Project Report "Intelligent Robotic Systems"

Giovanni's Micromouse

March 2024

Group members:
Daniele Gambaletta daniele.gambaletta@studio.unibo.it
Riccardo Omiccioli riccardo.omiccioli@studio.unibo.it
Cecilia Teodorani cecilia.teodorani@studio.unibo.it

Contents

1 Introduction

The project takes inspiration from the international competition *Micromouse*, organized by the Institute of Electrical and Electronics Engineers (IEEE). In this competition, participants design and build a physical robotic mouse capable of autonomously navigating a maze from a designated corner to its center. The winning team is determined by the robot that completes the maze in the shortest time. This competition challenges participants' design and programming skills and encourages innovation in autonomous navigation and artificial intelligence.
The project is developed based on the official rules of the competition, with some variations and adaptations. Therefore, strategies were devised and implemented to create a simulated system that enables a robot named Giovanni to navigate unknown mazes, identify the optimal path, and reach the destination point efficiently and accurately.
In the following report, after examining the details of the official *Micromouse* competition, key aspects of the project are explored, including robot design details, implemented navigation algorithms, challenges faced during development, and achieved results.

2 State of Art

Since the late 1970s, the IEEE has annually hosted *Micromouse* competitions, pioneering this type of event, although many similar competitions have since been organized worldwide. Given that the IEEE was the first to organize such competitions, the project *Giovanni's Micromouse* chose to adhere to their official rules. Participants in the competition are IEEE student members who compete individually or in groups of up to five people. They must design and develop a robotic mouse capable of autonomously navigating an unknown maze. The robot must fit within a maximum size of 25 cm x 25 cm and must not lose any parts during the competition. The robot's code cannot be modified after the maze is revealed at the beginning of the competition.

The maze comprises 16 x 16 unit squares, each measuring 18 cm, with walls obstructing the robot's passage. The robot starts from a start square and must reach the center of the maze, where the endpoint is located, aiming to do so in the shortest time possible. Each robot is assigned 10 minutes of access to the maze, including any adjustments made between runs, and the shortest run time within this period determines its official time. Multiple runs can be attempted within the 10-minute time limit. The timer starts when the mouse crosses the start line and stops when the mouse crosses the finish line. If the mouse has not entered the destination square, no run time is recorded. Prizes are awarded based on the shortest official times achieved.

To navigate within an unknown maze, elements of robot navigation are necessary, including mapping, planning, and localization, in addition to search algorithms. The *Wall-Following* algorithm cannot be used because the regulations specify that the maze is structured so that the robot cannot reach the final destination by following a single wall. *Flood Fill* is certainly the most efficient search technique recommended, whereas *Depth-First*, for example, is an intuitive but entirely inefficient algorithm due to the significant time waste involved in searching the entire maze.

3 Requirements Analysis

3.1 Functional Requirements

Functional requirements are the specific functionalities that the system must have to be considered correct:

- the robot must move from an initial point to a final point within an unknown maze;
- the starting point of the maze is located in one of the corners of the maze;
- the starting line is placed between the first and second cells of the maze;
- the initial cell has three walls around it and only one exit;
- the time starts running when the robot exits the initial cell;
- Giovanni recognizes the maze's starting point by checking the floor colour;
- while the robot journeys to the endpoint, it stores data related to the maze;
- while navigating, Giovanni moves while attempting to avoid contact with walls;
- if the robot touches a wall, a penalty is incurred for each second of contact;
- the robot identifies the maze's endpoint by checking the floor colour, which differs from the starting point's colour;
- the endpoint consists of 2x2 coloured cells positioned at the center of the maze;
- the endpoint has only one entrance;
- the robot and the time stop when Giovanni reaches the endpoint;
- during each run, the robot can access the maze for a maximum duration of 10 minutes;
- in case Giovanni fails to reach the endpoint, its path time is considered equal to the maximum time limit;

- the robot cannot be remotely controlled;
- once the maze is revealed, the robot's code cannot be modified until the end;
- Giovanni cannot jump, fly, or climb a wall of the maze, nor can he damage or destroy them;
- the maze consists of 8x8 cells;
- except for the endpoint, at least one wall must extend from each vertex of a cell. This means that there can be no "open spaces" formed by 2x2 cells or more inside the maze;
- the endpoint is positioned in such a way that a robot following one of the maze walls cannot reach it;
- after completing a journey from the starting point to the final one, Giovanni can undertake a subsequent run to try to achieve a better time. This is possible by resetting the simulation, as the robot is placed back in the initial cell, while retaining some stored data related to the previous run.

3.2 Non-functional Requirements

Non-functional requirements focus on how the system should behave and respond in terms of performance, reliability, security, usability, and other qualities that contribute to its overall efficiency:

- the system must be able to store relevant data consistently;
- the system must be fault-tolerant, meaning it should continue to operate reliably and should never halt in the event of an error, but rather continue without generating failures;
- the system's operation must be ensured using different developed solutions, considering the same simulation configurations.

3.3 Implementative Requirements

Implementative requirements define the guidelines and technical specifications that must be followed, focusing on programming languages, platforms, and specific tools necessary for software development:

- the system must use the robotic simulator "ARGoS", properly configured;
- the programming language "LUA" will be used for software development;
- the code must be version-controlled using "Git", with a repository available on the "GitHub" hosting platform.

4 Design

Considering the project's nature, a **Behavior-based** approach was chosen. The use of a **Subsumption Architecture** facilitated the independent development of individual robot behaviours, which were subsequently integrated to achieve the robot's overall functionality. The competences of Giovanni are:

- Level 0 - detection of floor colour: the robot identifies the starting and ending points, as well as the rest of the maze, based on the different floor colours;
- Level 1 - detection of proximity: if Giovanni has nearby obstacles and is touching them, then a penalty is recorded;
- Level 2 - detection of distant obstacles: monitoring obstacles at medium and long distances, not found in the previous level; this way, it is understood where the robot may encounter walls obstructing its passage;
- Level 3 - movement: Giovanni moves straight or turns within the maze;
- Level 4 - calibration: the robot repositions itself at the center of the cell it is traversing, if it had become slightly off-center during previous movements.

4.1 Algorithms

Since the robot does not have prior knowledge of the maze's structure, some algorithms are chosen to make it move within it:

- **Depth-First**: an algorithm that systematically explores the deepest branches of a graph before backtracking. Its exploration begins at a specified starting point, probing as deeply as possible along each path before retracing its steps. It marks visited nodes to prevent revisiting and gracefully backtracks from dead-end nodes, ultimately concluding when all reachable nodes have been traversed;
- **Breadth-First**: an algorithm that systematically explores all the neighboring nodes of a starting node before moving on to the next level of nodes. It starts at a specified starting point, enqueuing it into a queue and marking it as visited, before systematically visiting each node's unexplored neighbors

in a level-by-level fashion. This process continues until all reachable nodes have been traversed, terminating once the queue becomes empty;

- **Flood Fill**: an algorithm that, starting from a cell in a grid-like structure, it explores neighboring cells iteratively, marking them as visited and collecting data. This process continues until most or all reachable areas are covered. It's like pouring water from a starting point, filling all accessible cells. This algorithm is effective for maze solving because it enables efficient navigation and eventually discover the shortest path to a destination.

5 Implementation

5.1 Maze

To represent the maze and store information about its cells, a data structure has been created. It consists of a list containing all the cells of the maze, each of which includes the following details:

- row: a number indicating the row in the maze where the cell is located;
- column: a number indicating the column in the maze where the cell is located;
- visited: a flag set to true when Giovanni passes through the cell for the first time;
- parent: the row and column of the previous cell, from which the robot arrives at the current cell;
- reachable neighbours: a list of cells that are reachable from the current cell, implying no walls between the current and neighbouring cells;
- weight: a number used to assign a level of importance to the cell, useful in the **Flood Fill** algorithm 5.4.3.

5.2 Behaviours

5.2.1 Level 0 - Detection of floor colour

The detection of floor colour is possible thanks to the use of the robot's motor ground sensor. This sensor is chosen for its capability to recognize three distinct coloured areas: the endpoint is black, the starting cell is gray, and the rest of the maze is white. Consequently, this setup enabled the management of the run's start time, initiating it upon the robot's exit from the initial maze cell, and concluding both the time and robot movement upon reaching the final square.

5.2.2 Level 1 - Detection of proximity

The detection of obstacles around the robot is possible thanks to the various proximity sensors it possesses. In the event that Giovanni touches a wall at some point during the journey towards the endpoint, a penalty is incurred for each second that the robot continues to touch the wall.

5.2.3 Level 2 - Detection of distant obstacles

The four distance scanner sensors were used because they can detect obstacles at medium and long distances from the robot. They are crucial during the exploration of reachable neighbours of a cell, as when one of the sensors detects a wall, it returns the distance in cm from the robot and also indicates which sensor performed the measurement. This way, it is possible to understand if the cells adjacent to the one where the robot is present are directly reachable by Giovanni and especially which one; in this case, it will be added to his list of reachable neighbours. Thus, during the robot's movement in the maze, this useful information is saved for understanding the structure of the maze itself.

5.2.4 Level 3 - Movement and positioning

Movement and positioning of the robot inside the maze are mainly done utilizing directly the linear speed function and reading motors encoder position. Both of these low level functions are provided by the ARGoS simulator. The possible movements that the robot is able to do are defined in two different categories: basic movements and complex movements. In particular, basic movements represent the fundamental movements that the robot is programmed to do, such as to go straight for a certain distance and to turn on its center axis by a given angle. A complex movement is defined as a movement that is composed by one or more basic movements and those are: to go straight by a multiple amount of a unit length, to flip its orientation by turning 180 degrees, to flip its orientation and go straight by a unit length and to turn by 90 degreed and go straight forward by one unit. In this context, a unit length corresponds to the size of a maze cell, which is set at 50cm. Movements involving the rotation of the robot are always supposed to be executed in increments of 90 degrees, although it is technically possible to execute finer movements, which are not used in this project.

The following listing is an overview of the code present in the move.lua file which contains all of the code responsible for the robot movements. The following listings are not representative of the real code, numerous code lines are modified and pseudo code elements will be inserted in place of the original code to explain the movement inner workings without going into the specific code.
The first things present in the move code are some constants about the maze or the robot related to movements, followed by constant parameters regarding the robot velocity and acceleration. Some of those constants are tuned base on tests done inside the simulator trying to achieve the best compromise between speed and reliability in movements execution. Of particular significance is the move_array table, which stores a sequence of movements to be executed. As previously mentioned, the robot can perform complex movements by combining multiple simpler ones in succession.

```
MAZE_UNIT_LENGHT = 50
TURN_LINEAR_LENGTH = math.pi * robot.wheels.axis_length / 4
TURN_RADIUS =  robot.wheels.axis_length / 2

MAX_CONSTANT_VELOCITY = 100
MIN_LINEAR_VELOCITY = 100
MAX_LINEAR_VELOCITY = 1000
MAX_ACCELERATION = 4
MAX_DECELERATION = 1

is_moving = false // used to check if a movement is in progress
distance_traveled = 0
distance_to_travel = 0
current_velocity = 0
current_move = nil
move_array = {} // list of movements to be done

function is_stopped()
    return not is_distance_changed()
end
```

The function used to execute a basic movement usually assumes the form exposed in the following listing. This function is used either to start a new basic movement or to continue an existing one. The principle behind this function is to check the distance travelled in the previous step reading motor encoders, accelerate if

the current movement is less than half of the movement to be done, decelerate otherwise and stop if the distance traveled is more or equal of the distance to travel that was expected. The function used to turn the robot follows the same reasoning, but doesn't implement acceleration or deceleration logic, relying on a costant velocity for its movements.

```
// Starts a new straight movement or continues movement
function straight()
  distance_to_travel = current_move.delta or MAZE_UNIT_LENGHT
  if not is_moving then
    distance traveled = 0
    is_moving = true
  else
    distance_traveled = distance_traveled + robot.wheels.distance
    if distance_traveled >= distance_to_travel then
      is_moving = false
    end
  end
  if is a new movement then
    current_velocity = MIN_LINEAR_VELOCITY
  else
    if distance_traveled < distance_to_travel / 2 then
      current_velocity = current_velocity + MAX_ACCELETARION
    elseif distance_traveled >= distance_to_travel / 2 then
      current_velocity = current_velocity - MAX_DECELETARION
    end
    if distance_traveled >= distance_to_travel then
      current_velocity = 0
    end
  end
  if current_move.direction == MOVE_DIRECTION.BACKWARDS then
    current_velocity = -current_velocity
  end
  robot.wheels.set_velocity(current_velocity, current_velocity)
end

// Starts a new turn movement or continues
function turn()
  // similar to straight function
end
```

The move function, exposed in the following listing, is the one expected to be used to control the robot movements. When called passing a movement (and optionally a direction and delta) it put those movements to the list of movements to be done next. Called without any parameters, this function continues a movement if necessary or checks if the robot is stopped. In case the robot is stopped, it checks and eventually executes calibration movements or starts new movements if already scheduled to be executed.

```
function move(movement, direction, delta, has_priority)
  if movement then
    if is a basic movement then
      table.insert(move_array, {movement, direction, delta})
    elseif movement == COMPLEX_MOVE.N_STRAIGHT then
      table.insert(...)
    ... others complex movements
  else
    if is_moving then
      current_move.movement() // continue movement
    else
      if is_stopped() then
        if is_calibration_needed() then
          calibrate_position()
        end
        if #move_array > 0 then
          current_move = table.remove(move_array, 1)
          update_position(...)
          current_move.movement() // start next movement
        end
      end
    end
  end
  return is_moving, #move_array
end
```

In addition to tables that outline the robot's possible movements, the move file incorporates a function for optimizing the robot's movements. This function combines similar moves into a single continuous action, significantly improving both the speed and precision of the robot's movements.

```
BASIC_MOVE = { STRAIGHT = straight, TURN = turn }
COMPLEX_MOVE = { N_STRAIGHT, FLIP, ... }
```

```
function optimize_movements(movements)
    // optimize movements by connecting consecutive movements into
    one continuous movement
    return optimized_movements
end
```

The positioning of the robot is greatly tied with its movements as it relies uniquely on motor encoders to ensure its correct positioning. This is built on the assumption that tire slippage is absent. However, effective positioning drift control is achievable in most scenarios by maintaining low robot speeds and implementing basic calibration methods. The positioning of the robot is updated by passing a movement, direction and delta each time a new movement is started and it keeps track of the virtual position of the robot considering its current position and heading, assuming that the movement is executed perfectly without errors.

```
HEADING = { NORTH = 1, EAST = 2, SOUTH = 3, WEST = 4 }
MAX_MAZE_POSITION = 400

current_heading = HEADING.EAST
current_position = { x = -375, y = 375 }

function update_position(movement, direction, delta)
  if movement == BASIC_MOVE.STRAIGHT then
    if not delta then delta = MAZE_UNIT_LENGHT end
    if delta >= MAZE_UNIT_LENGHT / 2 then
      if direction == MOVE_DIRECTION.FORWARD then
        if current_heading == HEADING.NORTH then
          current_position.y = current_position.y + delta
        elseif current_heading == HEADING.EAST then
          current_position.x = current_position.x + delta
        ...
      end
    end
  end
end

function get_current_row_and_column()
    calculate and return current row and column
end
```

```
function get_current_heading()
    return current_heading
end

function get_current_position()
    return current_position.x, current_position.y
end
```

5.2.5 Level 4 - Calibration

To improve the precision in the robot position after repeated movements, a really simple but quite effective calibration function is implemented in the calibrate.lua file. The calibration uses distance sensors to check if a wall is present to the front or back of the robot and what is the distance from that wall. If the distance read from a wall is greater than a given error threshold, then a corrective movement is computed and scheduled to be executed next. The following listing is an overview of the code implemented for the robot calibration.

```
SENSOR_DISTANCE_FROM_WALL = 14
POSITION_ERROR_THRESHOLD = 1

function is_calibration_needed()
  distance = get front and back distance
  if front or back distance is available then
    if distance - SENSOR_DISTANCE_FROM_WALL >
   POSITION_ERROR_THRESHOLD then
      return true
    end
  end
  return false
end

function calibrate_position()
  local distance = get distance
  if distance > 0 then
    if distance < SENSOR_DISTANCE_FROM_WALL -
   POSITION_ERROR_THRESHOLD then
      local move_distance = SENSOR_DISTANCE_FROM_WALL -
   front_distance
```

```
      move(BASIC_MOVE.STRAIGHT, MOVE_DIRECTION.BACKWARDS,
   move_distance, true)
    ...
    end
  end
end
```

The calibration consists of two functions: one that returns a true or false value depending if the front or back distance from a wall deviates more than a given error threshold; one function that inserts a new corrective movement to adjust the robot position. After every movement, these functions are utilized to verify and act if the robot is either too near or too distant from a wall. They depend on the capability to insert a new move with priority over other pending movements, allowing for correction of positioning errors before proceeding with scheduled actions.

5.3 Pathfinding

The code used to find a path and navigate the maze is mostly contained in the path.lua file. In the case of this project, a path is described as a list of cells (each composed of row and column) that defines a possible path between two explored cells. Given that the maze composition is unknown to the robot, it is possible to calculate a path only between cells that have already been visited or seen by the robot during its journey. To find a path the robot checks the information discovered during the navigation and traces the path of a cell to the start checking in succession the parent of each cell until the initial cell is found. When computing a path between two cells that are already discovered, the robot calculates two paths (one starting from the initial cell and ending to the starting cell; the other one starting from the destination cell and ending to the starting cell) and then finds the first common cell between the two. A final path is created by combining the path to the common cell starting from the initial path with the path from the common cell to the destination cell. The following listings report the pseudocode implemented in the path.lua file. As already stated, the path to the start of the maze from a cell is calculated by checking the parent cells of each consecutive cell. That is done recursively with a function that creates a list of cells traversing the parent node of each one of them until the initial cell is found.

```
function trace_path_to_start(cell, maze)
```

```
  local path = {}

  local function trace_recursively(current_cell)
    insert current_cell in path
    if current_cell parent exists then
      trace_recursively(current_cell.parent)
    end
  end

  trace_recursively(current_cell)
  return path
end
```

As briefly exaplained before, the path between two already explored cells in the maze is done by calculating the two paths, finding the first common cell and creating the final path combining the two paths until the meeting cell.

```
function trace_path_to_target(start_cell, target_cell, maze)
  local start_path = trace_path_to_start(start_cell, maze)
  local target_path = trace_path_to_start(target_cell, maze)

  local path = {}
  local meeting_cell = first common cell between start_path and
   target_path

  // insert path from initial cell to meeting cell
  for i, cell in ipairs(start_path) do
    if cell == meeting_cell then
      break
    end
    table.insert(path, cell)
  end

  // unwind the target path to start until meeting cell
  for i = #target_path, 1, -1 do
    if target_path[i] == meeting_cell then
      break
    end
    table.remove(target_path, i)
  end

  // insert path from meeting cell to destination cell
```

```
  for i = #target_path, 1, -1 do
    table.insert(path, target_path[i])
  end

return path
```

The path file also provides a function that, given a path, calculates the necessary movements to follow that path, considering the heading and position that the robot will have during the navigation.

```
function calculate_path_movements(path)
  movements = {}
  heading = get_current_heading()
  current_row, current_col = get_current_row_and_column()

  // remove the first cell from the path as it is the current cell
  table.remove(path, 1)

  for i = 1, #path do
    local cell = path[i]
    // next path cell is in the same row
    if cell.row == current_row then
      // next path cell is east of the current cell
      if cell.column > current_col then
        if heading == HEADING.NORTH then
          insert turn right and forward movement to movements
        ...
        heading = HEADING.EAST
      ...
      current_row = cell.row
      current_col = cell.column
    // next path cell is in the same column
    elseif cell.column == current_col then
      ...
    end
  end

  optimized_movements = optimize_movements(movements)
  return optimized_movements
end
```

Considering that the creation of a path between two cells is done by traversing each

cell parent, the path found is not always the shortest available but this approach greatly simplifies path calculation and is considered to be good enough for the goal of this project.

5.4 Algorithms

Below, the implementations of the used algorithms are described. The chosen algorithm iterates each time the robot occupies a maze cell. The maze cells are considered as nodes of a graph, interconnected by edges based on the parent cell being the one from which the robot moves to the current cell, with reachable neighbours identified as children of the current cell. The initial cell acts as the root node of the graph.

5.4.1 Depth-First

The **Depth-First** algorithm verifies whether the robot's current cell has been visited. If not, it marks the cell as visited and updates its parent cell to the one inspected before. Additionally, considering the robot's orientation and distance sensor readings, the algorithm checks for surrounding walls and determines which adjacent cells are reachable without without hitting the walls. Regardless of whether the current cell has been visited, it is added as a reachable neighbour for each of its adjacent cells if they haven't been visited yet. The next destination cell is chosen as follows:

- if the current cell has unvisited neighbours, one of them is randomly selected;
- otherwise, the algorithm backtracks through the graph composed of cells to the parent cell with unvisited children.

The path to be followed is computed based on the current cell as the starting point and the chosen destination cell. Finally, the robot moves accordingly.

```
current_row, current_col = get_current_row_and_column()

if not maze.get_cell(current_row, current_col).visited then
    maze.update_visited(current_row, current_col, true)
    maze.update_parent(current_row, current_col, {row = parent.row
   , column = parent.column})
```

```
    check_walls_update_neighbours(maze, current_row, current_col)
end

local reachable_neighbours =
   update_parent_not_visited_reachable_neighbours(maze,
   current_row, current_col)

local not_visited_neighbours = depth_first.
   not_visited_neighbours_cells(reachable_neighbours)

if #not_visited_neighbours > 0 then
    destination = not_visited_neighbours[math.random(1, #
   not_visited_neighbours)]
else
    local parent_row, parent_column = depth_first.
   parent_with_neighbours_not_visited(current_row, current_col)
    destination = maze.get_cell(parent_row, parent_column)
end

calculate_path_and_move(maze, current_row, current_col,
   destination)

parent.row, parent.column = current_row, current_col
```

5.4.2 Breadth-First

The **Breadth-First** algorithm is very similar to the previously described **Depth-First**, the main difference lies in the selection of the destination cell. For this reason, a queue data structure is used, inserting all unvisited reachable neighbours of the current cell. Consequently, the destination cell is the head of the queue, i.e., the cell inserted before all others. This approach ensures the visit systematically traversing each level of the graph's depth.

```
current_row, current_col = get_current_row_and_column()

if not maze.get_cell(current_row, current_col).visited then
    maze.update_visited(current_row, current_col, true)
    maze.update_parent(current_row, current_col, {row = parent.row
   , column = parent.column})
```

```
    check_walls_update_neighbours(maze, current_row, current_col)
end

update_parent_not_visited_reachable_neighbours(maze, current_row,
   current_col, cells_queue)

if not cells_queue:isEmpty() then
    local destination = cells_queue:dequeue()

    calculate_path_and_move(maze, current_row, current_col,
   destination)
end

parent.row, parent.column = current_row, current_col
```

5.4.3 Flood Fill

The **Flood Fill** for maze solving is an algorithm that enables efficient maze exploration to find the shortest path. It consists in two fases: first the robot explores the maze to find most of the possible way to the destination, then it determines the shortest path. To work it requires to know where the destination is. That is why it is widely used in micromouse competitions, where the robot starts from one corner and the four destination cells are in the center of the maze. In this specific case, it has been used an alternative version called **Modified Flood Fill**. It differs from the classic one, only by the fact that instead of flooding the entire maze each time the micromouse reaches a new cell, it floods only the relevant neighbouring cells. This optimization saves computational time and is achieved by employing a queue data structure. It is composed by the following steps:

- Initialize the maze cells weights, calculating the Manhattan distance from the center. Trivially, Manhattan distance is the number of cells that are needed to cross to reach the destination cell, assuming there are no walls. It's calculated as follow:

```
// point1 at (x1,y1) and point2 at (x2,y2), it is |x1 - x2| +
    |y1 - y2|
return math.abs(second_row - first_row) + math.abs(
   second_column - first_column)
```

- After the maze initialization, Giovanni can start the exploration, going for the cell with lower weights, while attempting to avoid previously visited cells, whenever possible.

- When a lower weight cell can't be found, it is time to call the **Modified Flood Fill** algorithm. First the current cell is enqueued, then a while loop is performed until the queue is empty. Inside the loop, the cell is dequeued, its reachable neighbours are obtained and the minimum weight amongst them is calculated. If the weight of the dequeued cell is lower or equal to the minimum, then it needs to be changed in minimum plus one and all the accessible neighbours are enqueued. The code is the following:

```
flood_queue:enqueue(current_cell)

while not flood_queue:isEmpty() do
    local front_cell = flood_queue:dequeue()
    local reach_n = flood_fill.get_reachable_neighbours(
   front_cell)
    local minimum_n = flood_fill.get_minimum_weight_neighbour
   (reach_n)

    if front_cell.weight <= minimum_n.weight then
        front_cell.weight = minimum_n.weight + 1
        for _, r in ipairs(reach_n) do
            flood_queue:enqueue(r)
        end
    end
end
```

- Completed all the step and the flooding, eventually the robot will reach the destination. For complex and larger mazes, expanding exploration becomes particularly essential. To achieve this, it is necessary to designate a different cell as the destination (E.g.: another corner of the maze) by recalculating the Manhattan distance from the destination to the other cells as follows:

```
for i, cell in ipairs(maze) do
    cell.weight = flood_fill.manhattan_distance(row, column,
   cell.row, cell.column)
end
```

Done that, the mouse eventually will reach that cell, updating the maze data with new reachable neighbours and allowing to find an increasingly better path to the main destination.

6 Outcomes

Completed the implementation, to better understand how algorithms behave, it is necessary to test their performance. Each algorithm undergoes five runs. In each run, Giovanni starts, collects maze data until it reaches the endpoint, resets to start position, and then executes a fast run. The maze used for these tests has a size of 8x8 with two paths leading to the destination: one short and one long. This setup allows for a clearer evaluation of each algorithm's performance. A larger maze would have highlighted more the difference between each approach, but would require a considerable amount of time for the micromouse to reach the destination, making the testing difficult to complete.

6.1 Algorithms Comparison

After observing the robot moving in the maze and with the data obtained, it is possible, with the use of the available algorithm documentation, to get both the advantages and disadvantages associated with each approach.

Depth-First Search (DFS)

Run n.	Exploring time (sec)	Fast run (sec)
1	14.083	6.133
2	86.450	27.250
3	155.533	27.250
4	20.300	6.133
5	11.117	6.133

- Advantages: It generally uses less memory compared to BFS, because it explores as far as possible along each branch. It can be more suitable for mazes with deep paths as it explores depth-wise.
- Disadvantages: It does not guarantee the shortest path to the destination. It is strongly dependent by the path decisions taken randomly. It may get stuck in infinite loops if not implemented properly.

Breadth-First Search (BFS)

Run n.	Exploring time (sec)	Fast run (sec)
1	61.067	6.133
2	55.500	6.133
3	65.750	6.133
4	64.017	6.133
5	49.500	6.133

- Advantages: It guarantees the shortest path to the destination.
- Disadvantages: It requires more memory compared to DFS as it has to store all neighbours in a queue. It may not be suitable for mazes with very deep paths as it explores breadth-wise. The exploration time is always high, because it needs to explore all the neighbours before advancing to the next level.

Flood Fill

Run n.	Exploring time (sec)	Fast run (sec)
1	8.083	6.133
2	8.083	6.133
3	14.083	6.133
4	8.083	6.133
5	14.083	6.133

- Advantages: It guarantees to find the destination. The more the robot explores, the more there is the probability of finding the shortest path.
- Disadvantages: It can be inefficient for large mazes or mazes with complex structures due to its nature. Discovering the shortest path is only achievable after exploring a good portion of the maze.

6.1.1 Conclusion

- Completeness: Flood Fill and BFS are complete as they can search all possible paths. DFS is not necessarily complete.

- Optimality: BFS guarantees the shortest path. Flood Fill and DFS do not guarantee optimality.
- Space Complexity: DFS typically uses less memory than BFS and Flood Fill, because it is the only one that does not use a queue.
- Time Complexity: BFS generally has higher time complexity than DFS and Flood Fill, due to its breadth-first nature.

Based on the provided data, it is confirmed that the Flood Fill algorithm is optimal for maze solving in *Micromouse* competitions. It permits to find the best path in a reasonable amount of time and without requiring the complete exploration of the maze.

6.2 Improvements

Given the scope of the project and the limited amount of time available to develop Giovanni's behavior, there are a few areas that could be improved and some assumptions that can not be made in the real world. The following paragraph will be an insight into some of the problems that the team has encountered and how they could be improved with further development.
First of all, this project makes the assumption that the robot moves with very high accuracy and that tires never slips. This is obviously not the case in the real world, expecially when trying to go as fast as possible trough a maze. In fact, after some time has passed, the robot accuracy in movement shows some drift againts the position that the robot should have, expecially for the rotational positioning of the robot. That was partially solved by calibrating the robot front and back distance from a wall when a wall to reference was available. However this behavior did nothig to solve the rotational positioning error as more distance data was required in different positions to calculate some corrective moves. That could be solved by rotating the distance sensor but, given the complexity of handling a spinning sensor that was already fundamental to check the maze walls placement, the team decided to not solve this issue.
Another improvement that could be made if more distance data was available is to check walls ahead of the robot instead of stopping in a cell and then checking the walls placement. That would have improved the time necessary to explore

the maze as the robot could take decisions earlier and wouldn't need to stop in a cell to check the maze composition. In addition, movements could be smoothed out instead of relying on unitary straight movements and 90 degrees rotations to decrease the length to travel to move between cells and to reduce the need to slow down frequently. That would have required a more complex handling of the robot movement, but it is a very common techinque already used in official competitions to save a considerable amount of time during runs.