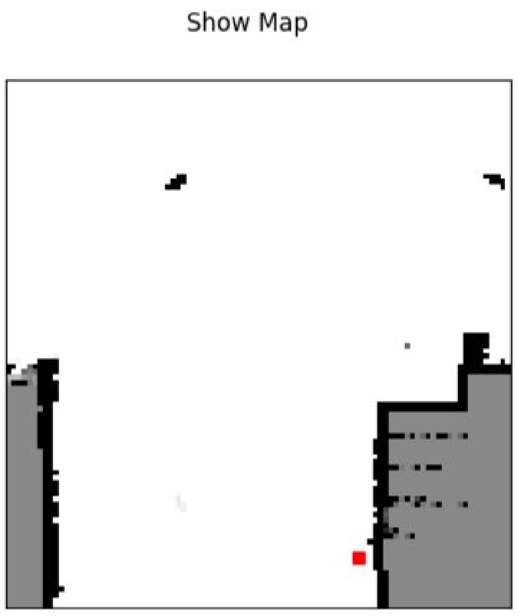
Artificial Intelligence - Methods and Applications

(5DV181)

Map Maker



Matilda Sandberg

tfy17msg@cs.umu.se

Jiangeng Sun

mrc20jsn@cs.umu.se

Table of contents

Introduction 3
Methods 3
Cartographer 3
Planner 4
Navigator 5
Reactive part 7
Testing and Running the Code 7
Delegation of work 8
Results and Analysis 8
References 9

Introduction

A lot of work is currently being put into the field of autonomous vehicles, where we recently have been seeing great progress. One example of that is Tesla's Full Self-Driving Autopilot. The software behind autonomous vehicles requires lots of different parts. In this report a robot control software is explained where the Kompai robot simulated in MRDS is controlled. The Lokarria http interface was used to access the robot's sensors and actuators. The code consists of both deliberative and reactive parts and all together controls the robot in such a way that it will move around in the environment while creating a map of its surroundings.

Methods

The most important parts of the code are described below in the different sections. The deliberative parts of the code consists of the cartographer (although this is also reactive), planner and navigator. The reactive part consists of a path-following-algorithm (pure pursuit) and obstacle avoidance. Before going into the different sections some overall important information about the method is necessary. For the map making a Cspace is applied, meaning we represent the environment to be explored as a grid. The chosen grid size was 0.4 m/grid for small areas (on side smaller than 20 meter) and 0.5 m/grid for large areas. The robot moved at a speed of 0.4 m/s. A laser scanner in the robot was used to sense the objects. The map is finished when there are no reachable unseen areas left.

Cartographer

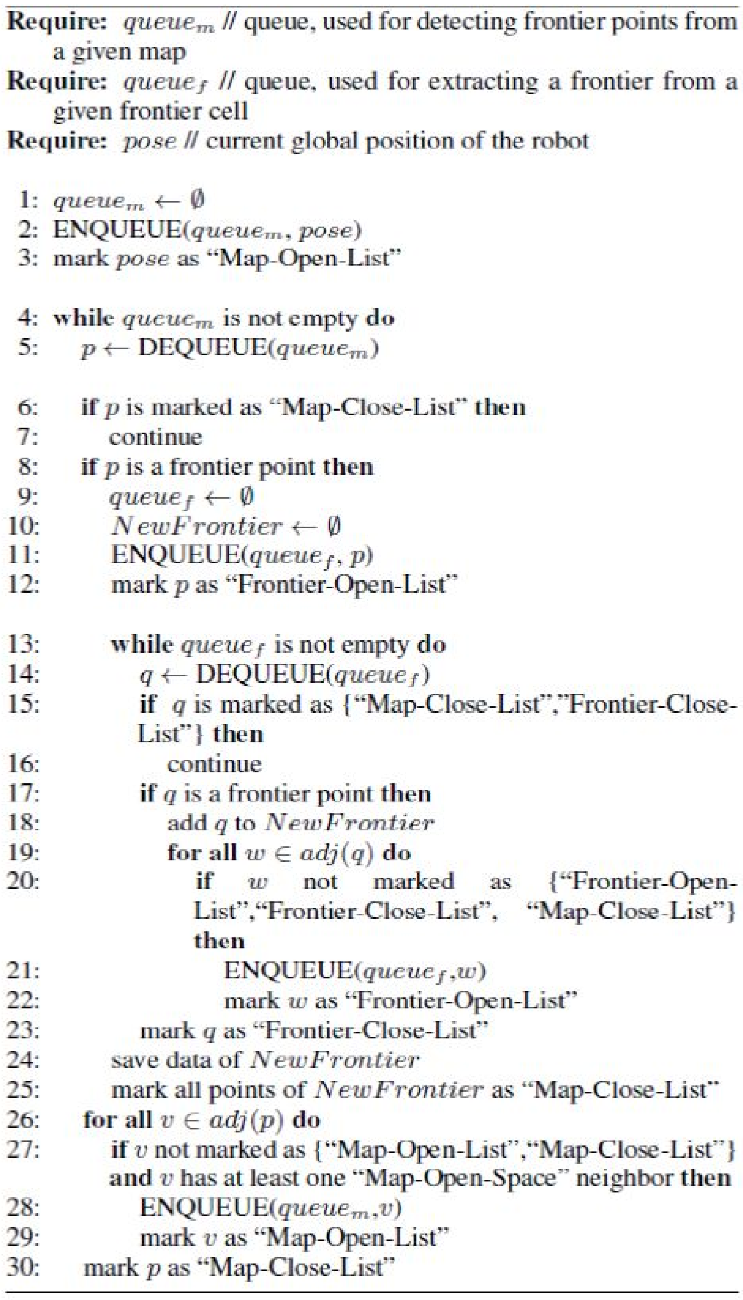
The cartographer is often called in the program because it updates the grid based on the laser input. It uses the HIMM method meaning that an obstacle is given an additional +3 and empty space (between robot and obstacle) is given -1. The maximum value of a grid is 15 and minimum 0. For the empty space Bresenham's line algorithm was used to get the grid coordinates between the robot and obstacle. Since the obstacle position was given in the robot's coordinate system it had to be changed into the world coordinate system before being transformed into grid coordinates (row and column). One important thing to notice is even though we only update grid points within the grid obstacles outside the grid are still used to notice empty space. We found that obstacles should be no further away from the robot than 39 meters to be considered while for the empty parts we considered a distance from the robot of 44 meters. This is because we found more errors of obstacles further away than 39 meters.

We also decided if a grid point had the value of 15 this value shouldn't be decreased. This because if the value is 15 it is fairly certain that an obstacle is there. Without this feature we found that some grids never settled, probably because some part of the grid was an obstacle and some other part empty or because of errors. Another thing we experimented with was a so-called "mask". In our case this means that if a gridpoint is surrounded by obstacles it's value will be changed to 15.

Planner

The planner is used to find the nearest frontier based on the WFD (Wavefront Frontier Detector) algorithm. The basic idea is based on four different lists:
1. Map-Open-List: points that have been enqueued by the outer BFS.
2. Map-Close-List: points that have been dequeued by the outer BFS.
3. Frontier-Open-List: points that have been enqueued by the inner BFS.
4. Frontier-Close-List: points that have been dequeued by the inner BFS.

Pseudo-code of the WFD algorithm**[1]** is shown in **Fig.1**.



```
Require: $queue_m$ // queue, used for detecting frontier points from
    a given map
Require: $queue_f$ // queue, used for extracting a frontier from a
    given frontier cell
Require: $pose$ // current global position of the robot

 1: $queue_m \leftarrow \emptyset$
 2: ENQUEUE($queue_m$, $pose$)
 3: mark $pose$ as "Map-Open-List"

 4: while $queue_m$ is not empty do
 5:   $p \leftarrow$ DEQUEUE($queue_m$)

 6:   if $p$ is marked as "Map-Close-List" then
 7:     continue
 8:   if $p$ is a frontier point then
 9:     $queue_f \leftarrow \emptyset$
10:     $NewFrontier \leftarrow \emptyset$
11:     ENQUEUE($queue_f$, $p$)
12:     mark $p$ as "Frontier-Open-List"

13:     while $queue_f$ is not empty do
14:       $q \leftarrow$ DEQUEUE($queue_f$)
15:       if $q$ is marked as {"Map-Close-List","Frontier-Close-
          List"} then
16:         continue
17:       if $q$ is a frontier point then
18:         add $q$ to $NewFrontier$
19:         for all $w \in adj(q)$ do
20:           if $w$ not marked as {"Frontier-Open-
              List","Frontier-Close-List", "Map-Close-List"}
              then
21:             ENQUEUE($queue_f$,$w$)
22:             mark $w$ as "Frontier-Open-List"
23:       mark $q$ as "Frontier-Close-List"
24:     save data of $NewFrontier$
25:     mark all points of $NewFrontier$ as "Map-Close-List"
26:   for all $v \in adj(p)$ do
27:     if $v$ not marked as {"Map-Open-List","Map-Close-List"}
        and $v$ has at least one "Map-Open-Space" neighbor then
28:       ENQUEUE($queue_m$,$v$)
29:       mark $v$ as "Map-Open-List"
30:   mark $p$ as "Map-Close-List"
```

Fig.1 *WFD algorithm*

For our code, we improved the algorithm. Because the principle of the algorithm is to search for frontier points from the current position of the robot to the neighbour grid, and then pass it layer by layer. The first frontier point it finds must be the frontier point closest to the current position. When the algorithm finds the first piece of frontier, we will break and end the loop. In this way, the algorithm no longer needs to traverse the entire map and this will save time.

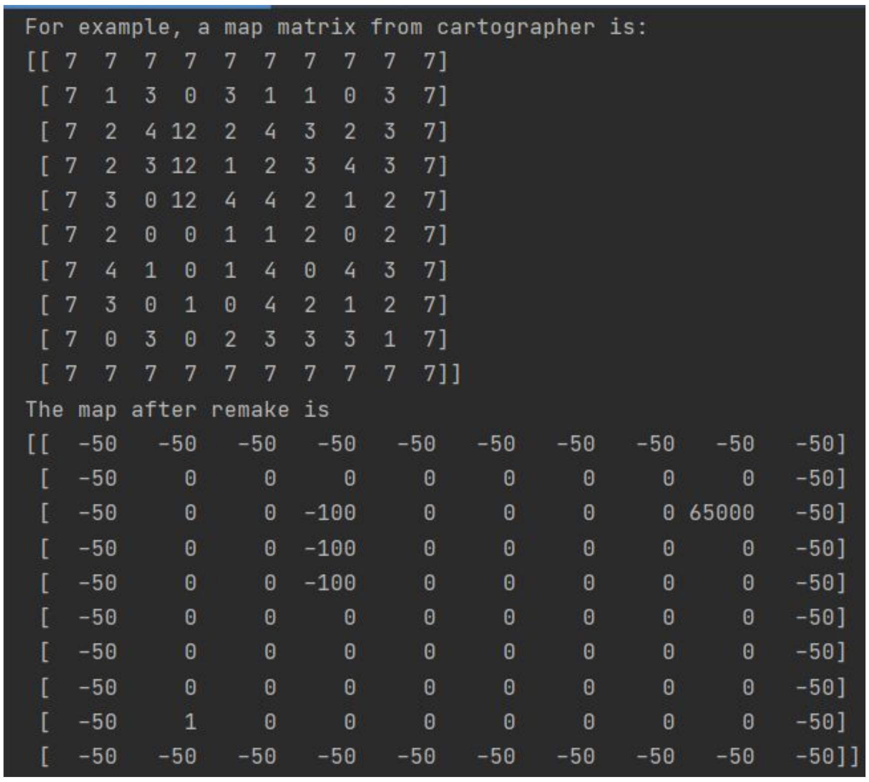
Then, in this piece of frontier, calculate the frontier point closest to the robot's position and mark it as the target point.

Navigator

The navigator is designed based on the wavefront algorithm. Just like the transmission of waves, starting from the current position of the robot (usually set to 1), the value of each adjacent grid increases by 1 until it meets the target point (usually set a very large number to target point, such as 65000). In this way, the robot only needs to go to an adjacent grid whose value is larger than the current grid's value. At the end, the robot will reach the goal point.

In the matrix created by the cartographer, the value of the unexplored grid is 7, the value of the unoccupied grid is 0 (or a number less than 7 and close to 0), and the occupied grid The grid is a number greater than 7. Such a map cannot use the wavefront algorithm.

The first step is to remake the map:
Let the value of the unexplored grid be -50, the occupied grid is -100, and the value of the unoccupied grid is still 0. At the same time, set the position of the robot to 1, and the value of the target grid to 65000, so that now the processed matrix can be used to navigate by the wavefront algorithm. A small example is shown in **Fig.2**.



```
For example, a map matrix from cartographer is:
[[ 7  7  7  7  7  7  7  7  7  7]
 [ 7  1  3  0  3  1  1  0  3  7]
 [ 7  2  4 12  2  4  3  2  3  7]
 [ 7  2  3 12  1  2  3  4  3  7]
 [ 7  3  0 12  4  4  2  1  2  7]
 [ 7  2  0  0  1  1  2  0  2  7]
 [ 7  4  1  0  1  4  0  4  3  7]
 [ 7  3  0  1  0  4  2  1  2  7]
 [ 7  0  3  0  2  3  3  3  1  7]
 [ 7  7  7  7  7  7  7  7  7  7]]
The map after remake is
[[  -50   -50   -50   -50   -50   -50   -50   -50   -50   -50]
 [  -50     0     0     0     0     0     0     0     0   -50]
 [  -50     0     0  -100     0     0     0     0 65000   -50]
 [  -50     0     0  -100     0     0     0     0     0   -50]
 [  -50     0     0  -100     0     0     0     0     0   -50]
 [  -50     0     0     0     0     0     0     0     0   -50]
 [  -50     0     0     0     0     0     0     0     0   -50]
 [  -50     0     0     0     0     0     0     0     0   -50]
 [  -50     1     0     0     0     0     0     0     0   -50]
 [  -50   -50   -50   -50   -50   -50   -50   -50   -50   -50]]
```

Fig.2 *Remake the map*

The second step is to use the wavefront algorithm. The pseudo code is as follows **[2]**:

```
check node A at [0][0]
now look north, south, east, west, south west, south east, south west,, north west, north east of this
node(boundary nodes)
if it is the point where robot is
        continue
if (boundary node is a wall or unknown node)
        ignore this node, go to next node B
else if (boundary node is robot location && has a number in it)
        path found!
else if (boundary node has a goal)
        mark node A with the number 3
else if (boundary node is marked with a number)
        find the boundary node with the smallest number
        mark node A with (smallest number + 1)
if (no path found)
        go to next node B at [0][1]
        (sort through entire matrix in order)
if (no path still found after full scan)
        go to node A at [0][0]
                (start over, but do not clear map)
        (sort through entire matrix in order)
        repeat until path found
if (no path still found && matrix is full)
        this means there is no solution
        clear entire matrix of obstacles and start over
        this accounts for moving objects! adaptivity!
```

There are a few things to note:

1. The value of the target point must be set to be large enough, otherwise, if the map is huge and the number of rows and columns is large, the navigator will get some errors.
2. The first row, last row, first column, and last column of the map are 'dangerous areas' when we use the wavefront algorithm, because they do not have four neighbor grids (up, down, left, and right). The solution is to insert an extra row in the first and last row of the map after redoing the map, and same for columns. When using the algorithm, it only needs to traverse from the grid in the second row and second column to the grid in the penultimate row and penultimate column.

A small demo of planner and navigator is shown in **Fig.3**.

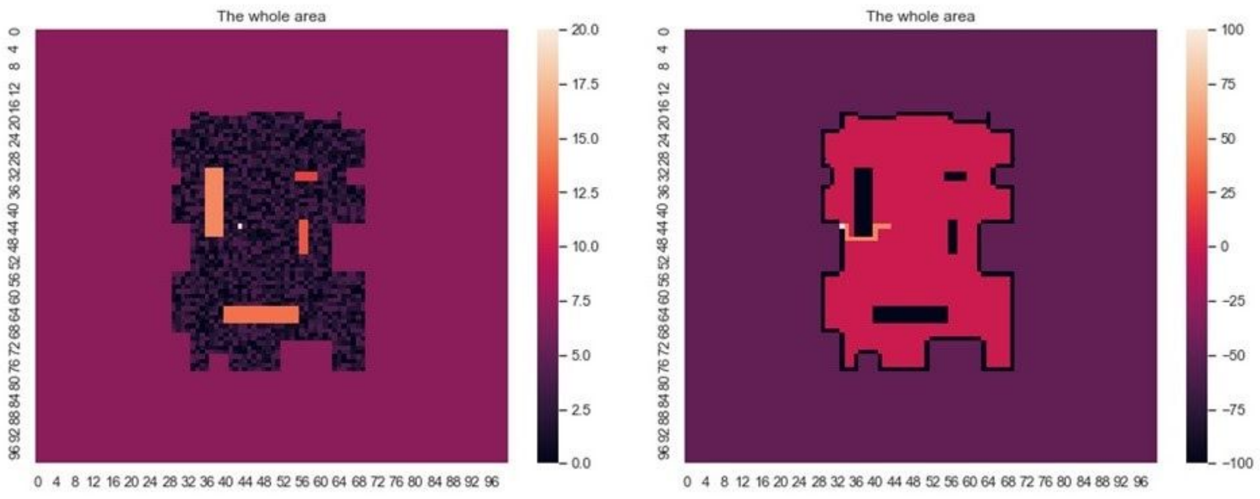


Fig.3 *Small demo of planner and navigator*

On the left is an example environment, the grids which value is equal to seven are unknown , the grids which value larger than 7 are occupied, the grids which value around 0 are unoccupied. The small light point which is equal to 20 is where the robot is.
The image on the right shows the path found by planner and navigator. The orange line which value equal to 50 is the path, the small light point which equal to 100 is the goal frontier point the planner found which is the nearest frontier point for the robot.

Reactive part

In order to follow the path given by the navigator firstly all points were changed into the robots coordinate system. Then the pure pursuit method was applied but without a specific lookahead point distance, instead all points on the path were lookahead points one after the other. We have a velocity of 0.4 m/s. While the robot was moving towards the lookahead point obstacle avoidance was applied. If an obstacle was within 0.3 meter from the robot in the direction it was heading the robot would stop, move backwards a bit and then get a new goal point. One option that was considered was to just find a new path towards the same goalpoint, but we decided it might be better to get a new goal point. This was partly because we found the obstacle avoidance in this case was used quite rarely, meaning the extra work of finding a new goal point was not significant. Also, if the goal point is no longer a frontier point it is not necessary to move towards that point anymore. However, if the robot were to act in an environment where obstacles commonly get in the way it would be better not to use the planner to find a new goal point. As part of the obstacle avoidance the robot will also back up a bit if it gets too close to the edge of the area (because the cartographer etc., only works inside the grid).

Testing and Running the Code

The code is accessed via a bash script *mapper.sh* which calles our python code *Mapper.py*. The bash script takes the following input arguments: First the url address corresponding to the computer on which MRDS is running on and what port is being used, then four numbers corresponding to xmin, ymin, xmax, ymax (in that order) which is the boundary of the area that the robot is going to map. The last input argument is a 1 if you want the program to output images of the map while running the code and 0 if you don't want this.

Delegation of work

We divided the work between each other in the following way. Jiangeng worked on the planner and navigator and Matilda worked on the cartographer, reactive part and putting all the code together in the end. We continuously had meetings where we helped each other and made sure both were up to speed with what was going on.

Results and Analysis

In **Fig.4** you can see some pictures of different runs on different areas.

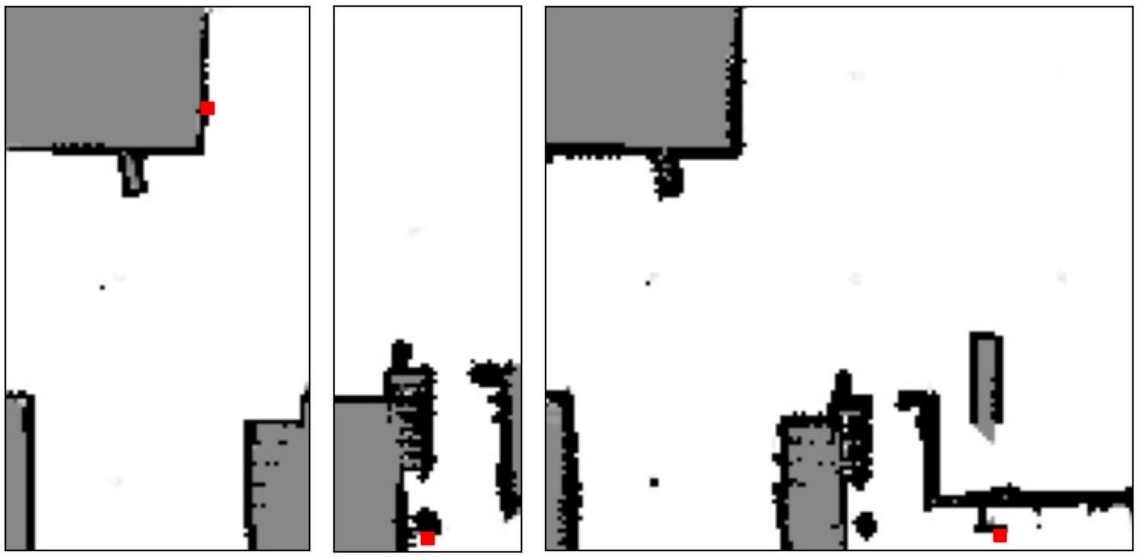


Fig.4 *The left image shows the left side of the factory environment in which we ran the simulation. The red square represents the robot. In the middle you can see the robot mapping inside a house. To the right you can see when the robot was mapping a bigger area.*

Since we're using HIMM with a mask the obstacles tend to get a little bigger than they really are on the map. If a more exact map is wanted another method should be used, for example the Bayesian approach. Also our robot is moving rather slowly in general. This is mainly because of problems occurring while the robot is mapping inside the house (in smaller areas) or when it is in the rightmost area (don't know why), however it should be possible to increase both the speed of the robot and the program as a whole.

Some problems we runned into was that the navigator sometimes couldn't find the goal point. Usually this was because we increased the obstacles before finding the path in the navigator (so the robot wouldn't try to move close to obstacles and run into them). To fix this we didn't increase obstacles close to the robot's current position or close to the goal point. We also added a limit so the navigator can't go on forever, instead it will send back two empty vectors if no path is found in time. If the outer loop gets this feedback it will get another goal point.

Another problem that we struggled with was that we received the message *connection refused* in the middle of mappings. This was because we sent and received signals via the http interface too often so by limiting the use of the interface this problem was solved.

We also found that sometimes the cartographer will mark some grids behind the wall as unoccupied cells even if it shouldn't be able to see through the wall. It means that the planner may believe that the incorrectly marked “unoccupied grids" are frontier points (because these points do meet the definition of frontier point). To solve this problem, we changed the value of unreachable grid points to 10 so they would no longer be considered frontier points and therefore not be chosen as goal points again.

To summarize our code works well for some areas, however for smaller areas (for example inside the building) we sometimes run into trouble.

References

1. Anirudh Topiwala,Pranav Inani,Abhishek Kathpal *Frontier Based Exploration for Autonomous Robot* (arXiv.org, 2018).
2. https://www.societyofrobots.com/programming_wavefront.shtml (retrieved on 2021/1/7).