Umeå University
Department of Computing Science

Artificial intelligence
Methods and applications

5DV181

Map maker

Thomas Ranvier

supervised by
Ola Ringdahl
Juan Carlos Nieves Sanchez

January 10, 2019

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Contents

Chapter 1

How to use our program

For this assignment we used Python 3 as programming language, to run the program you can use the following syntax:

```
> python3 main.py
```

Our program is divided in two main modules, the mapping module and the planning 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 path that the robot must follow in order to explore the world.

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.

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 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 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:

   $$inc\_iro\_certainty = min\_increment \texttt{ if } is\_empty(cell) \texttt{ else } max\_increment$$

   – If the cell is not a hit cell:

   $$inc\_iro\_certainty = min\_increment \texttt{ if } is\_obstacle(cell) \texttt{ else } max\_increment$$

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.

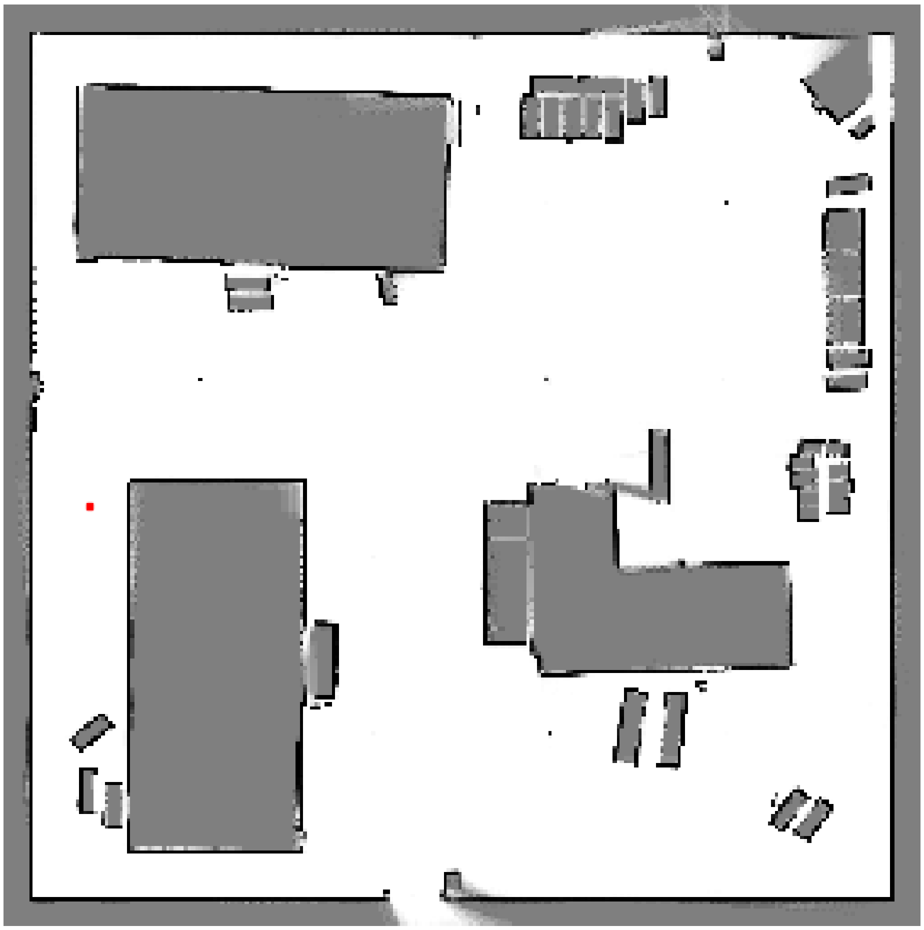

Figure 1: Explored map by hand

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

```
procedure GET_FRONTIERS(map)
    frontiers is an empty array
    for cell in map do
        if is_unknown(cell) then
            for neighbour in von_neumann_neighbourhood(cell) do
                if neighbour not in frontiers and is_empty(neighbour) then
                    frontiers.append(neighbour)
                end if
            end for
        end if
    end for
    return frontiers
end procedure
procedure BUILD_FRONTIERS(frontiers, current_frontier, cell)
    neighbours ← moore_neighbourhood(cell)
    for neighbour in neighbours do
        if neighbour in frontiers then
            current_frontier.append(neighbour)
            frontiers.remove(neighbour)
            build_frontier(frontiers, current_frontier, cell)
        end if
    end for
end procedure
procedure GET_DIVIDED_FRONTIERS(map)
    frontiers ← get_frontiers
    divided_frontiers is an empty array
    while frontiers is not empty do
        current_frontier is an empty array
        cell ← frontiers.pop(0)
        current_frontier.append(cell)
        build_frontier(frontiers, current_frontier, cell)
        divided_frontiers.append(current_frontier)
    end while
    return divided_frontiers
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.

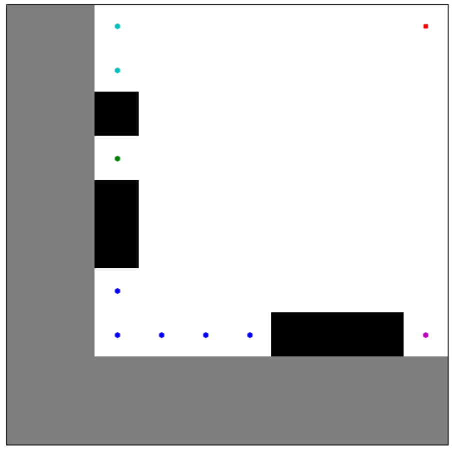

Figure 2: Frontiers test map

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:

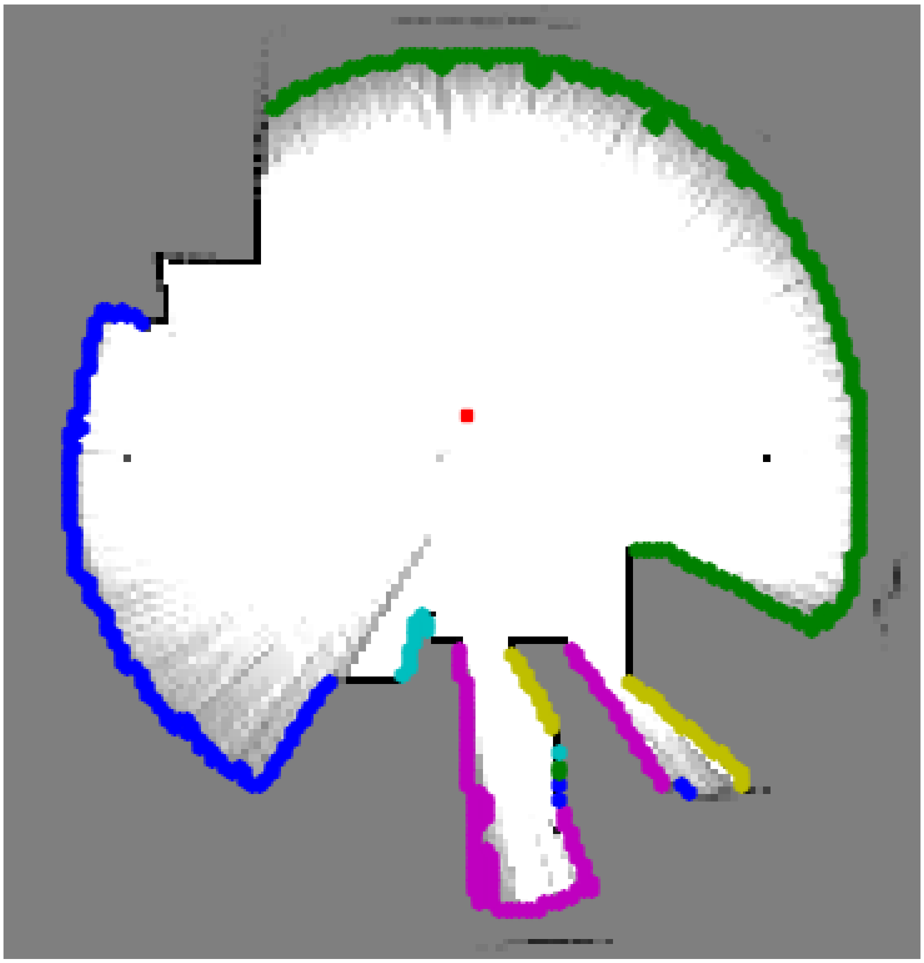

Figure 3: Frontiers real map

3.1.1.2 Final approach, using the Wavefront Frontier Detector algorithm

We found this algorithm in a scientific paper, 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.

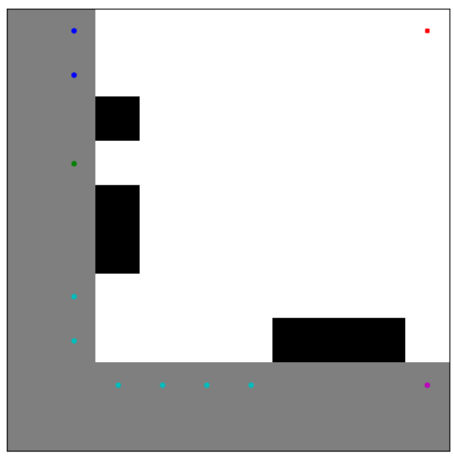

Figure 4: Frontiers test map WFD

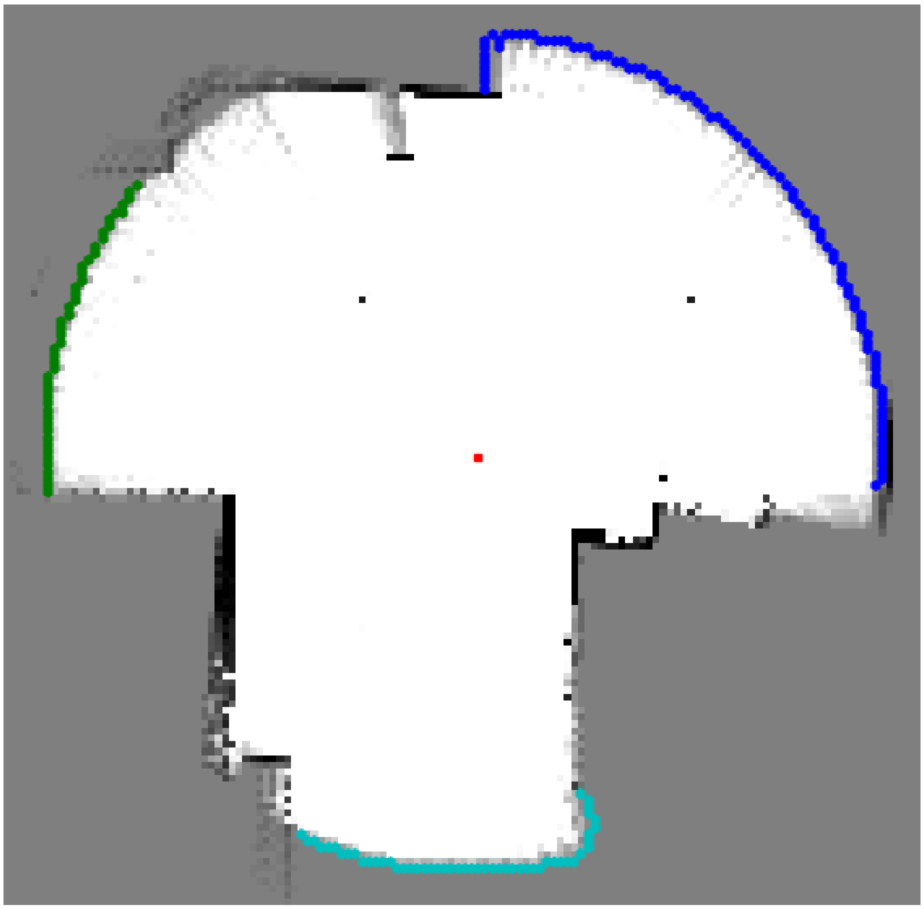

Figure 5: Frontiers real map WFD

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:

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

The process of finding the frontiers and a goal point is done every 10 seconds or every time the robot reaches the goal point, that way when the frontier moves the robot will follow it.

3.2 Reaching the goal

3.2.1 Computation of the force to apply

Now that we are able to define a goal point we still have to find a way to reach it while avoiding the obstacles.

At first we thought about building a path from the map and then implement an algorithm to avoid the obstacles while tracking the path. That solution would have worked well but we decided to use a potential field, in that way we have to compute the attractive and repulsive forces to apply to the robot to reach the goal while avoiding the obstacles.

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 0.4.

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 6.

2. In this circle we select the 5 closest obstacle, is considered an obstacle a cell with a value greater or equal to 0.75.

3. Then we compute the 5 corresponding vectors using the same method as for the attractive force.

4. After that we have to add the 5 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, in green is the attractive force, in red is the repulsive force and in magenta is the general force.

3.2.2 Convert the force into commands for the robot

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

Conclusion

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

List of Algorithms