Artificial Intelligence in Mobile Robots

Final challenge

By *Tobias L* & *Özgun*

Professor *Alessandro Saffiotti*

Lab Assistant *Ali Abdul Khaliq*

Örebro University

October 28, 2016

Contents

1 Overview 3

I Implementing Fuzzy Rule-based Control for Obstacle Avoidance 4
2 Task 5
3 Conclusions / Tests 5

II Updating Map with Obstacles 7
4 Task 8
5 Conclusions / Tests 9

III Implementation of Custom Made Fuzzy Rules 10
6 Task 11
7 Conclusions / Tests 11

IV Breadth-First Search vs. A* 13
8 Task 14
9 Conclusions / Tests 14
9.1 Path comparison 14
9.2 Time Comparison 19

V Distance Map Generation & Gradient Following 21
10 Task 22
11 Conclusions / Tests 22

VI Discussion 24

Student information

Tobias Lindvall
870603-6657
tobiaslindwall@gmail.com

Özgun Mirtchev
920321-2379
ozgun.mirtchev@gmail.com

Report handed in: 2016-10-28

1 Overview

In the final challenge the main task is to write a robot navigation program, using every system that was completed in the previous labs, into a single system to be able to control the ePuck using the Hybrid architecture, a combination of the Sense-Plan (planning) and Reactive (fuzzy rules) architectures.

Given tasks are to make the ePuck move to a target position, implement an obstacle detection procedure which updates the map accordingly and re-plan if the ePuck was following a path. Other tasks are to compare path-finding algorithms and implement a different path-generation and path-following procedure as called as "distance map generation & gradient following".

In each part of this document every task that was completed will be presented with problems and possible solutions. After each implementation, a testing part which shows how the systems were tested and possible conclusions from the tests.

Further discussions will take place in the discussion part.

Part I

Implementing Fuzzy Rule-based Control for Obstacle Avoidance

2 Task

The purpose of this task is to make the ePuck able to re-actively reach a specific goal while avoiding obstacles in an unknown environment.

More specifically what is needed to be done is a merge of two functions from the previous labs. One function is called Goto() which makes the ePuck move to a specified target. The second function is called AvoidObstacles() which enables the ePuck to be able to avoid obstacles through the use of its IR-sensors and the use of predicate rules.

There were a couple of things that had to be kept in mind while solving this problem:

- Threshold values for detecting obstacles with IR-sensors had to be set and tuned.
- Movement and rotation speed to be used.
- Monolithic or modular approach

Implementing the fuzzy rule-based control was done with help from pseudo code that was given in lecture slides. The control takes the coordinates of where the ePuck should end up in the environment as input.

To control the ePuck in a fuzzy way and make it head towards the goal, fuzzy predicates were used. These predicates represents the relationship between the ePuck's current position and the goal.

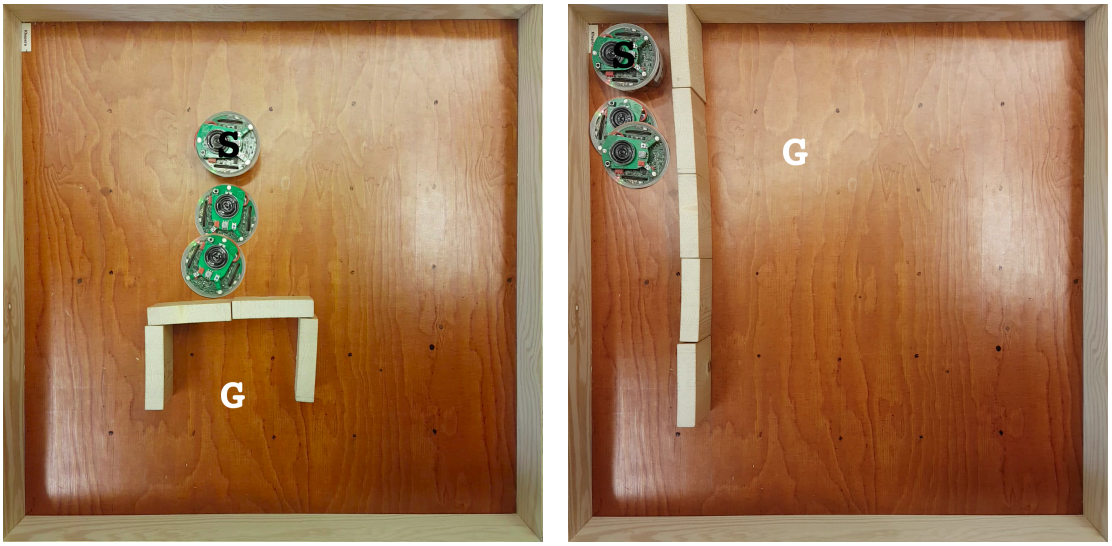
To make the ePuck avoid obstacles on it's way to the goal, another set of fuzzy predicates was used. These predicates are calculated from output values of the IR-sensors on the ePuck. When calculated, they describe if there are any obstacles present, in what direction and how far away these possible obstacles are.

When these predicates are calculated, they are the basis for setting the rotation and movement speed of the ePuck. The IR tuning could depend on the precision and eventual noise in the IR-sensors.

The merging of the two functions was done by using the Monolithic approach.

3 Conclusions / Tests

When testing with the provided rules from the lectures, the ePuck was unable to find its goal in most situations since there simply weren't enough rules for all possible states, so there would be a lot of empty fuzzy sets, causing the speed of the wheels to not be set and the ePuck to get stuck. Figure 3.1 illustrates this pretty well.



(a) Gets stuck at wall (video)

(b) Fails at moving out of the obstacle setup (video)

Figure 3.1: Basic Fuzzy Avoid Rules

Of course the path planning and the tracking functions were implemented since before and can be found in the appendix:

- Follow Path
- Track

Part II

Updating Map with Obstacles

4 Task

The purpose of updating the map with obstacles is to get a correct model of the environment which makes upcoming searches and planning better.

Problems that had to be solved to be able to update the map with new obstacles:

- Which on-board IR sensors should be used?
- How far from the ePuck is the detected obstacle?
- In which grid node is the detected obstacle?
- How to see the difference between a new obstacle and one that is already in the map?

To make the ePuck able to detect obstacles, four of the on-board IR sensors values (one left, one right and two forward) are read. When read, they are verified to see if any object is nearby. If there is, the distance to the obstacle is calculated which is used to see in which map cell the object is. If the cell already contains an obstacle or a wall piece, no action is taken. Otherwise if the cell is empty, the cell is updated with information about containing the obstacle. If this updated cell happens to be in the planned route for the ePuck, the current tracking will abort and a new plan will be made.

To make the distance to the obstacle from the ePuck be more precise, each IR-sensor's value was measured in length. Doing this, enough data was provided to be able to generate a graph (Figure 4.1) in MATLAB, and a formula to use. This formula would be used to convert the values of the IR-sensors into millimeters in the program, making the calculations even more accurate. The formula is as follows: $8.7 * 10^{-7} * IRvalue^2 - 0.0087 * IRvalue + 23 * F$ where F is a factor that was added to try and compensate for the differentiating values of the IR-sensors. In example, if the value of IR-sensor1 is at 500 with the distance to the closest obstacle right in front of it at 10 mm, the value of the IR-sensor2 might be 400 instead at the same distance. This formula will try to make up for that difference so that each IR-sensor will will output the correct value at the same distance. Making it slightly more sensitive and accurate to respond to its surroundings.

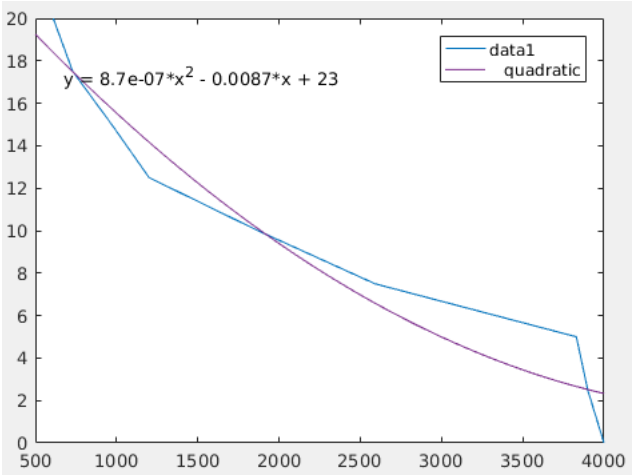


Figure 4.1: Formula for distance calculation

5 Conclusions / Tests

(a) Moving to goal

```
@@@ @@@ @@@ @@@ @@@ @@@ @@@ @@@
@@@                         @@@
@@@       4   S     4       @@@
@@@   4   3   *     3   4   @@@
@@@   3   2   *     2   3   4 @@@
@@@   2   1   G     1   2   3 @@@
@@@   3   2   1     2   3   4 @@@
@@@ @@@ @@@ @@@ @@@ @@@ @@@ @@@
          -> Replan ->
@@@ @@@ @@@ @@@ @@@ @@@ @@@ @@@
@@@                         @@@
@@@   5   4         4       @@@
@@@   4   *     S   3   4   @@@
@@@   3   *   ###   2   3   4 @@@
@@@   2   *     G   1   2   3 @@@
@@@   3   2     1   2   3   4 @@@
@@@ @@@ @@@ @@@ @@@ @@@ @@@ @@@
```

(c) Map updated with obstacle

(d) Moving back to start point

```
@@@ @@@ @@@ @@@ @@@ @@@ @@@ @@@
@@@   3   2   1   2   3   4 @@@
@@@   2   *   G   1   2   3 @@@
@@@   3   *     1   2   3   4 @@@
@@@   4   *   ###   3   4   5 @@@
@@@   5   *   S     4   5     @@@
@@@   6   5         5       @@@
@@@ @@@ @@@ @@@ @@@ @@@ @@@ @@@
```

(e) Re-plan with updated map

Figure 5.1: Updating map with an obstacle (video)

Code to be found in appendix: Detect Obstacle Plan and Replan

Part III

Implementation of Custom Made Fuzzy Rules

6 Task

Implementing new custom made fuzzy rules can improve the behaviour of the ePuck and make it more cautious to nearby obstacles. There are a lot of things to keep in mind when developing more rules.

Problems that was encountered and had to be solved:

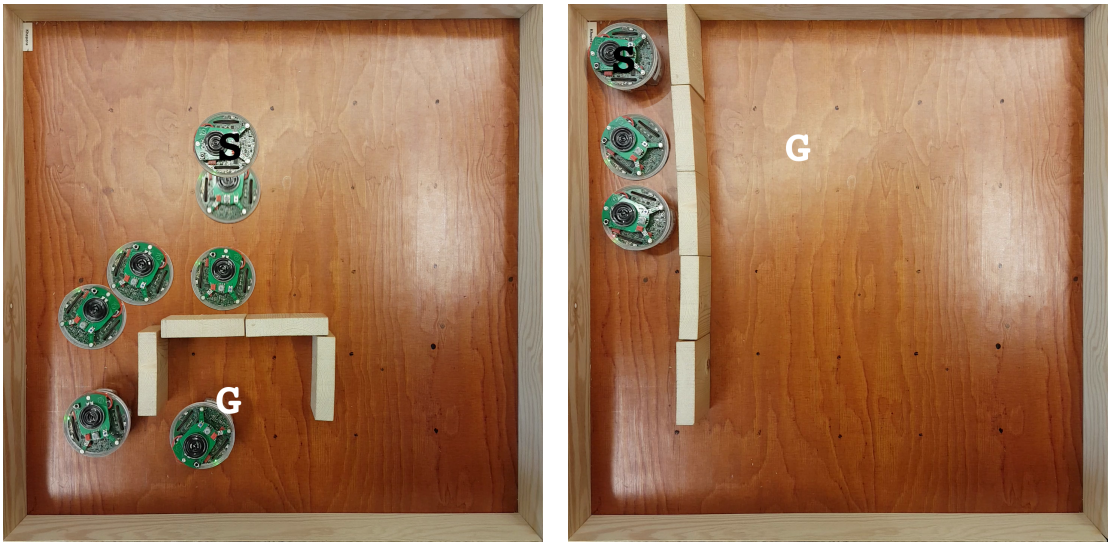
- Find and implement rules for certain situations where the ePuck would get stuck.
- Make up new rules by modifying/combining already existing ones?
- Make up new rules by writing from the scratch?
- Different threshold values than before?
- Making sure to not make contradicting rules

The custom made fuzzy rules was implemented with the monolithic approach. The fuzzy rules that was given as an example in the lecture slides was implemented at first. After testing scenarios with different setups of obstacles on the map, new rules were formed and the old were modified.

7 Conclusions / Tests

When implementing the rules we thought about which situation a rule would apply to. It was very difficult to see when and where a rule would do as we expected and to make sure it wouldn't conflict with the other rules. A lot of time was spent to try and find the best rules by testing the ePuck extensively with different obstacle setups. Some setups may perhaps have been a bit extreme since we like it to be as precise as possible when moving around. This might've been a bit overkill.

As seen from the figures below, there are situations the custom made rules will be able to handle, but also some it won't, depending on where the goal is relative to its position and possible obstacles in-between.



(a) Moves to goal successfully (video)

(b) Fails at moving out of the obstacle setup (video)

Figure 7.1: Custom Fuzzy Avoid Rules

In figure 7 one can see that in the first map the ePuck successfully managed to get to its target by avoiding the obstacles. However in the second map the ePuck could not find an opening that enabled it to get through without failing the rules. Since the goal was on the other side of the obstacle-wall, it couldn't go too far without following the rules, which told it to face the goal position as much as it could, while still avoiding the nearby obstacles causing it to go up and down between the top cells. Had there been an opening in the corner from where it started, it would have been able to get to the goal point without any problems. The provided video-links will give a better visualization of how the ePuck behaves in each situation.

Code in appendix: Custom Fuzzy Rules

Part IV

Breadth-First Search vs. A*

8 Task

Breadth-first search and A* are two well-known path-finding algorithms that always finds the best solution. In general, Breadth-First Search (BFS) takes more time than A*. The higher number of nodes that exists in the map, the bigger time difference between the algorithms. This has to be taken into account to be able to state the advantages and the drawbacks of the algorithms.

A* generates a path that goes in a zig-zag pattern while Breadth-First search generates a path with a minimum number of turns. In the enforcement with the ePuck, the Breadth-First search could be better. This is because every time the ePuck rotates, some precision in the positioning is lost causing it to be misplaced in the environment even though it still thinks its on the correct path.

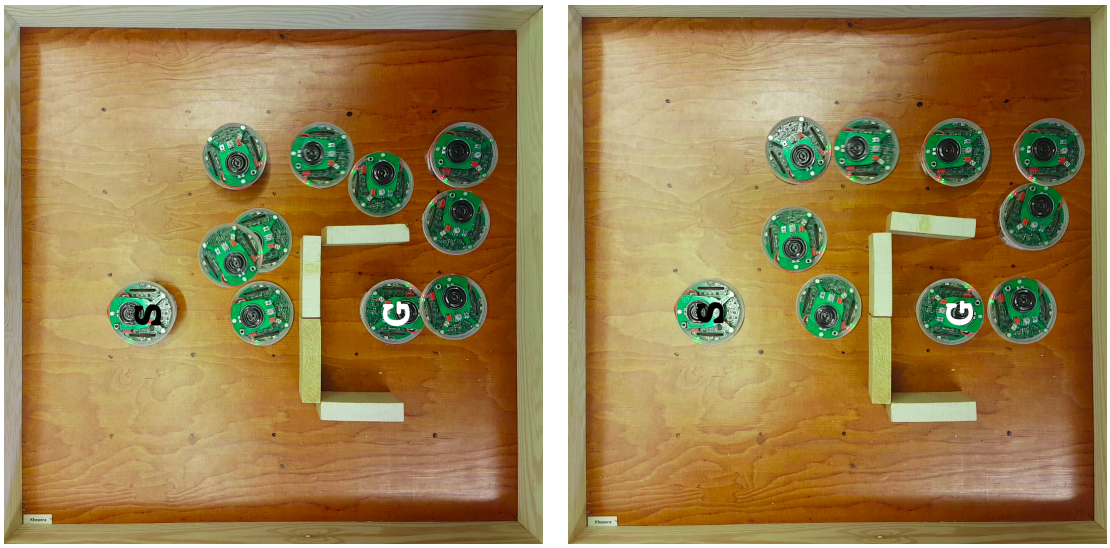
9 Conclusions / Tests

In these tests we're using our own fuzzy rules and the implemented obstacle detection procedure.

9.1 Path comparison

Map 1

Here the path of the ePuck is compared between the algorithms, to see which of them takes a better path and how long it takes to get to the goal. The measured time to get to the goal should not be considered as an exact number, since there are many factors involved in how the ePuck will behave in different lighting.

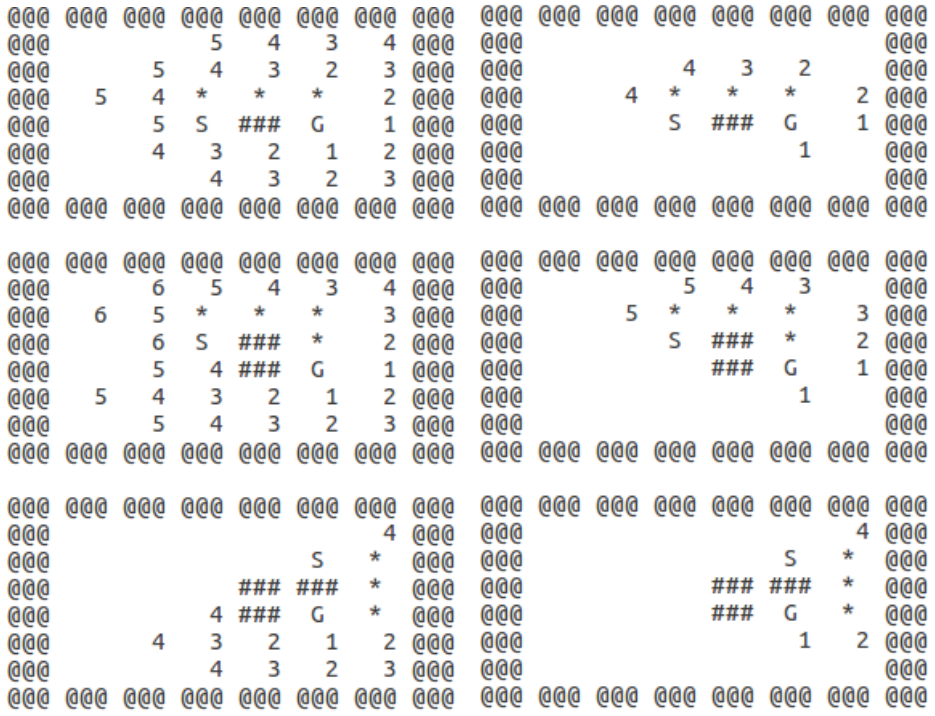


(a) BFS (video)

(b) A* (video)

Figure 9.1: Map 1

In figure 9.1 both paths that are generated from the algorithms are similar, however the position of the ePuck is a bit off with the A*. The path with BFS took 16 seconds and with A* took 15 seconds which makes them very similar in this case. In the figure below the obstacle detection is shows for each algorithm.



(a) BFS (b) A*

Figure 9.2: Obstacle detection map 1

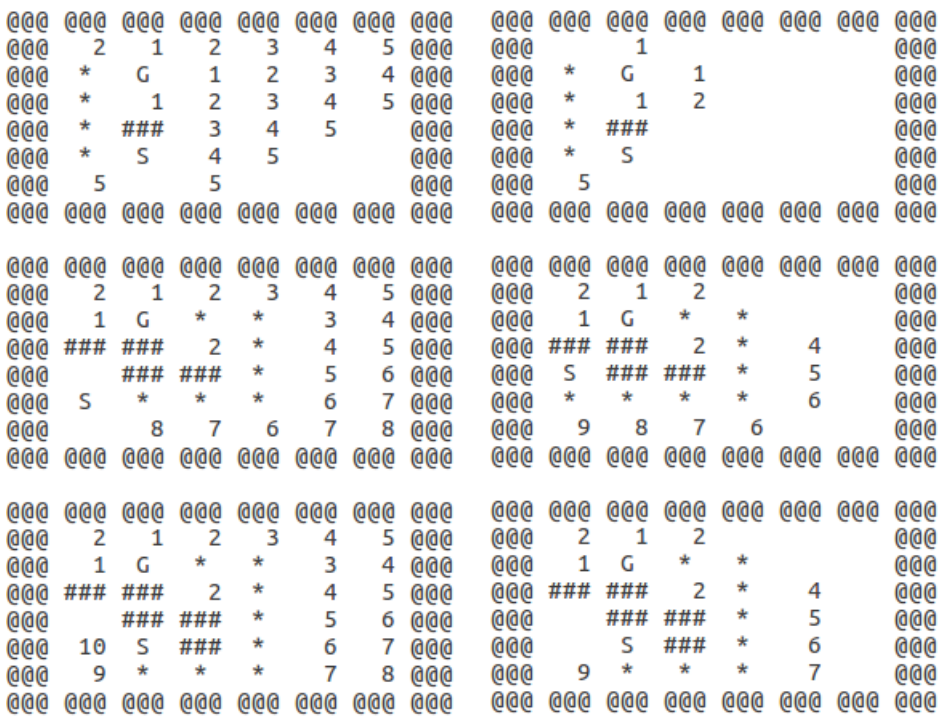
Map 2

(a) BFS (video)

(b) A* (video)

Figure 9.3: Map 2

In figure 9.3 the generated paths are still similar but they still have some differences in their planning. When the path was generated with A* the ePuck deviated from the environment a bit, which was likely caused by some over-rotation when turning around a lot. It took 20 seconds for both paths for the ePuck to go from start to goal.



(a) BFS (b) A*

Figure 9.4: Obstacle detection map 2

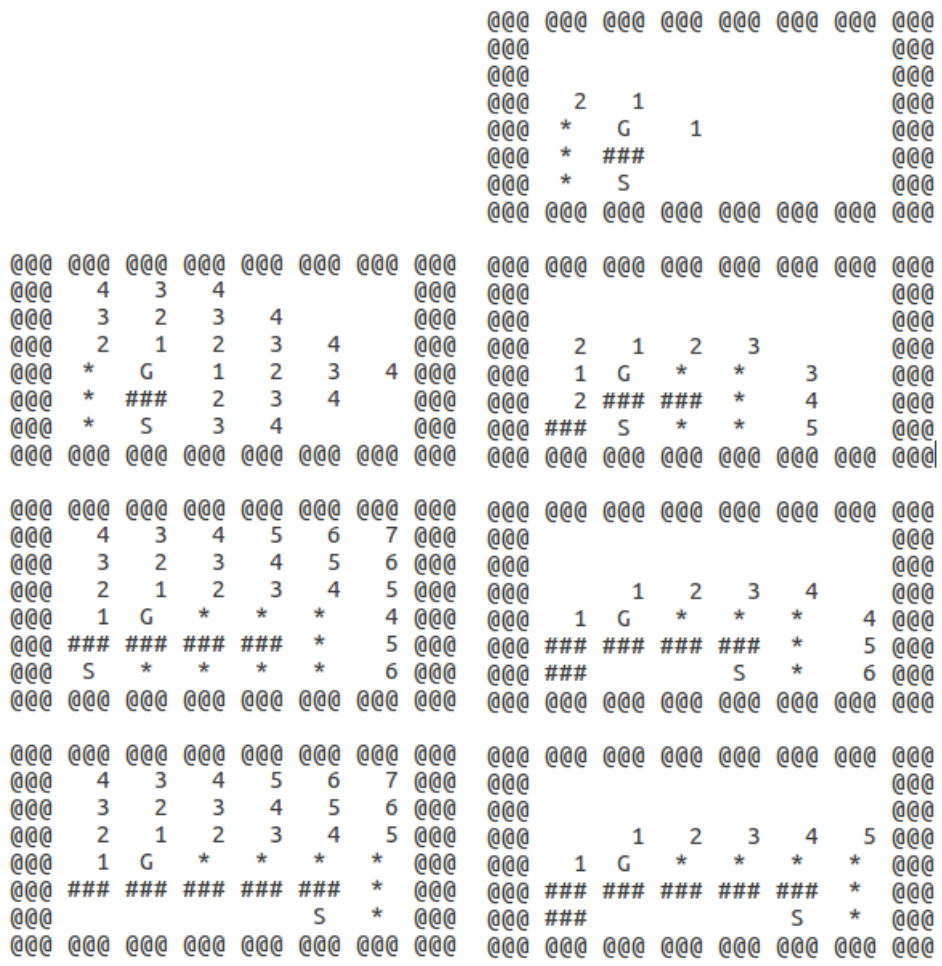
Map 3

(a) BFS (video)

(b) A* (video)

Figure 9.5: Map 3

Figure 9.5 shows us yet again the similarity between the algorithms. This time however, the path generated from BFS was about 1 second faster than A* (18 seconds vs 19 seconds). Again there are a lot of factors which may cause this. One is that the ePuck didn't rotate as much on one path, maybe it also didn't get as close to the wall as the other one did, which made it slow down.



(a) BFS (b) A*

Figure 9.6: Obstacle detection map 3

As seen in the obstacle detection for A* in figure 9.6 there are times when an obstacle is added when there's no obstacle there. This is likely caused by the inaccuracy of the sensors and how the ePuck rotated when it was following the path. This will be further discussed in the discussion section.

In theory both of the paths generated by respective algorithm should've taken the exact same time to execute, with this map size of course. In the next section the time measurements will be presented on how long it takes to find a path for each algorithm.

9.2 Time Comparison

Comparing the run time of the algorithms was done in five simple tests. Only the actual run time of the algorithms themselves was measured with the C library function clock(). No printing to the

console or similar was included to eliminate as much time interference as possible. The different maps are filled with empty cells.

The start position is always at the upper left corner and the goal position is always at the lower right corner for all tests.

Test 1

Map size	Cells	Start	Goal	A* (ms)	BFS (ms)
8x8	6^2	(2, 2)	(7, 7)	0.072	0.076

Test 2

Map size	Cells	Start	Goal	A* (ms)	BFS (ms)
50x50	48^2	(2, 2)	(49, 49)	0.452	1.127

Test 3

Map size	Cells	Start	Goal	A* (ms)	BFS (ms)
100x100	98^2	(2, 2)	(99, 99)	1.053	3.891

Test 4

Map size	Cells	Start	Goal	A* (ms)	BFS (ms)
1000x1000	998^2	(2, 2)	(999, 999)	12.197	181.990

Test 5

Map size	Cells	Start	Goal	A* (ms)	BFS (ms)
10000x10000	9998^2	(2, 2)	(9999, 9999)	125.602	16037.370

As one can see, A* wins in terms of how long it takes to find a path. However that's not to say that Breadth-First search is bad. Both of these algorithms performs very well on small maps but A* is the clear winner if there are bigger maps with more cells to search.
Code to be found in appendix:

- A*
- Breadth-First Search
- Time Measurement

Part V

Distance Map Generation & Gradient Following

10 Task

The purpose of using a gradient following controller is to make the ePuck reach a specified destination without planning any sort of path. Instead it will look into each adjacent cell and go to the one with the least distance to the destination, similar to how Greedy Search works. This makes the system closer to a reactive architecture since there are no planning occurring and no detection of obstacles. However, it won't be guaranteed to take the shortest path.

In this case there was an environment model provided which was used to place out obstacles that are in the environment. Doing so makes the distance generation for each cell in the grid be correctly calculated and also eliminates the uncertainty of the environment.

The distance generation for the map was generated by using the Breadth-First Search algorithm, which expands in all cells around the destination, until it reaches the start position. The cells were labelled and assigned a heuristic value using the Euclidean formula from the position of the ePuck to the destination. The algorithm was activated for every new cell the ePuck entered, so that the heuristic values were updated.

Using the gradient following on the map, the ePuck was able to go to one of the adjacent cells with the least heuristic value, moving it closer to the goal, while avoiding obstacles.

11 Conclusions / Tests

One problem that was encountered at first when using a set of difficult obstacles in the environment was that in some cells the heuristic values was the same. This caused the ePuck to move back and forth between two cells infinitely, since it would update the values when moving into each cell. It was solved by adding the value of the label of the cell together with the heuristic value. This solved the problem since the labels were smaller the closer it was to the goal. In figure 11.1 you may see a map on how the ePuck travelled in its own mind and compare it to a video which shows that it deviated a bit from the environmental position. This was most likely caused by the small motions caused when it was trying to pass the first obstacle.

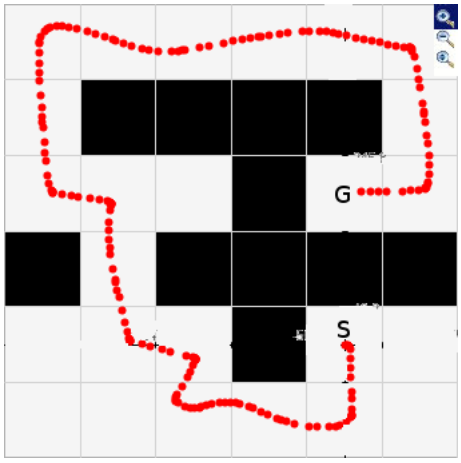
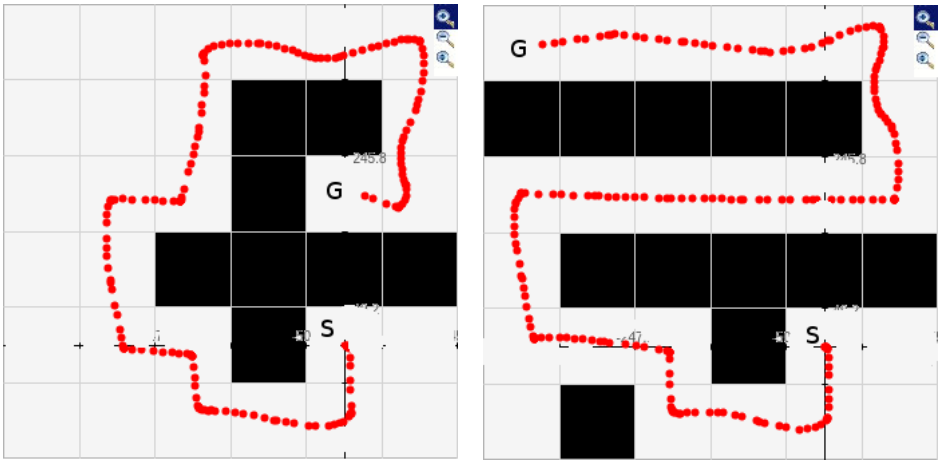


Figure 11.1: Gradient following Map 1(video)

In the video it was seen that the ePuck touched the obstacle. One possible reason for this might be that the top-right sensor wasn't properly tuned and noticed the barrier too late, which caused it to rotate to left after it had hit it. Another reason is that it went too fast, thinking it had free passage and didn't stop or reduce the speed until the sensor on the right had a high value. Overall, it managed to complete some difficult maps without deviating too much from the environment seen in figure 11.2.



(a) Map 2 (b) Map 3

Figure 11.2: Different Gradient following Maps

Code to be found in appendix: Gradient Following

Part VI

Discussion

Combining rules

For implementing the fuzzy rule-based control for obstacle avoidance, we first used the monolithic approach first because we didn't really have a preference. Without making our own fuzzy rules, the ePuck performed okay if we used a static obstacle map without unexpected obstacles. After some tuning on the thresholds and increasing the speed factors, we could see some improvements (Increased speed, quicker turns etc.). A higher speed also sometimes resulted in better position estimation. It seemed like the ePuck was misplaced while turning too slow for example.

The same fuzzy rules but with added unexpected obstacles made the ePuck perform very poor and get stuck even in what could be considered as the simplest situations. The figures in the tests of Part I can show this. Sometimes the fuzzy sets were empty which means that no value is given to, for example, the rotation fuzzy predicate set. To solve this, more rules are needed for handling these situations. We also made a few tests with the modular approach which at that point didnt make any difference, so instead we went back to monolithic and stayed there.

Custom fuzzy rules

The implementation of our own fuzzy rules took a lot of time for us. We had trouble to get into the concept of fuzzy logic, so that caused us to be ineffective in creating good rules.

We tried different combinations depending on each imaginable situation of the ePuck's state, but ended up in situations where the rules were too long and hard to follow.

But in the end, we started from the beginning with no rules and started to construct them them from scenarios where it failed and continued that way until we had a ruleset that worked. The program still fails sometimes, but the pictures from the tests in Part III shows the results from how it most of the time works.

Obstacle detection

We already had a plan about updating the map with obstacles was implemented. The IR-sensors was the most essential tool combined with good position estimation. With our implementation we had to decide in which cell a detected obstacle was located, depending on the location of the ePuck, so some calibration had to be done to each IR-sensor to compensate for the eventual noise. All in all, the updating of the map went good. Though sometimes the lack in precision made an obstacle appear in the wrong cell, often when the obstacle was on or near the edge between two cells. An example is when the ePuck moves near the edges (walls) of the map. Better calibration and a more precise MATLAB generated function could possibly help solving these problems.

Gradient following

The implementation of gradient following was made quickly in the end without combining it with obstacle detection. This made it work fine on static maps were no unexpected obstacles existed. An advantage with gradient following is that no plan is needed, but therefore it is not able to guarantee that it is moving on the closest path.

Appendix

Path following function

```
// Wrapper function for being able to use Track() again after a run is
   done.
void FollowPath()
{
    InitMap = *CreateMapFromFile(SYM, MapFile_Save);
    WorkMap = *CreateMapFromFile(SYM, MapFile_Save);
    int run = 1;

    while (run)
    {
        WorkMap.map[StartCell.i][StartCell.j] = MAP_START;
        WorkMap.map[GoalCell.i][GoalCell.j] = MAP_GOAL;

        Plan(StartCell, GoalCell);
        PrintMap(&WorkMap);

        int nRoutes = GenerateRoute();
        TranslateCellsToCoords(nRoutes);
        PrintWaypoints(nRoutes);
        Track(WaypointsCoords, nRoutes);
        if (ObsDetected)
            Replan();
        else
            run = 0;
    }
    printf("End of FollowPath\n");
    SaveMapToFile(&InitMap, SYM, MapFile_Save);
}
```

Track function

```
// Goes through a list of waypoints to instruct the ePuck where to go.
void Track(Position positions[], int size)
{
    int i;
    for (i = 0; i < size; ++i)
    {
        printf("Track position coords: (%.2lf, %.2lf)\n",
           positions[i].x, positions[i].y);
        GoToAvoid_fuzzy(positions[i].x, positions[i].y);
        if (ObsDetected)
        {
            // Abort the track to replan
            return;
        }
    }
}
```

Obstacle Detection function

```
// Scans for nearby obstacles.
int DetectObstacle()
{
    IR = GetIR();
    int obsAheadVal = 500;
    int obsLeftVal = 700;
    int obsRightVal = 700;
    double distOffset = 15.0;

    if (IR.sensor[0] > obsAheadVal || IR.sensor[7] > obsAheadVal)
    {
        double distAvg = ObsDistance(max(IR.sensor[0], IR.sensor[7]),
           DIST_F_AHEAD);
        SetObstacle(0.0, distAvg + distOffset);
    }

    if (IR.sensor[2] > obsRightVal)
    {
        double distance = (ObsDistance(IR.sensor[2], DIST_F_RIGHT));
        SetObstacle(-PI / 2, distance + distOffset);
    }

    if (IR.sensor[5] > obsLeftVal)
    {
        double distance = (ObsDistance(IR.sensor[5], DIST_F_LEFT));
        SetObstacle(PI / 2, distance + distOffset);
    }

    return ObsDetected;
}
```

Setting Obstacle function

```
// MATLAB generated function which calculates distance from the ePuck
   to an obstacle.
double ObsDistance(int IRvalue, float F)
{
    return (8.7 * pow(10, -7) * pow(IRvalue, 2) - 0.0087 * IRvalue +
       23) * F;
}

// Update a cell with a detected object.
void SetObstacle(float offsetAngle, float distance)
{
    // Check in which cell the IR sensor detected something.
    Cell c = TranslateCoordToCell(cos(RobPos.th + offsetAngle) *
       (distance + ROBOT_RAD) + RobPos.x,
                                  sin(RobPos.th + offsetAngle) *
                                     (distance + ROBOT_RAD) +
                                     RobPos.y);

    int cVal = WorkMap.map[c.i][c.j];
    if (cVal != MAP_OBSTACLE && cVal != MAP_BORDER)
    {
        // printf("Obstacle not in map!\n");
        InitMap.map[c.i][c.j] = MAP_OBSTACLE;
        if (cVal == MAP_TRACE)
        {
            ObsDetected = 1;
        }
    }
}
```

Plan and Replan functions

```
// Plans the shortest waypoint path between start and goal.
void Plan(Cell start, Cell goal)
{
    // Comment out the algorithm below which will not be used.
    Search(start, goal);
    // AStarSearch(start, goal);
    Cell curr = start;
    Cell next;
    nWaypoints = 0;
    printf("C:(%d, %d) (Start)\n", curr.i, curr.j);
    do
    {
        next = GetNextNeighbour(curr);
        if (nWaypoints)
            WorkMap.map[curr.i][curr.j] = MAP_TRACE;
        PathCells[nWaypoints] = next;
        nWaypoints++;
        printf("C:(%d, %d) (MVal: %d)\n", next.i, next.j,
           WorkMap.map[next.i][next.j]);
        if (curr.i == next.i && curr.j == next.j)
        {
            printf("PATH NOT FOUND\n");
            exit(0);
        }
        curr = next;
    } while (WorkMap.map[curr.i][curr.j] != MAP_GOAL);
}
```

Replan

```
// New obstacle aborted current run. Plan a new route.
void Replan()
{
    printf("Added obstacle to cell ahead. Replanning!\n");

    // Updating Map with new obstacles
    SaveMapToFile(&InitMap, SYM, MapFile_Save);
    WorkMap = *CreateMapFromFile(SYM, MapFile_Save);

    ObsDetected = 0;

    // Sets the new position to start from
    Cell RobCell = TranslateCoordToCell(RobPos.x, RobPos.y);
    if (WorkMap.map[RobCell.i][RobCell.j] == MAP_OBSTACLE)
    {
        // Obstacle detected in same cell as ePuck. Return to last
           visited path cell.
        ReturningtoPrev = 1;
        prevpos = TranslateCelltoCoords(PrevCell);
        printf("Going back to prev cell\n");
        GoToAvoid_fuzzy(prevpos.x, prevpos.y);
        ReturningtoPrev = 0;

        if (WorkMap.map[PrevCell.i][PrevCell.j] != MAP_OBSTACLE)
        {
            StartCell.i = PrevCell.i;
            StartCell.j = PrevCell.j;
        }
    }
    else
    {
        StartCell.i = RobCell.i;
        StartCell.j = RobCell.j;
    }

    // Resets used lists
    memset(PathCells, 0, sizeof(PathCells));
    memset(WaypointCells, 0, sizeof(WaypointCells));
    memset(WaypointsCoords, 0, sizeof(WaypointsCoords));
}
```

GetNextCellNeighbour

```
// Checks the four neigbours of a cell to plan a path.
Cell GetNextNeighbour(Cell c)
{
    int next_nr = 99999;
    Cell next_c;
    if (WorkMap.map[c.i - 1][c.j] < next_nr && WorkMap.map[c.i -
       1][c.j] >= 0)
    {
        next_nr = WorkMap.map[c.i - 1][c.j];
        next_c.i = c.i - 1;
        next_c.j = c.j;
    }
    if (WorkMap.map[c.i + 1][c.j] < next_nr && WorkMap.map[c.i +
       1][c.j] >= 0)
    {
        next_nr = WorkMap.map[c.i + 1][c.j];
        next_c.i = c.i + 1;
        next_c.j = c.j;
    }
    if (WorkMap.map[c.i][c.j - 1] < next_nr && WorkMap.map[c.i][c.j -
       1] >= 0)
    {
        next_nr = WorkMap.map[c.i][c.j - 1];
        next_c.i = c.i;
        next_c.j = c.j - 1;
    }
    if (WorkMap.map[c.i][c.j + 1] < next_nr && WorkMap.map[c.i][c.j +
       1] >= 0)
    {
        next_nr = WorkMap.map[c.i][c.j + 1];
        next_c.i = c.i;
        next_c.j = c.j + 1;
    }
    return next_c;
}
```

Generate a route with waypoints from the pathplanning without unnecessary waypoints

```
// Generates the final waypoints by removing all redundant waypoints
   in the path.
int GenerateRoute()
{
    int i_change = 0, j_change = 0, route_index = 0;
    WaypointCells[route_index++] = PathCells[0];
    for (int i = 1; i < nWaypoints; ++i)
    {
        if (!i_change)
            if (PathCells[i].i - PathCells[i - 1].i != 0)
            {
                i_change++;
            }
        if (!j_change)
            if (PathCells[i].j - PathCells[i - 1].j != 0)
            {
                j_change++;
            }
        if (abs(i_change) && abs(j_change)) // Did both i and j values
           change? (Did the ePuck turn?)
        {
            WaypointCells[route_index++] = PathCells[i-- - 1]; // If
               the ePuck turned, add waypoint.
            i_change = 0;
            j_change = 0;
        }
    }
    WaypointCells[route_index++] = PathCells[nWaypoints - 1]; // The
       goal has to be the last waypoint.
    return route_index;
}
```

GoToAvoid and Fuzzy Rules

```
void GoToAvoid_fuzzy(double xt, double yt)
{
    double vlin, vrot;
    do
    {
        // New detected obstacle in the current planned path? Abort
           and replan.
        if (DetectObstacle() && !ReturningtoPrev)
        {
            Stop();
            return;
        }

        UpdatePosition();
        ClearFSet(f_set_vlin);
        ClearFSet(f_set_vrot);
        GotoAvoidRules(xt, yt);

        DeFuzzify(f_set_vrot, 3, &f_vrot);
        DeFuzzify(f_set_vlin, 4, &f_vlin);

        vlin = ResponseToVel(f_vlin);
        vrot = ResponseToRot(f_vrot);

        SetPolarSpeed(vlin, vrot);
        Sleep(10);
    } while (Epos > 30);
    Stop();
}
```

```
void GotoAvoidRules(double xt, double yt)
{
    // GoTo ------------------------------------------------------------
    double dx, dy;
    FPred Pos_Left, Pos_Right, Pos_Ahead, Pos_Here;

    // Compute Eth and Epos
    dx = xt - RobPos.x;
    dy = yt - RobPos.y;
    Epos = sqrt(pow(dx, 2) + pow(dy, 2));
    Eth = NormAng(atan2(dy, dx) - RobPos.th);

    Pos_Left = RampUp(Eth, 0, PI / 3.0);
    Pos_Right = RampDown(Eth, PI / -3.0, 0);
    Pos_Ahead = min(RampUp(Eth, PI / -7.0, 0), RampDown(Eth, 0, PI /
       7.0));
    Pos_Here = RampDown(Epos, 5, 50);

    // Avoid
       ------------------------------------------------------------
    FPred Obs_Left, Obs_Right, Obs_Ahead, Obs_Back;

    IR = GetIR();
    Obs_Left = RampUp(max(IR.sensor[5], IR.sensor[6]), 300, 1000);
    Obs_Right = RampUp(max(IR.sensor[1], IR.sensor[2]), 300, 1000);
    Obs_Ahead = RampUp(max(IR.sensor[0], IR.sensor[7]), 200, 800);

    RULESET;
    // Rotation rules
    IF(AND(Pos_Left, NOT(Obs_Left)));    ROT(LEFT);
    IF(AND(AND(Obs_Right, NOT(Obs_Left)), NOT(Pos_Ahead)));
       ROT(LEFT);
    IF(AND(Pos_Ahead, Obs_Right));    ROT(LEFT);

    IF(AND(Pos_Right, NOT(Obs_Right)));    ROT(RIGHT);
    IF(AND(AND(Obs_Left, NOT(Obs_Right)), NOT(Pos_Ahead)));
       ROT(RIGHT);
    IF(AND(Pos_Ahead, Obs_Left));    ROT(RIGHT);

    IF(AND(Pos_Ahead, NOT(Obs_Ahead)));    ROT(AHEAD);

    // Velocity rules
    IF(AND(AND(AND(Pos_Ahead, NOT(Pos_Here)), AND(NOT(Obs_Left),
       NOT(Obs_Right))), NOT(Obs_Ahead)));    VEL(FAST);
    IF(OR(Pos_Here, NOT(Pos_Ahead)));     VEL(NONE);
    IF(Obs_Ahead);                    VEL(NONE);
    IF(AND(OR(Obs_Right, Obs_Left), NOT(Obs_Ahead)));    VEL(SLOW);

    RULEEND;
}
```

Breadth-First Search

```
// Breadth-First Search
void Search(Cell start, Cell goal)
{
    ClearList(&SearchList, FIFO);
    Push(&SearchList, goal);
    while (!IsListEmpty(&SearchList))
    {
        Cell current = Pop(&SearchList);
        if (current.i == start.i && current.j == start.j)
            break;
        int dist = WorkMap.map[current.i][current.j] + 1;
        MarkCell(current.i, current.j - 1, dist);
        MarkCell(current.i, current.j + 1, dist);
        MarkCell(current.i - 1, current.j, dist);
        MarkCell(current.i + 1, current.j, dist);
        PrintMap(&WorkMap);
    }
}
```

A*

```
void AStarSearch(Cell start, Cell goal)
{
    ClearList(&SearchList, SORTED);
    Push(&SearchList, goal);
    while (!IsListEmpty(&SearchList))
    {
        Cell current = Pop(&SearchList);
        if (current.i == start.i && current.j == start.j)
            break;
        int dist = WorkMap.map[current.i][current.j] + 1;
        MarkCell(current.i, current.j - 1, dist);
        MarkCell(current.i, current.j + 1, dist);
        MarkCell(current.i - 1, current.j, dist);
        MarkCell(current.i + 1, current.j, dist);
        PrintMap(&WorkMap);
    }
}
```

Time measurement function for BFS and A*

```
// Small time measuring program for comparing A* with BFS.
void TimeMeasureAstarBFS()
{
    int size;
    int algChoice;

    printf("Choose algorithm:\n1. A*\n2. BFS\n");
    scanf("%d", &algChoice);
    printf("Enter map side size:\n");
    scanf("%d", &size);

    StartCell = {1, 1, 0.0};
    GoalCell = {size - 2, size - 2, 0.0};

    WorkMap = *CreateBorderedMap(size, size, DEFAULT_MAX_VALUE);
    WorkMap.map[StartCell.i][StartCell.j] = MAP_START;
    WorkMap.map[GoalCell.i][GoalCell.j] = MAP_GOAL;

    clock_t start;
    switch (algChoice)
    {
    case 1:
        printf("Running A* on map side size %d...\n", size);
        start = clock();
        AStarSearch(StartCell, GoalCell);
        break;
    case 2:
        printf("Running BFS on map side size %d...\n", size);
        start = clock();
        Search(StartCell, GoalCell);
        break;
    default:
        break;
    }
    clock_t end = clock();
    float ms = (float)(end - start) / CLOCKS_PER_SEC * 1000;
    printf("Measured milliseconds: %lf\n", ms);
}
```

MarkCell function for BFS and Astar functions

```
// Labels cells with information while searching.
void MarkCell(int i, int j, int label)
{
    Cell c = {i, j, 0.0};
    int di, dj;
    switch (WorkMap.map[i][j])
    {
    case -1: // Initial position
        Push(&SearchList, c);
        break;
    case -2: // Unexplored c
    case -6:
        WorkMap.map[i][j] = label;
        c.h_value = heuristic(i, j, StartCell);
        HValueList[i][j] = heuristic(i, j, GoalCell);
        Push(&SearchList, c);
        break;
    case -3: // Obstacle
        c.h_value = 0.0;
        break;
    case -4: // Out of boundaries
        break;
    default: // Already explored
        break;
    }
}
```

Get the best cell for FollowGradient function

```
// Evaluates the four neighbours of a cell while gradient following.
   Returns the current best choice.
Cell GetBestGradient(Cell robc)
{
    double next_nr = 99999.99;
    Cell next_c;
    Cell neighbours[4] = {{robc.i, robc.j - 1},
                          {robc.i - 1, robc.j},
                          {robc.i + 1, robc.j},
                          {robc.i, robc.j + 1}};
    for (int i = 0; i < 4; ++i)
    {
        neighbours[i].h_value =
           HValueList[neighbours[i].i][neighbours[i].j];

        double cellVal = WorkMap.map[neighbours[i].i][neighbours[i].j];
        double cellH = HValueList[neighbours[i].i][neighbours[i].j] +
           cellVal;

        if (cellH < next_nr && cellH > 0.0 &&
            cellVal != MAP_BORDER &&
            cellVal != MAP_OBSTACLE)
        {
            next_c.i = neighbours[i].i;
            next_c.j = neighbours[i].j;
            next_c.h_value = next_nr = neighbours[i].h_value = cellH;
        }
        else if (WorkMap.map[neighbours[i].i][neighbours[i].j] ==
           MAP_GOAL)
        {
            next_c = GoalCell;
            break;
        }
    }
    return next_c;
}
```

Gradient Following function

```
// Function for making the ePuck follow the gradient.
void FollowGradient()
{
    InitMap = *CreateMapFromFile(SYM, MapFile_Gradient);
    WorkMap = *CreateMapFromFile(SYM, MapFile_Gradient);

    WorkMap.map[StartCell.i][StartCell.j] = MAP_START;
    WorkMap.map[GoalCell.i][GoalCell.j] = MAP_GOAL;
    do
    {
        Search(StartCell, GoalCell);
        PrintMap(&WorkMap);
        Cell newCell = GetBestGradient(CurrCell);
        Position coords = TranslateCelltoCoords(newCell);
        GoToAvoid_fuzzy(coords.x, coords.y);
        StartCell = CurrCell;

    } while (!(CurrCell.i == GoalCell.i && CurrCell.j == GoalCell.j));
    printf("End of FollowGradient\n");
}
```

Update Position

```
// Calculates and updates the estimated position of the ePuck by
   approximation.
void UpdatePosition()
{
    newSteps = GetSteps();
    diffSteps.l = newSteps.l - prevSteps.l;
    diffSteps.r = newSteps.r - prevSteps.r;
    double diffmmL = (diffSteps.l) / STEPS_PER_MM;
    double diffmmR = (diffSteps.r) / STEPS_PER_MM;

    double dist = (diffmmL + diffmmR) / 2;
    double ang = (diffmmR - diffmmL) / CRD;

    double dx = dist * cos(ang / 2);
    double dy = dist * sin(ang / 2);

    prevSteps = newSteps;
    RobPos.x += dx * cos(thPrev) - dy * sin(thPrev);
    RobPos.y += dx * sin(thPrev) + dy * cos(thPrev);

    double th = NormAng(ang + thPrev);
    RobPos.th = th;
    thPrev = th;

    CurrCell = TranslateCoordToCell(RobPos.x, RobPos.y);

    // Keeps track of the last visited cell that was part of the
       calculated path.
    for (int i = 1; i < nWaypoints; ++i)
    {
        if (PathCells[i].i == CurrCell.i && PathCells[i].j ==
           CurrCell.j)
        {
            PrevCell.i = PathCells[i - 1].i;
            PrevCell.j = PathCells[i - 1].j;
        }
    }
}
```

Translation Functions from Coordinates to Cell vv.

```
// Translates the position of each waypoint cell into real coordinates.
void TranslateCellsToCoords(int nCoords)
{
    for (int i = 0; i < nCoords; ++i)
    {
        Position coords = TranslateCelltoCoords(WaypointCells[i]);
        WaypointsCoords[i].x = coords.x;
        WaypointsCoords[i].y = coords.y;
    }
}

// Translate a map cell into coordinates.
Position TranslateCelltoCoords(Cell c)
{
    double x = (c.j - RefCell.j) * mmCell;
    double y = (RefCell.i - c.i) * mmCell;
    Position coords = {x, y};
    return coords;
}

// Translate coordinates into a map cell.
Cell TranslateCoordToCell(float x, float y)
{
    Cell result;
    result.j = round(x / mmCell + RefCell.j);
    result.i = round(RefCell.i - y / mmCell);
    return result;
}
```

Small Functions

```
int ResponseToVel(float response)
{
    // If the response is 0.33, the linear speed should be 0.
    const double limit = 0.33;
    return -(limit - response) * VMAX_;
}

double ResponseToRot(float response)
{
    // If the response is 0.5, the rotation speed should be 0.
    const double limit = 0.5;
    double factor = (limit - response);
    return factor * PI * ROTFACTOR;
}

// Calculates heuristic value with the Pythagorean theorem.
double heuristic(int i, int j, Cell c)
{
    int di = i - c.i;
    int dj = c.j - j;
    return sqrt(pow(di, 2) + pow(dj, 2));
}

// Translates an angle to a value between -PI and PI.
double NormAng(double a)
{
    double norm_a = a;
    if (a > PI)
        norm_a -= 2 * PI;
    else if (a < -PI)
        norm_a += 2 * PI;
    return norm_a;
}
```