Image Library is needed, either *PIL*[1] or *Pillow*[2].

[1] http://www.pythonware.com/products/pil/
[2] https://pillow.readthedocs.io/en/4.0.x/

3. Algorithm Description

The program runs in main.py file. When run the first function called is _init function that returns the map (grid) and the map display class object used to display the map. It also does the first initial scan of the map while the robot rotates a few times. The map is also displayed during the process.

The two objects are then forwarded to main_loop function

3.1 Main loop

In the main loop the following algorithm is performed:

1. Create an empty list for banned positions and enter the loop
2. Create an empty list for frontier centroids
3. Get the robot position in the world coordinate system, calculate the position in the map coordinate system
4. Enlarge obstacles on the map (see Chapter 3.3 for more details).
5. Get a list of list. The inner lists consist of frontier points belonging to the same frontier from the map with enlarged obstacles (see Chapter 3.4 for more details).
6. Create centroids from the inner lists. Search for the closest centroid that is at the same time not on the banned points list. If no centroids are available save the map and close the application.
7. Plan the path to the selected centroid using A* planner (see Chapter 3.5 for more details). The path has to be reversed, the original one has the last point first.
8. Prepare the final path by adding every third point from the original path to a list. Feed the list to the navigator (see Chapter 3.6 for more details).
9. Repeat starting from point 2. The only way to break the loop is by finding no frontiers to explore on the map.

In between those points scanning using the laser sensor and creating a map from those reading is taking place whenever possible and/or necessary. For more details on how the map is created see Chapter 3.2.

3.2 Scanning the area and creating the grid

Scanning:
In the scanning function in Cartographer class the robot scans the area with a laser sensor and updates the representation of the world saved as a cspace grid. For obtaining the information whether the cell is empty or consists of an obstacle, the HIMM algorithm[3] is used. The algorithm is as following:

1. Scan the area with a laser scanner
2. Update the grid with scanned information:
   For every scanned angle:
   2.1. Count the coordinates of the obstacle.
   2.2. Perform the Bresenham algorithm for updating the grid cells along the line between the robot and obstacle[4].
      2.2.1. For every cell before obstacle:
         2.2.1.1. Decrease its value by 1, unless it is equal 0.
      2.2.2. For the cell with obstacle:
         2.2.2.1. Increase its value by 1, unless it is equal 15.

The grid created consists values from 1 to 15. 1 is the highest possible probability of not containing an obstacle while 15 is the opposite. When the grid is initialized it consists only of values equal to seven. The values are then scaled to values from 0 to 255 (gray scale) which are displayed using the provided Python code. Each pixel represents one grid cell. The size of the grid obviously depends on the size of the area which the robot needs to explore and the resolution. Throughout various runs it has been decided to stick to 0.5 resolution which means that each meter is represented by 2 pixels on the map. With this resolution it is possible for the robot to use the map without running into any problems, a user can clearly see all the obstacles while looking at the map and no visible performance issues occur.

3.3 Obstacle enlargement

Obstacle enlargement is a feature that allows rather simple path following algorithm to be effective without using any obstacle avoidance. During development of this program a problematic issue of cutting the corners has been encountered. To solve it, instead of modifying the existing path following algorithm another work-around has been implemented. Every obstacle is enlarged by 2 pixels in every direction. This somehow takes robot physical size into consideration while creating paths thus leading to crashless navigation.

3.4 Finding frontiers

Frontiers are points which are bordering known and unknown regions of a map. To find those it is not always easy and many different methods have been found. In this program

[3] https://pdfs.semanticscholar.org/1acb/287b825916f7677f1a9092fae9e8eb485c28.pdf
[4] https://en.wikipedia.org/wiki/Bresenham's_line_algorithm

finding frontiers is based on Wavefront Frontier Detector[5] which is a breadth-first search. The algorithm finds all the frontiers however it does not scan the whole map. It only scans the area of the map which have been visited by the robot, boosting the overall performance. If the discovered area grows very big the performance will obviously decrease – yet it remained on acceptable levels throughout testing.

In order to avoid scanning the whole grid the points are marked as:

- Map open list – points that have been already enqueued by the outermost search
- Map closed list – points that have been already dequeued by the outermost search
- Frontier open list – points that have been enqueued by the frontier extraction
- Frontier closed list – points that have been already dequeued by the frontier extraction

The marks above indicate the status of each map point and determine if it should be taken into account in a given time.

If a frontier point is found all the neighbouring frontier points are added – this is because of frontier points interconnectivity. When all the neighbouring frontier points are added the process continues for the neighbours' neighbours and so on. When all the points are gathered they are added to a set data structure. The set contains all subsets of frontier points.

The final point that is forwarded to A* is called a centroid. Centroids are simply centres of gravity of figures created from frontiers. The centroid forwarded to A* planning algorithm is the one closest to the robot. Since moving the robot around the map takes considerable amount of time this feels like the best solution timewise.

3.5 Finding and creating the path with A*

A* is an informed search algorithm, starting from a starting node of a graph towards the goal node while maintaining the smallest cost – in this assignment that would be distance.

[5] http://www.ifaamas.org/Proceedings/aamas2012/papers/3A_3.pdf

At each iteration the algorithm needs to choose which path should be extended. The decision is based on minimizing:

f(n) = g(n) + h(n)

- *n – is the next node on the path*
- *g(n) – is the cost of the path from the start node to n*
- *h(n) - is the heuristic function that estimates the cost of the cheapest path from n to the goal*

Heuristic function is problem-specific, it should be admissible (it never overestimates the actual cost). Then A* is guaranteed to find the shortest (or the least-coast) path from start to goal. It is similar to Dijkstra's algorithm, which in fact can be described as A* where h(n) is always equal to 0.

The code has been based on http://code.activestate.com/recipes/578919-python-a-pathfinding-with-binary-heap/ and adapted to the rest of the code.

3.6 Moving to a path point:

The main loop prepares the final path by adding every third point to a list. Then the list is reversed since the A* function returns the path from the goal to the robot. The list is then fed to the path follower. The path follower is a follow-the-carrot algorithm. First the algorithm reads the map, the robot position (which is translated into grid coordinates). Then the angle error between the carrot point and the robot's heading is calculated. The final step is to calculate the time required to rotate the robot with 0.2 angular speed to reduce the angle error to zero. If the rotation time is very low the robot just stops for 0.2 seconds. This helps to avoid spamming the MRDS with position requests which sometimes could completely "jam" it. In order to make it work faster the navigator always chooses the shortest path for rotation – it will always pick the smaller angle to rotate. When the rotation is finished the robot drives forward for a short period of time. The robot barely steers away from the desired path which makes it simple yet effective way of exploring the map. Due to obstacle enlargement the path follower never crashes into any objects. Before implementing this work-around the robot used to cut the corners which is one of the cons of follow-the-carrot algorithms in general. The map is updated while the algorithm is performing desired rotations.

4. Systems Description

The following chapter is focused on describing the project main program architecture.

4.1 System architecture

The system consists of a few classes in order to meet the requirement of having different modules. The projects has a Planner module (planner_revised.py), a Communicator module (communicator_robot.py), a Navigator module (navigator.py), a Mapper module (show_map.py) and a Cartographer module (Cartographer.py). Additionally a package of help functions (help_functions.py) is available to use it whenever necessary. The main Python file connects them all together.

4.2 Planner:

- Astr(self, map, start, goal): - performs A* search for the most optimal path on a map provided. A start point and a goal point have to be provided as well. It returns either None or when the finishing conditions are met returns construct_path function.
- Construct_path(self, came_from, current): - constructs a path consisting of points to use for the navigator

4.3 Communicator:

The communicator class has not been modified and is used as provided on Cambro. It used to send requests to the MRDS http interface to get information regarding the robot, sensor and so on.

4.4 Navigator:

- Robot_to_goal(map, x_goal, y_goal, display) -
- Angle_diff(goal_x, goal_y) – calculates the angle error between the robot's orientation and a goal (goal_x, goal_y)

4.5 Mapper:

The mapper class has been barely modified. Two functions that are heavily based on updateMap have been added. Those functions are responsible for displaying the path and printing the frontiers.

4.6 Cartographer:

This is the biggest class of them all. It has functions for initialization of the grid, converting points from one coordinate system to another, updating the global map using the laser scanner, executing the Bresenham algorithm as well as Growth Rate Operator, finding

frontiers, enlarging the obstacles and last but not the least returning a frontiers centroid coordinates.

4.7 Help functions

This set of functions consists of:

- Quaternion2Euler(array): calculating Euler angles from a quaternion returned by MRDS
- getData(): reading the input data and making accessing it easy in every class
- neighbours_ban(ban_list, range_var, x_ban, y_ban): adding neighbouring points to the ban list (ban_list) within a certain range (range_var). The first banned point is x_ban, y_ban around which extra points are banned as well.
- Grid_to_World(x,y): transformating x,y coordinates from the grid to the world (MRDS) coordinate system.

5. Test runs

The test runs were run separately for different modules. After all those tested, some general ones for whole system were performed.

5.1 Environment scanning and map creating.

Below the pictures created with testing the environment scanning and map creating are presented. The white points stand for no obstacle detected – the closer to pure black the colour is the higher the probability of an obstacle in that very point it. When the frontier centroid is chosen and the path is planned, it is displayed on the map for a few seconds. Then during the robot going to the goal phase the closest carrot point is displayed as green square (as visible below in Figure 5.1.)

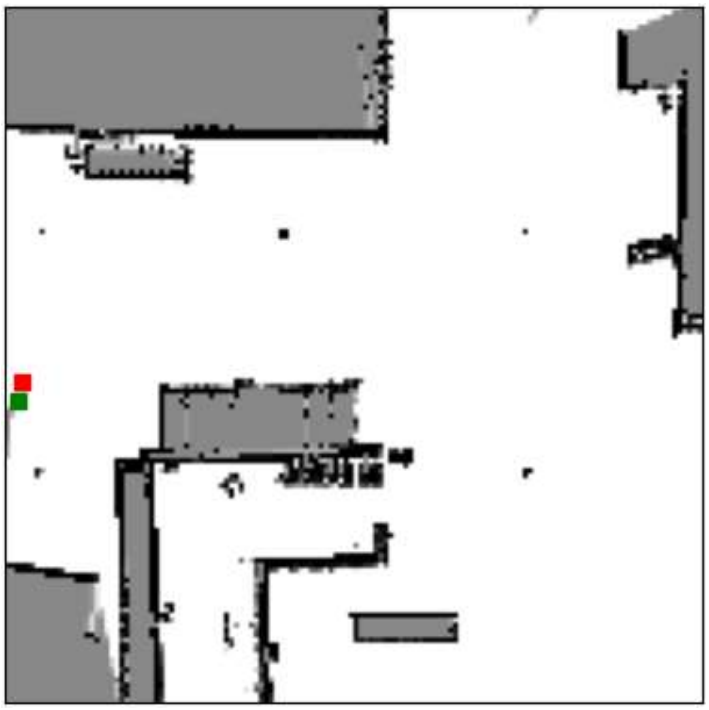

Figure 5.1 Map after discovering all the frontiers picked up by the algorithm. The red square is the robot and the green square is the last carrot point.

5.2 Frontier detection and path planning

Frontier detection works well, with several successful detections below as seen in Figure 5.2, Figure 5.3 and Figure 5.4. The paths found were the shortest possible. The robot is marked as a red square and the path is marked with black, small dots. Keep in mind that the grid displayed is the one with enlarged obstacles.

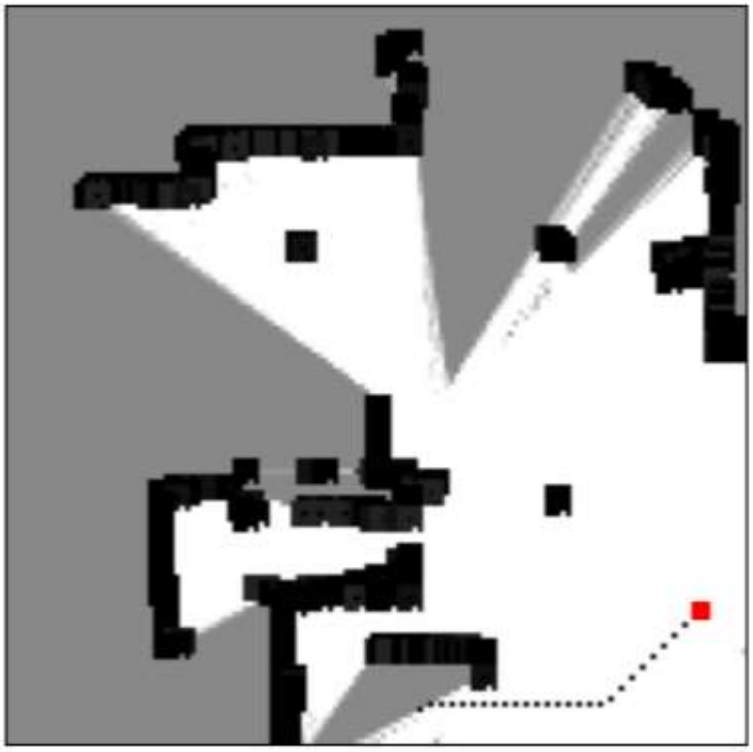

Figure 5.2 A frontier detected with the path displayed as black dots. Keep in mind that this is the map with enlarged obstacles The red square is the robot position.

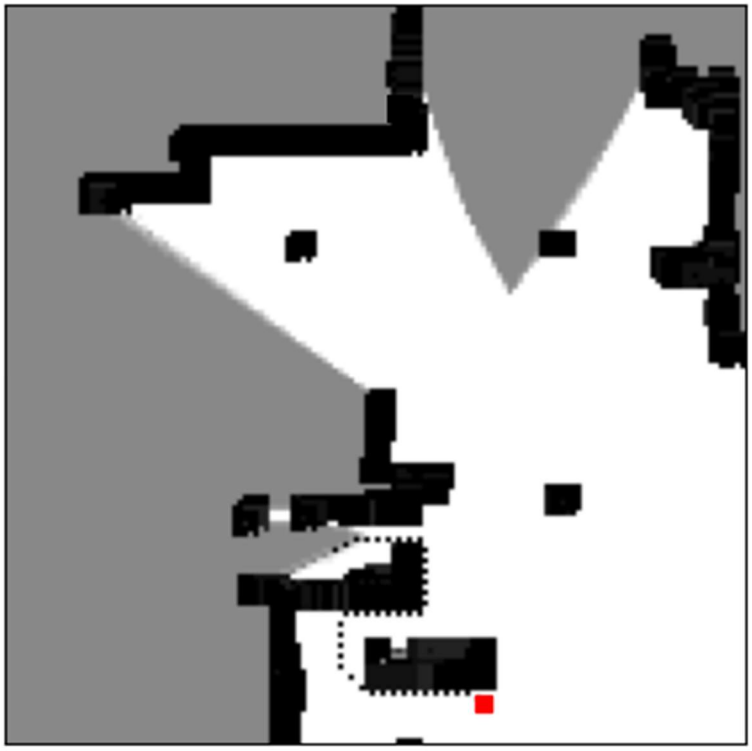

Figure 5.3 A frontier detected with the path displayed as black dots. Keep in mind that this is the map with enlarged obstacles The red square is the robot position.

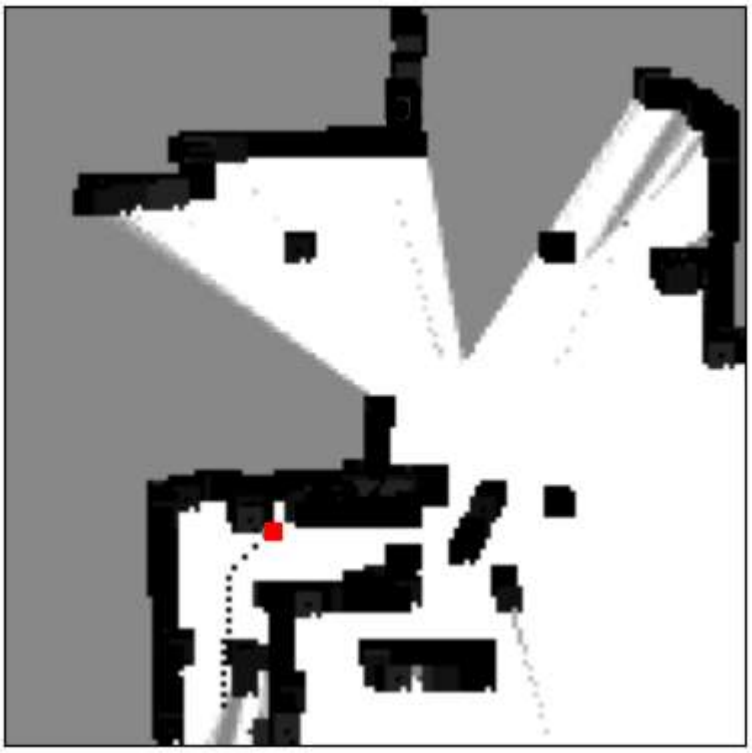

Figure 5.4 A frontier detected with the path displayed as black dots. Keep in mind that this is the map with enlarged obstacles The red square is the robot position.

6. Problems and Reflections

The project has been completed successfully, the robot is able to traverse the environment without crashing, detects frontiers to explore, scans the area using the laser scanner. The scans are then used to create a csgrid and then the grid is displayed as a map using the helper code provided.

The amount of work required to finish the project as a single-person team has been tremendous. Testing all parts of the code required a lot of time and patience. MRDS4 isn't a supported software anymore, luckily no unexpected errors occurred.

An issue with laser scanning was encountered. The issue resulted in big black (of value 15) areas as seen in the figure below. In order to get rid of the issue a quick fix has been implemented: only readings of value 39 or higher are accepted by the HIMM function. So if the obstacle is further than 39 meters away the reading of that specific angle is ignored. This also allowed the readings to be more precise because it could be observed that obstacles detected far away tend to be noisy. The result of not having distance filtering is visible in Figure 6.1

A little bit unexpected problem with show_map.py file provided could be easily solved before putting any time into fixing it. Whenever the map display function is called while using PyCharm on Windows 10 or Windows 7 it freezes and crashes. The solution was really

simple, using console to run the main.py file. Since the file is provided to students and PyCharm is probably one of the most used environments for Python that could be stated in the project description on Cambro. Luckily teachers are here to help and the issue has been solved.

During the first phases of development wavefront path planner has been deployed however due to occurring bugs it has been dropped completely. A* search has been implemented instead which seems to work flawlessly.

The assignment is a very good way to put many different ideas that have been mentioned during the course lectures. Probably it would be a good idea to add more information regarding frontier detection to the lectures to make this assignment a little bit easier.

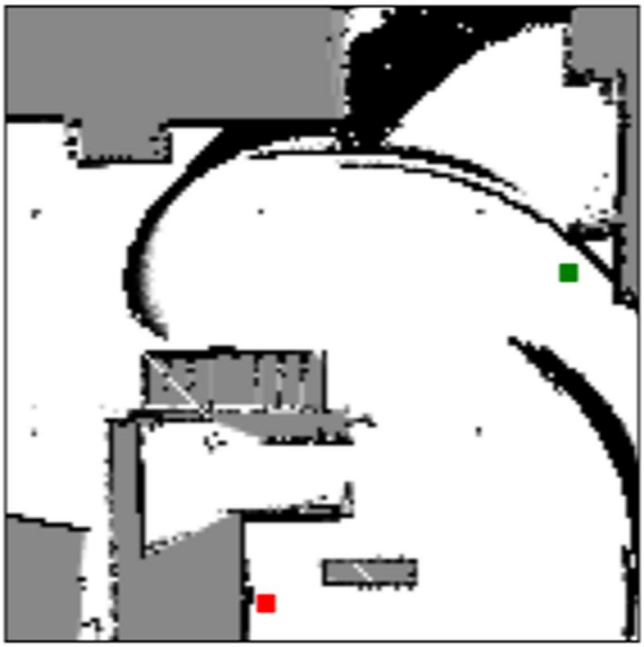

Figure Laser errors 6.1

Nevertheless the project requirements are a little bit vague and it is hard to decide whether the implementation is good enough or not. This could be specified more clearly (time limit, quality of the map etc.)

7. Limitations

During creating the project some limitations have been found. First of all some of algorithm used for searching the values in the grid are not optimal and will last the longer the bigger the map to explore is.

One of the most visible limitations is not using the time during which the robot is going to a centroid. The new frontier or path is not going to be chosen until the robot reaches its

destination. The better solution would be finding a new goal point while going to the already found one in a different thread. This is however once again a trade-off between the project runtime and complexity. Also the development of this project has been limited by time constraints while the field of science connected to finding frontiers, finding the most optimal strategy to discover the map and so on is rapidly developing.

The other limitation is scaling the grid to the surrounding. The smaller the grid cells are, the larger the map and the computations time are. This can be optimized with using a quadtree algorithm for mapping the environment. It does not require so much space for saving all the cells states and still is very precise.

Another problem was dividing the algorithm into separate classes. It took some time to plan the goal of each class and ways of communication between each other.