Umeå University
Department of Computing Science

Artificial intelligence
Methods and applications

5DV181

Map maker

Thomas Ranvier
ens18trr

Valentin Lecompte
ens18vle

supervised by
Ola Ringdahl
Juan Carlos Nieves Sanchez

January 10, 2019

Contents

1 Context **2**
- 1.1 Code structure . . . 2
- 1.2 How to run our program . . . 2

2 Mapping **4**

3 Planning **6**
- 3.1 Find a goal point . . . 6
  - 3.1.1 Detect the frontiers . . . 6
  - 3.1.2 Choosing the goal point . . . 10
- 3.2 Reaching the goal . . . 11
  - 3.2.1 First approach, the potential field . . . 11
  - 3.2.2 Building a path . . . 13

4 General improvements **15**
- 4.1 Inaccessible frontiers . . . 15

5 Optimisation **16**
- 5.1 Cartographer . . . 16
- 5.2 Show map . . . 16
- 5.3 Frontiers limiter . . . 16
- 5.4 Communication with MRDS . . . 16

6 Results **18**

List of algorithms **21**

Bibliography **22**

Chapter 1

Context

1.1 Code structure

Our program is divided in three main modules, the mapping module, the planning module and the controller module. The role of the first one is to build the map of the environment of the robot using the echoes of the lasers. The role of the second one is to plan the goal of the robot and the path that the robot must follow in order to explore the world. The role of the controller module is to make the robot move towards its goal while avoiding the obstacles in its way.

To communicate with the MRDS server we created a class named 'Robot', it is used as an interface to send and receive informations to and from the MRDS server easily. The received informations are directly adapted to our needs and stored using our 'Position' and 'Laser' datastructures in this class.

1.2 How to run our program

For this assignment we used Python 3 as programming language, to run the program run the 'mapper.sh' script with the following syntax:

```
mapper.sh $1 $2 $3 $4 $5 $6
```

The 'mapper.sh' script above launches our program with the 6 parameters.

If you'd rather not use the script from above you can directly launch the 'main.py' python script. The usage of the python program is the following:

```
usage:python3 main.py [-h] url lower_left_pos_x
    lower_left_pos_y upper_right_pos_x upper_right_pos_y show_gui

positional arguments:
  url                 The url of the MRDS server with the port,
                      the format is url:port
```

```
  lower_left_pos_x    The X coordinate of the lower left position.
  lower_left_pos_y    The Y coordinate of the lower left position.
  upper_right_pos_x   The X coordinate of the upper right position.
  upper_right_pos_y   The Y coordinate of the upper right position.
  show_gui            1 if you want to show the GUI, 0 otherwise.

optional arguments:
  -h, --help          show this help message and exit
```

Chapter 2

Mapping

The first point on which we worked was to find a way of building a map of the environment of the robot using the lasers echoes.

To do so we created a class 'Map' that contains a grid of values between 0 and 1. Those values represent the probability that there is an obstacle on that cell. All the values are initialized to 0.5 which is the average value between 0 and 1 since we do not know if there is an obstacle or not at that place.

To update this grid we created a class 'Cartographer' that uses the echoes of the lasers. For each laser echoe we compute the distance between the robot cell and the cell hit by the laser in the grid. Then we use the 'Bresenham' algorithm[1] to update all the cells in between the two above. For all those cells we compute an increment that is added or subtracted to them using the following procedure:

1. First we compute an increment value in regard of the value of the cell, the values for max and min increments are respectfully 0.15 and 0.015:
   - If the cell is a hit cell, meaning that it is the cell where the laser echoed back:

     $inc\_iro\_certainty = min\_increment$ `if` $is\_empty(cell)$ `else` $max\_increment$
   - If the cell is not a hit cell:

     $inc\_iro\_certainty = min\_increment$ `if` $is\_obstacle(cell)$ `else` $max\_increment$
2. Then we compute an increment factor in regard of the distance between the robot and the cell to update:

$$inc\_factor\_iro\_dist = 1 - abs(\frac{distance}{max\_lasers\_distance})$$

3. The final increment is computed by multiplying the factor with the defined increment:

$$final\_increment = inc\_iro\_certainty \cdot inc\_factor\_iro\_dist$$

4. The final increment is added to the cell if the cell corresponds to the cell hit by the laser and that the distance of the echoe is below the maximum laser distance. Otherwise it is subtracted.

With this method when a cell is qualified as obstacle it will be 10 times harder to decrement its value than to increment it and reversely. That makes the map more consistent because even when the robot hits something which make it shake or turns very fast we will not have false values in our grid. The distance factor is useful because it will make the cells near the robot update more easily than the cells far away, since the further away the cell is from the robot the less precise the laser information is.

This is the factory map built moving the robot by hand using the cartographer described above.

Figure 1: Explored map by hand

The mapping module also contains the 'ShowMap' class which is used to display the built map.

Chapter 3

Planning

The planning module is divided in three different parts.

1. The first thing to do is to find a goal point, that goal point must be chosen in a way that will make the robot explore unknown parts of the environment.
2. Then we have to build a path for the robot to follow between the actual position of the robot and the target.
3. The final thing to do is to use a path tracking algorithm that will make the robot follow the built path.

3.1 Find a goal point

In order to find a goal point we have to detect an unexplored zone that we can access to, to do so we used an approach based on frontier detection. A frontier is a region on the border between an explored zone and an unexplored zone. Then the first thing to do in order to determine the next goal point is to detect the frontiers.

3.1.1 Detect the frontiers

At first we thought that a naive approach could be enough for this part, but we later noticed that it was not efficient enough.

3.1.1.1 First naive approach

To detect the frontiers we go through all the unexplored cells of the grid and if that cell has an explored empty cell in its Von Neumann neighbourhood we know it is part of a frontier. The Von Neumann neighbourhood is composed of the four adjacent cells around a cell. Once we went through the whole grid we

have a list of all the cells that are on a frontier, the next step is to divide them into several regions.

To divide the frontiers in regions we go through the previously built list, each time we put a cell in its region we delete it from the initial list. For each cell we go through its Moore neighbourhood (The entire 8 cells neighbourhood). If one of its neighbour is in the initial list we recursively call the same function.

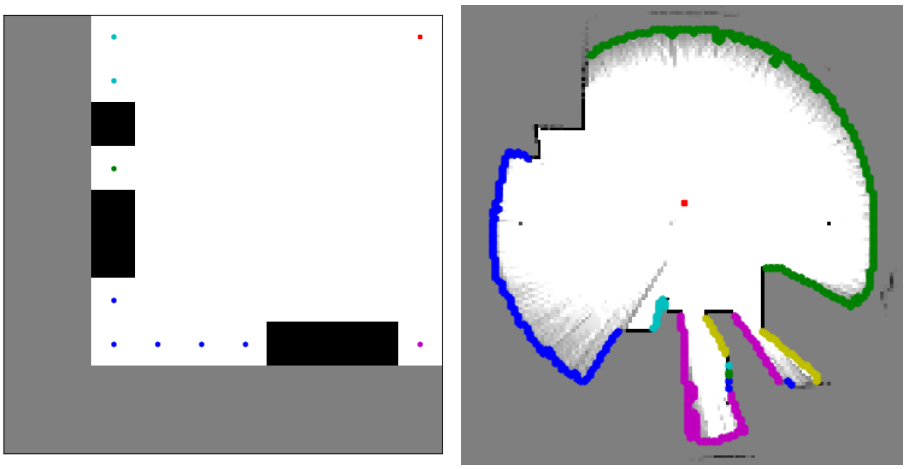
This is the pseudocode of the 'get_divided_frontiers' function:

Algorithm 1 get divided frontiers

```
1: procedure GET_FRONTIERS(map)
2:     frontiers is an empty array
3:     for cell in map do
4:         if is_unknown(cell) then
5:             for neighbour in von_neumann_neighbourhood(cell) do
6:                 if neighbour not in frontiers and is_empty(neighbour) then
7:                     frontiers.append(neighbour)
8:                 end if
9:             end for
10:        end if
11:    end for
12:    return frontiers
13: end procedure
14: procedure BUILD_FRONTIERS(frontiers, current_frontier, cell)
15:    neighbours ← moore_neighbourhood(cell)
16:    for neighbour in neighbours do
17:        if neighbour in frontiers then
18:            current_frontier.append(neighbour)
19:            frontiers.remove(neighbour)
20:            build_frontier(frontiers, current_frontier, cell)
21:        end if
22:    end for
23: end procedure
24: procedure GET_DIVIDED_FRONTIERS(map)
25:    frontiers ← get_frontiers
26:    divided_frontiers is an empty array
27:    while frontiers is not empty do
28:        current_frontier is an empty array
29:        cell ← frontiers.pop(0)
30:        current_frontier.append(cell)
31:        build_frontier(frontiers, current_frontier, cell)
32:        divided_frontiers.append(current_frontier)
33:    end while
34:    return divided_frontiers
35: end procedure
```

On the following figure we can see an example of the detected frontiers, the black pixels are obstacles, the white ones are empty cells, the red spot is the robot position and the other spots are the regions of frontiers. The map is

10 by 10 and there is a different color for each region. By testing this same function on a real map built by the robot in MRDS we noticed that despite being theoretically precise it is not efficient enough (it took around 5 to 10 seconds to get the frontiers). However, the function gave a very good result as we can see below:



(a) Frontiers test map

(b) Frontiers real map

3.1.1.2 Final approach, the Wavefront Frontier Detector algorithm

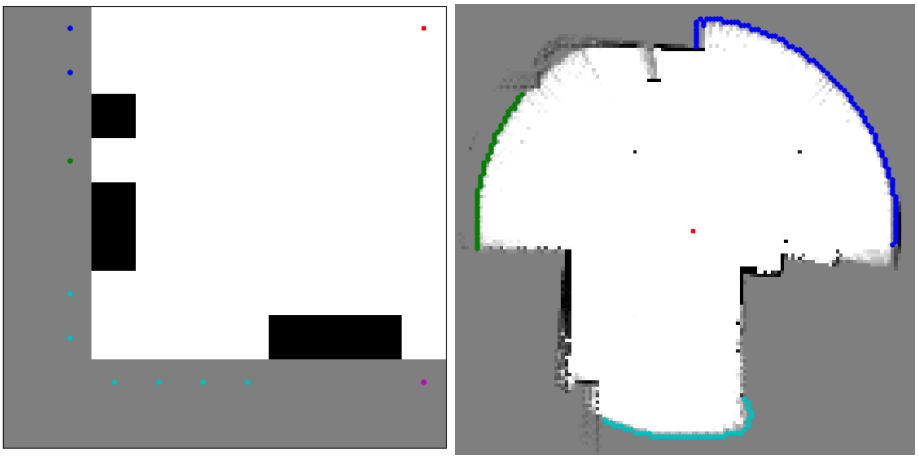
We found this algorithm in a scientific paper[2], happily there was a very comprehensible pseudocode in the paper that we simply followed to implement it in our project.

This is the pseudocode of the algorithm:

Algorithm 2 Wavefront Frontier Detector

```
queue_m ← []
queue_m.append(robot_cell)
frontiers ← []
map_open ← set([])
map_close ← set([])
frontier_open ← set([])
frontier_close ← set([])
map_open.add(robot_cell)
while queue_m is not empty do
    p ← queue_m.pop(0)
    if p in map_close then
        continue
    end if
    if is_frontier_point(p) then
        queue_f = []
        frontier ← set([])
        queue_f.append(p)
        frontier_open.add(p)
        while queue_f is not empty do
            q ← queue_f.pop(0)
            if q in map_close and q in frontier_close then
                continue
            end if
            if is_frontier_point(q) then
                frontier.add(q)
                for w in moore_neighbourhood(q) do
                    if w not in frontier_open and w not in map_close and w not in frontier_close then
                        queue_f.append(w)
                        frontier_open.add(w)
                    end if
                end for
            end if
            frontier_close.add(q)
        end while
        frontiers.append(frontier)
        for cell in frontier do
            map_close.add(cell)
        end for
    end if
    for v in moore_neighbourhood(p) do
        if v not in map_open and v not in map_close and has_open_neighbour(v) then
            queue_m.append(v)
            map_open.add(v)
        end if
    end for
    map_close.add(p)
end while
return frontiers
```

Those are the frontiers detected with this new algorithm in a test and a real case, in the real case we only displayed the frontiers with more than 20 points in it.



(a) Frontiers test map WFD

(b) Frontiers real map WFD

The main difference with our naive approach is of course the efficiency of this last method, it can almost instantaneously find the frontiers in the real map.

3.1.2 Choosing the goal point

Now that we are able to find the frontiers we have to first choose which frontier we want to go to and then what point in this frontier we should choose. To do so we determined that we should always try to explore the closest frontier, that way the robot will not have to cross through the entire map again and again.

To determine the goal point to go to we decided to choose a point that would be around the middle of the frontier, to do so we have to find the centroid of the selected frontier.

The coordinates of the centroid of a frontier is calculated in that way, with x_i and y_i the points of the frontier:

$$x = \frac{\sum_{i=0}^{n} x_i}{n} \qquad\qquad y = \frac{\sum_{i=0}^{n} y_i}{n}$$

The process of finding the frontiers and a goal point is done every 5 seconds, that way when the frontier moves the robot will follow it since it updates the point a lot.

3.2 Reaching the goal

3.2.1 First approach, the potential field

Initially we tried to use a potential field to make the robot go toward the goal while avoiding the obstacles around itself.

3.2.1.1 Computation of the force to apply

To compute the attractive force we use this method:

1. The first step is to compute the length of the vector, we compute the distance in the grid between the robot and the goal and we apply a weight of 0.5 and then limit the maximum length to 14.
2. Then we compute the angle of the vector, the formula is $atan2(\Delta y, \Delta x)$.
3. To finish we compute the coordinates of the vector using those formulas: $x : length * cos(angle)$, $y : length * sin(angle)$.

To compute the repulsive force we use the following method:

1. We consider a circle area around the robot with a radius of 7 cells.
2. In this circle we select the 6 closest obstacle, is considered an obstacle a cell with a value greater or equal to 0.75.
3. Then we compute the 6 corresponding vectors using the same method as for the attractive force. Except that we apply a logarithmic function to the computed length and then apply a weight of 3.2 to each of the computed vectors.
4. After that we have to add the 6 of them into one vector, to do so we simply sum all the x and y together to obtain the final repulsive vector.

To obtain the general force to apply to the robot we add the x and y of both attractive and repulsive forces together.

In the following figure we can see the forces applied on the robot. The blue hexagon is the goal point, the green arrow is the attractive force, the red one is the repulsive force and the one in magenta is the general force. Only half of the points of the frontiers are displayed out to save some time, this is why they are not very clear.

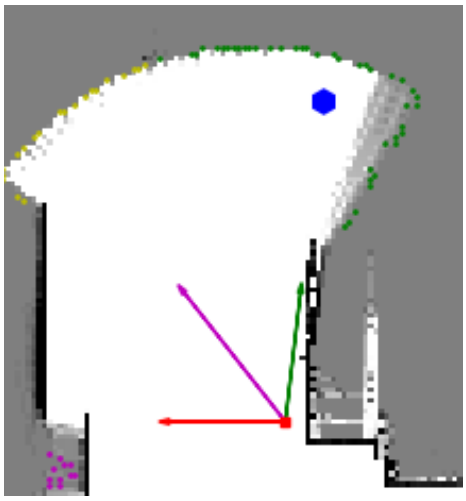


Figure 4: Forces applied to the robot

3.2.1.2 Convert the force into commands for the robot

Once the force to apply to the robot is computed we have to make the robot orientate itself in the direction of the vector and adapt its speed to the situation.

To do this we created a function in Controller called $apply\_force$, this function uses part of the algorithm we previously developed for the 'follow the path' assignment of the 'Fundamentals of artificial intelligence' course.

The coordinates x and y of the force vector to apply are given in parameters of the function.

- We first compute the length of the vector using:

$$\sqrt{x^2 + y^2}$$

- Then its angle using:

$$atan2(y, x)$$

- Then we compute Θ in that way:

$$sin(force\_angle - robot\_angle)$$

- With Θ we can compute the angular speed, with max_ang_speed being 3 and $weight$ being 0.8, in that way:

$$max\_ang\_speed \cdot \Theta \cdot weight$$

- The angular speed computed is then confined between $-max\_ang\_speed$ and max_ang_speed.

– Then we apply the following function to the angular speed to obtain an adapted linear speed:

$$max(0, 0.5 + log10(-ang\_speed + (max\_ang\_speed - 0.5)))$$

This worked pretty well in all open fields areas, but we had some problems of local minimums. Indeed when the goal was right behind an obstacle the robot could stay stuck behind the obstacle. Or when the robot was inside the 'garage' it was almost impossible for it to go out.

After seeing all those issues we decided to keep the potential field way for guiding the robot and that we were going to build a path to make the robot follow subgoals between its position and the goal.

3.2.2 Building a path

To build a path between the goal and the robot we decided to use a very classical algorithm called A*, we implemented the pseudocode from the wikipedia page[3].

We had almost nothing to add or change to adapt it to our case, we just made sure that the path would only include empty cells (value below 0.5).

On the figure bellow we can see the path between the goal and the robot, the big blue hexagon is the goal and the more little ones are the subgoals. The robot goes to the first subgoal of the path and when it is in reach of it the subgoal is deleted, then the first subgoal becomes the next one until it reaches the goal.

Figure 5: Path while exploring

With this method the robot can now explore anywhere, it can even go in the garage and then go out without much trouble.

Chapter 4

General improvements

At this point the robot was able to explore and discover its environment by itself, but there were some details that we could improve.

4.1 Inaccessible frontiers

Sometimes a frontier is found in an inaccessible place, then the robot will go to the closest frontier it can and the closest frontier will always remain an inaccessible one, the robot will then stay stuck.

To fix this issue we had to find a way to detect when the robot stays stuck and then delete the frontier that it is trying to reach.

We created a new process called 'frontiers_limiter' that would be executed in the back ground, it runs every second and memorises the actual position of the robot. It communicates with the goal planner which send to it the last closest frontier as soon as it is computed, the frontiers limiter sends to the goal planner a list of all the cells to ignore when it is building the frontiers.

It only memorise the last 20 positions of the robot, which make it delete every position older than 20 seconds. Once it has memorised 20 positions it computes Δx and Δy and if those Δs are bellow 3.5 meters for both of them the robot is considered as stuck, because it means that the robot stayed in a square of 3.5 meters by 3.5 meters for the last 20 seconds.

Also if the robot is detected as being immobile for 5 consecutive seconds the frontier is deleted.

If the robot is detected as being stuck the closest frontier will be deleted in this way: we go through all the points in the frontier and add all the cells in a radius of 6 cells around each point to the list of cells to ignore, then it sends the updated list to the goal planner. In that way the goal planner will ignore the cells in the list when it is building the frontiers in the future.

Chapter 5

Optimisation

To optimise the performances we choose to use multiprocessing.

5.1 Cartographer

We've put the cartographer in its own subprocess, it updates the map every 0.1 seconds. In that way the map is always efficiently updated.

The map is then 'sent' to the main program through a Queue (object of the multiprocessing python library). In that way the main program can recuperate the last version of the map at any time needed.

5.2 Show map

Since we usually print out many informations on our map: the frontiers, goal point and forces applied to the robot, the update of the graphical window takes some time. It came to a point where the update of the graphical window was the longest part in our program. We could not afford to lose that much time so we created a subprocess for the ShowMap.

The graphical window is updated every half a second.

5.3 Frontiers limiter

The frontiers limiter job is the one that detects when the robot is stuck, it needs to be executed once per second at any time. The simplest and most efficient solution was to put it in its own subprocess.

5.4 Communication with MRDS

To communicate with MRDS we were using the Robot class as an interface, the only thing that it did was send the request. Eventually we ran into a case where

more than one process tried to access to the robot position through the Robot object simultaneously. In that case we had an error message from the MRDS server and our program crashed.

To fix this situation we implemented a delay system in the Robot class. When we access to the Robot interface to request the position or the lasers if a request has already been made in the last 0.1 seconds it will return the result of that last request. By doing that we are able to use the Robot interface with as many processes as we want.

Chapter 6

Results

Our program can make the robot explore most of the map, the issues still present are the fact that when we compute a path we stop the robot to avoid it going into obstacles. That can be annoying because the robot will stop every 5 seconds since we compute the path every 5 seconds.

An other issue is the fact that we only consider the frontiers with more than 20 points in it, this becomes an issue when we want the robot to explore little areas. The robot then performs better when it has to explore a big part of the map.

An other issue is the way that we select the frontier, indeed we select the closest frontier from the robot, sometimes the closest frontier is behind a wall. When we are in that case an other issue appears, the fact that the A* algorithm will sometimes find a path that goes through a wall (When the wall is not yet fully discovered), leading the robot to stay stuck. A way of correcting that issue would have been to create a map on which the obstacles are expanded which would have make it easier to find a correct path.

Under we can see a map almost fully explored by the robot.

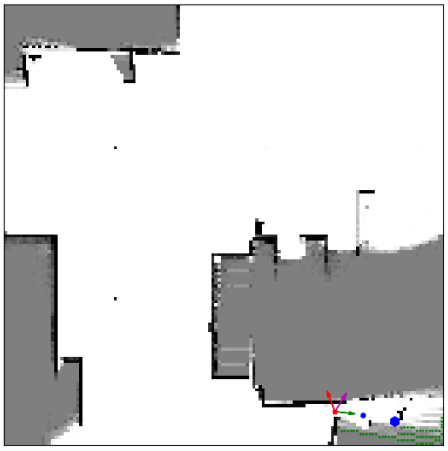


Figure 6: Map almost fully explored

Under we can see the robot while exploring a map.

Figure 7: Full screen

List of Algorithms

1 get divided frontiers 7
2 Wavefront Frontier Detector 9

Bibliography

[1] Bresenham line algorithm, 2018. http://www.roguebasin.com/index.php?title=Bresenham%27s_Line_Algorithm#Python.

[2] Abhishek Kathpal Anirudh Topiwala, Pranav Inani. *Frontier Based Exploration for Autonomous Robot*. PhD thesis, University of Maryland, 2019. https://arxiv.org/pdf/1806.03581.pdf.

[3] A* search algorihtm, 2019. https://en.wikipedia.org/wiki/A*_search_algorithm.

[4] colinday. Midpoint circle algorithm for filled circles, 2014. https://stackoverflow.com/questions/10878209/midpoint-circle-algorithm-for-filled-circles.