Artificial Intelligence - Methods and Application

Assignment 2 - Map maker

5DV181 Tobias Nebaeus (CAS: tone0006, CS: c14tns)
Cyril Bos (CAS: cybo0001, CS: ens17cbs)

Teachers: Ola Ringdahl
Juan Carlos Nieves Sánchez

University of Umeå

Contents

1 Changes in version 2

An important difference from the first version is that **the location of the source code and program have been moved**, to `~c14tns/edu/5DV181/lab2/`. Almost all modules of the program have been improved since the first version:

- The laser model, the laser maximum distance has been corrected and the map update improved
- The goal planner, which frontiers are managed in a better way
- The path planner, which better avoids paths near obstacles
- Obstacle avoidance
- the whole communication between processes and main deliberation algorithm

The robot is now able to fully explore most areas of the factory environment, although some situations still present problems, especially if the robot is unable to detect obstacles due to the position of the laser scanner.

The report has been updated to reflect the changes in the program.

2 General information

The assignment has been developed in Python 3.5.3. The source code is in the directory The bash script `mapper` that respects the specifications is located at `~c14tns/edu/5DV181/lab2/`, and the source code under the `src/` subdirectory. The script can be launched using the following example syntax:

```
1 bash mapper nereid.cs.umu.se:50000 -50 -50 50 50
```

It does not work if `http://` is added in front of the url. It works when run on a linux machine. In order to synchronize the multiple processes, the robot will start moving several seconds after the script has been launched. When running the script on the same machine as MRDS, it can be launched using:

```
1 python3 mapper_main.py localhost:50000 -50 -50 50 50
```

The `mapper_main.py` script is located at `umu-ai-mapmaker/src`.

3 Architecture

The solution developed consists of several modules. The `mapper_main.py` uses them to control the main workflow. The `map` package corresponds to a cartographer module, containing the laser

model and the cspace grid. The `controller` package offers methods to send the GET and POST requests to MRDS and a reactive path tracking. The `goal_planner` package implements a frontier-based method to plan the goals of the robot, and `path_planner` implements an A-star based method of planning a path to these goals.

3.1 Module execution

In order to improve performances and allow true parallelism, some of the modules are run as separate processes. Blocking queues are used for communication between the processes. The processes that run concurrently are:

The cartographer. Runs in the main process (mapper_main.py), reading laser scans and pushing map and robot position updates to the other modules.

ShowMap (in `mapper/update_map_process.py`). The GUI of the program that shows a graphical representation of the map. This process updates the GUI for every change that is pushed by the cartographer, and also sleeps for 0.5s to prevent unnecessary CPU usage.

The planner and path tracking process (in `planner/planning_process.py`. Three modules run in this process: the goal planner, the path planner, and the path tracker. The process executes the following loop indefinitely (pseudocode):

```
goal = GoalPlanner.findGoal()
path = PathPlanner.findPath(goal)
while (not stuck or not finished):
    PathTracker.followPath(path)
```

4 Implementation

4.1 Cartographer (mapper package)

The cspace grid we used is of size $(x2 - x1) * (y2 - y1) * scale$. The scale value is how many cells there are in the cspace for each meter in the world. The value that worked the best for us was 2. The laser model implemented uses every other laser of the scan to update the map, as this resulted in less noise than using all available lasers.

We adapted the occupied probability formula of the generic sonar model presented in lecture 7. We considered the angle α is always equal to zero (the laser is a straight line). The maximum distance (R in the formula) value we used is 40, since this is the maximum distance the laser scan is able to operate.Bresenham's line algorithm is used to approximate a line to the coordinate of the laser hit. A cell is considered an obstacle if the probability value is superior to 0.7, and unexplored if it is exactly equal to 0.5. We are using only one laser out of two to optimize the map update and keep a good enough precision.

4.2 Navigator (controller package)

As previously stated, this package is used to send requests to the MRDS server to get the sensor information and control the movements of the robot.

The robot uses pure pursuit on the given path, skipping positions that are too close to each other. Reactive controls include some linear and angular speed corrections when obstacles are getting closer, and a complete stop when the robot is nearby enough an obstacle in front of him. It then proceeds to back off and rotates depending on the closest laser feedback. It will backpedal until the nearest obstacle is at least 1m away from the front of the robot (about 20 degrees from each side of the heading). This will be called "reactive stop" in the rest of the report.

Before starting the pure pursuit, it quickly rotates until it faces the first position of the path (until the angle difference between the heading and the angle between the two positions is minimal). This is useful when a goal has been reached. Indeed, the goals are often close to obstacles, so the robot often needs to do a u-turn. If regular pure pursuit was started right away, it would make the robot reactive stop trigger inefficiently.

4.3 Goal planning

The goal planning module is responsible for choosing the next area to explore. This is done using frontier-based exploration. A *frontier* is a set of open, adjacent squares in the CSpace grid that are also adjacent to unknown, unexplored squares. Choosing which frontier to explore can be done in several ways, such as using the frontier closest to the robot or the largest one. The implemented goal planner chooses the centroid of the closest frontier as the goal point.

The task of finding frontiers is not trivial and can be costly in terms of performance. The paper *Robot Exploration with Fast Frontier Detection: Theory and Experiments*[1] describes several algorithms that deal with finding frontiers. The algorithm labeled "Wavefront Frontier Detector" from this paper was used in the goal planner, and its pseudocode is presented below.

```
queuem = empty
ENQUEUE(queuem, pose)
mark pose as "Map-Open-List"

while queuem is not empty do
    p = DEQUEUE(queuem)

    if p is marked as "Map-Close-List" then
        continue
    if p is a frontier point then
        queuef = empty
        NewFrontier = empty
        ENQUEUE(queuef , p)
        mark p as "Frontier-Open-List"

        while queuef is not empty do
            q = DEQUEUE(queuef )
            if q is marked as {"Map-Close-List","Frontier-CloseList"} then
                continue
            if q is a frontier point then
```

```
            add q to NewF rontier
            for all w in adj(q) do
                if w not marked as {"Frontier-OpenList","Frontier-Close-List",
                        "Map-Close-List"} then
                    ENQUEUE(queuef ,w)
                    mark w as "Frontier-Open-List"
        mark q as "Frontier-Close-List"
    save data of NewFrontier
    mark all points of NewFrontier as "Map-Close-List"
for all v in adj(p) do
    if v not marked as {"Map-Open-List","Map-Close-List"}
        and v has at least one "Map-Open-Space" neighbor then
        ENQUEUE(queuem,v)
        mark v as "Map-Open-List"
mark p as "Map-Close-List"
```

The algorithm consists of two breadth-first searches: one for finding all open space points reachable by the robot, and one to find all frontier points adjacent to another frontier square. A number of lists are used in order keep track of the points that have been processed already. The resulting frontiers are added to a list, which is later sorted using the distance from the robot to the centroid of each frontier. The closest frontier is then chosen for exploration and returned by the goal planner. If this frontier has already been visited or banned, this frontier is removed from the list and the second-best frontier is chosen. This is repeated until the new frontier is suitable.

4.4 Path planning

The path planner uses the A* search presented during the course. We had trouble implementing it at first trying to transform the cspace grid into a graph represented by a dictionary, where each key is the tuple (row, column) and the associated value the list of neighbours. We instead adapted the implementation found at http://code.activestate.com/recipes/578919-python-a-pathfinding-with-binary-heap/. It uses the grid directly, and keeps the opened nodes in a binary heap to sort them automatically.

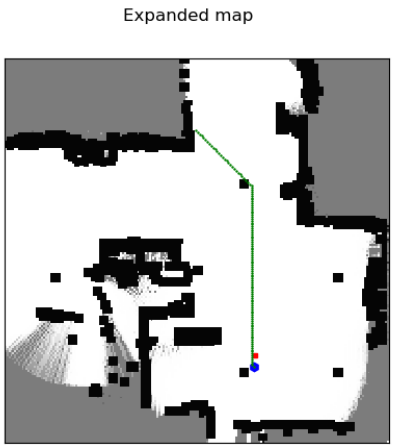
The goal planner often gives a goal that is cornered with unknown cells. To prevent the path planner from creating a path in obstacles that are yet to be discovered, we stop the search at an arbitrary depth and use the last opened node as a sub-goal. This depth limit is fixed depending on the size of the map, with trial and error we came up with the formula $8 * scale * (width + height)$.

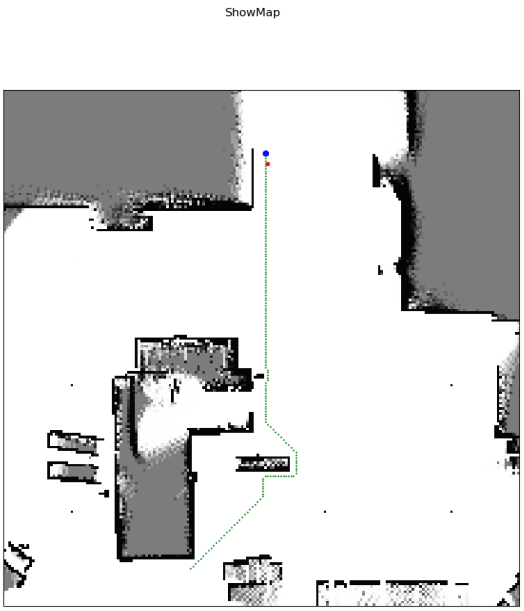
Contrary to Wavefront, A* tends to plan paths that are too close to obstacles. To fix this issue, the robot uses a copy of the occupancy map which obstacles are extended by two cell in every direction. Since each cell is 0.5 meters long, and the robot is about 0.9m long, most paths will need cells expanded at least twice. But the robot reactive stop might interfere with this too, so additional expansions are sometimes required. If the pure pursuit is stopped while aiming for a sub goal, the real goal will not be banned since it's not sure yet it is a blocking goal. The goal is however added to another list, representing already planned goals. And every time the same goal is found by the goal planner, it triggers an additional level of obstacle extension up to 5. Above this limit, the robot tries to unblock itself from the obstacle (in a similar way as the reactive stop), the goal is banned and the planning process starts again. This is implemented in `planner/planning_process.py`.

The program will stop if the frontier algorithm can't find any frontier. The main process waits for the planning process to stop and then terminates the GUI process (SIGTERM signal). Also, in the script
`mapper/update_map_process.py`, two lines can be uncommented to visualize the map with expanded obstacles used by both planning algorithms. This visualization doesn't adapt itself to the additional obstacle extensions when treating a blocking goal, it shows the first level of expansion that is always used.

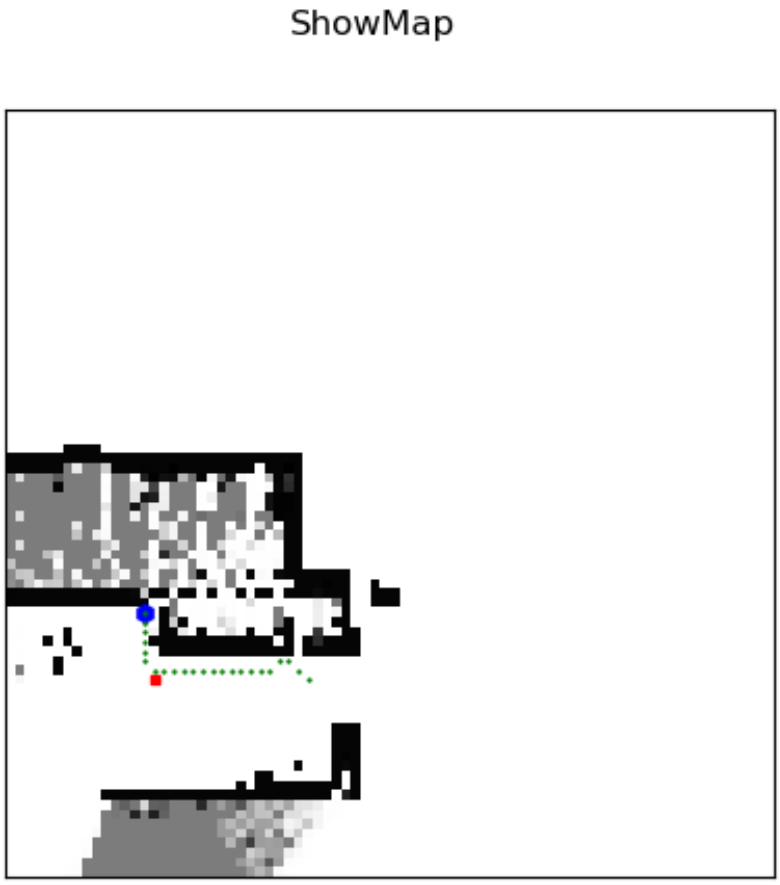
5 Results

Here is an example result of the map of the factory of running the map maker on the area from $(-50, -50)$ to $(50, 50)$. The blue point represents the current goal and the green line the current planned path. It took about 7 minutes for the robot to explore the area. The first figure is the expanded obstacle map used by the planning module. The robot had started on the initial position of the MRDS Factory environment. The second figure is the actual occupancy map, same area but started from the bottom right of the map (where the robot had stopped after computing the shown expanded map).





On smaller areas, the robot still performs well, here is an example for area $(-20, -20)$ to $(20, 20)$. It took 2 min for the robot to explore that area.



A video hosted at https://youtu.be/6mEpk-EFsFs shows the execution that leads to these results (at x4 speed). It also quickly shows how it behaves in the room environment, with some adjustments to parameters (frontiers parameters, speed, obstacle detection distance, ...).

On the performance side, the process that takes the most CPU usage is the real-time map, around 15% with a sleep of 0.5s. Without this sleep, it took up to 30%, which is the limit for one process on a 4-core CPU (100/4 = 25%). The two other processes usually do no take much more than 10-15%, unless the goal is really far away. These figures come from an i5 4690k CPU, 4 cores clocked at 4.17 GHz. Results may vary depending on the processing power.

6 Encountered problems

We encountered several problems during the development. A first problem was to correct the laser model and the calculation of the angles. We discovered there is an offset to apply given in the laser properties.

The solution to the goals being outside of the known space sometimes makes the path planner plan an incorrect path, if the goal is blocked by an obstacle not detected on the map. It also tends to create paths that are too close to obstacles. Wavefront algorithm would probably be better suited. We still managed to adapt A* successfully. A problem that still resides is that extending the obstacle always at least once when planning the path might bug the robot in narrow spaces, making it think it's inside an obstacle. So it is obliged to try with the occupancy map as is, which will often lead the robot to get blocked, triggering the reactive stop and replanning. This makes the robot lose a lot of time but is necessary to cover as many cases as possible. To fix this issue, we use the first nearest empty cell to search for a goal and a path. We heard from another group that they ran into problems with the Wavefront algorithm, so a mix of both approaches would probably be a better solution, i.e Wavefront with stopping at subgoals in the general direction of the originally planned goal.

Another problem was to synchronize the different processes. In the beginning, the path and goal planning were made in different processes. This created problems where the robot started a new path planned from a different position than its current one. The path planner was starting a new search from a previous position while the robot was moving, resulting in the robot trying to go back to its previous position to take the new path. Putting both planning inside the same process as long as the pure pursuit made these synchronization problems much easier to

Some obstacles were difficult to detect or not detected at all, such as the ramp at the top right, near real word coordinates $X : 10, Y : 55$. The reactive controls still detect it and prevent the robot from colliding. This might be due to the Bayesian updating that considered from first scans that there was no obstacle there, and then subsequent updates do not update it correctly since they are based on that almost equal to zero probability value. After the robot approaches these specific areas a couple time, the map finally shows them correctly. We solved this problem my modifying the laser model to add a minimum increase of probability value on the precise cell hit by a laser, if this cell isn't considered an obstacle yet.

During previous tests, it sometimes crashed into some obstacles that seem to have buggy hitboxes (for instance one of the trucks at the bottom left), which make us think some obstacles on the map are bugged or maybe exploit the laser specular reflection problem. The way it is implemented, the robot should always stop quickly enough. However, it might backpedal and crash into a wall, we wanted to make the reactive stop manage that issue but since the robot does not really have lasers at the back it was difficult to.

7 Division of the work

As we worked in pair, we assigned to each other different parts of the work. One of us worked more specifically on the goal planning and reactive controls and the other on the laser model and path planning. We worked together on the path tracking cspace grid representation and of course exchanged ideas on each other's parts. For the second submit, we tried to exchange our parts and really both understand how they work so that we come up with better ideas where we had problems.

References

[1] Gal A. Kaminka Matan Keidar. *Robot Exploration with Fast Frontier Detection: Theory and Experiments*. URL: http://www.ifaamas.org/Proceedings/aamas2012/papers/3A_3.pdf (visited on 01/04/2018).