Path Finding

- as used in Rescuing Robots -

Project Report
ED3-1-E17

Aalborg University
Electronics and Computer Engineering

LaTeX was used for typesetting this report, Code::Blocks for prototyping the code, IAR-Workbench for programming the microcrontroller and GitHub for collaborating as a group [1].

Electronics and Computer Engineering
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Path Finding

Theme:
Microprocessor based Systems

Project Period:
Fall Semester 2017

Project Group:
ED3-1-E17

Participant(s):
Daniel Frederik Busemann
Razvan-Vlad Bucur
Troels Ulstrup Klein

Supervisor(s):
Akbar Hussain

Copies: 1

Page Numbers: 87

Date of Completion:
December 20, 2017

Abstract:

This report describes robotic pathfinding in a fictional rescuing scenario.
The theory got tested on a microcontroller-based robot. A system design for a robot was made, based on a set of requirements. The functionality of the subsystems gets explained more thoroughly in their own chapters, going into all relevant details.
Movement includes a brief overview over stepper motor technologies and our specific circuits to handle them.
Map Handling explains the internal handling of a dynamically changing map, going into detail about the necessary programming.
Scan evaluates different options to monitor the environment, explaining the chosen solution in detail.
Pathfinding discusses different algorithms, explaining how they relate to each other, what functionality to consider for grid-based maps, and goes into detail about how the final choice was made.

Contents

Preface

This report was made by three students from Aalborg University Esbjerg attending the Electronics and Computer Engineering course, with the purpose of completing the P3 project in the third semester. From this point on, every mention of **we** or **the group** refers to the three co-authors listed below.

All code written for this project can be found in the appendix or the GitHub repository [1].

Aalborg University, December 20, 2017

Daniel Frederik Busemann
<dbusem16@student.aau.dk>

Razvan-Vlad Bucur
<rbucur16@student.aau.dk>

Troels Ulstrup Klein
<tklein11@student.aau.dk>

Chapter 1

Introduction

This report documents the development of a pathfinding robot, with the purpose of bettering the group's understanding of computation on a microprocessor based platform.

The set goal of the robot was to be able to find the most efficient way from a given starting point to a given destination on any map. The idea for this project stems from rescue situations, where it would be unsafe for a human rescuer to enter the area.

In order to make this feasible within the time limits of this project, we adapted the RoboCup Junior Rescue rules [2] into specific limitations. These are rules for a competition for high school students, where the goal is to educate about building robots. The focus here is set on rescuing robots, but there are also other RoboCup competitions with different topics.

Those limitations are mainly about the map and surface, and have been altered slightly, to fit the semesters requirements and account for the size of our group.

Analysing the problem showed us, that our high level requirements for a rescue robot were: movability, knowing where the robot is, reacting to a changing environment and calculating a path. We defined limits for these required fields, as can be seen in Table 1.1, to make it possible for us to develop a satisfying result inside the time frame of the project.

Table 1.1: Delimitations for the defined Fields of Focus

Field	Limits
Movability	Even terrain
Placement	Limited map, known size
Reaction to a changing environment	Limited change allowed
Path Calculation	Map can be approximated as a grid

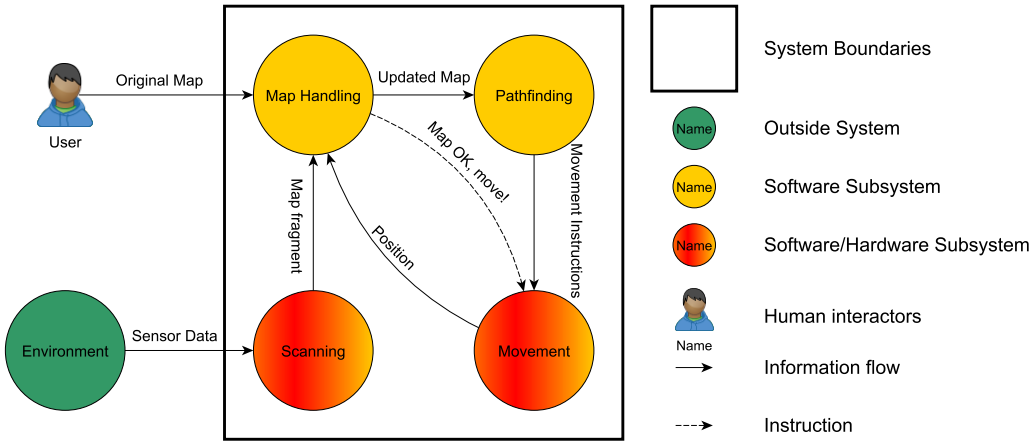


Figure 1.1: Information-Flow diagram of Robot Subsystems

After analysing the problem, we came to the conclusion, that a modular system, as described in Figure 1.1, would fit our needs best. A flow diagram of the whole system working together can be found in Appendix K.

Based on knowledge we got from the book "Object-Oriented Systems Analysis and Design Using UML" [3], we decided to define our system with logical subsystems. This allowed us to handle the different sections of the environment independently, and makes future changes more feasible. It also makes it possible to individually test the subsystems, while other systems may not work yet. We defined four subsystems, each handling one of the high level requirements, mentioned earlier in Table 1.1. Two of those have to handle inputs from the environment (*Scanning* and *Map Handling*), the other two only get inputs from inside the system.

Modularising a system makes simultaneous development and independent testing possible. Our approach created two pure software systems, the other two are a mixture of software and hardware.

We had some issues with acquiring the needed hardware, but thanks to subdivision, we were still able to develop and test the software subsystems.

Reading Guide

The chapters in this report are to be understood as explanations of the previously mentioned subsystems.
Chapter 3 covers how the changing environment is handled. It contains a detailed view of requirements and map design, as well as an in depth discussion of our implementation. Chapter 2 gives an introduction to stepper motors, explains the type of wheels used and goes into detail about the custom built circuit used for

controlling the direction. Chapter 4 introduces the used sensors and explains how their values get combined to be further useful in the other subsystems. Chapter 5 covers basic graph theory, develops an understanding of different pathfinding algorithms, explains the reasons to chose a specific algorithm in our context and goes into detail about the programmatic implementation.

Chapter 2

Movement

For our robot to be able to show the results of path finding, it needed to be able to move. We decided to move only along a simple 2D grid-like structure, therefore wheels were the easiest solution. The following chapter aims to provide information about components and theory needed for building the movement system of the robot.

2.1 Stepper Motors

A stepper motor is a motor that moves one step at a time, with its step defined by a step angle.

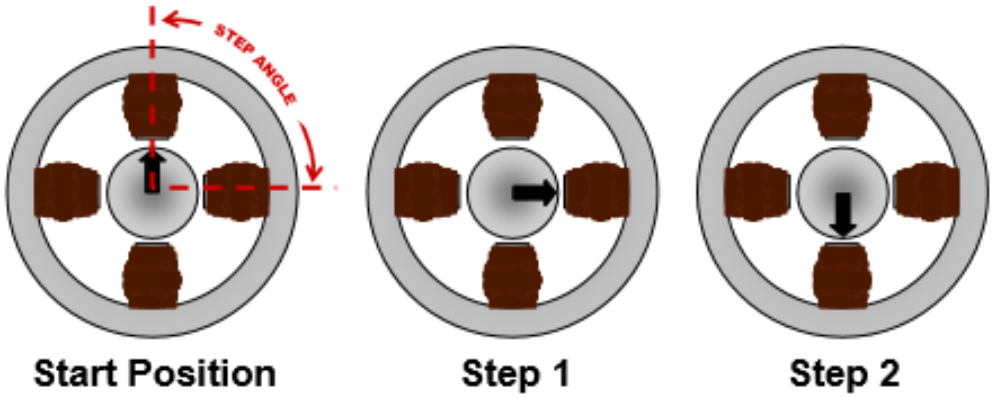


Figure 2.1: Step Angle
[4]

Figure 2.1 represents a stepper motor that requires 4 steps to complete a 360° rotation. This determines the step angle to be 90°.

The main components of a stepper motor are represented in Figure 2.2, and they consist of stators, windings(phases), and rotor. Attached to the output axle is the rotor, depending on the type of motor it can be magnetized.

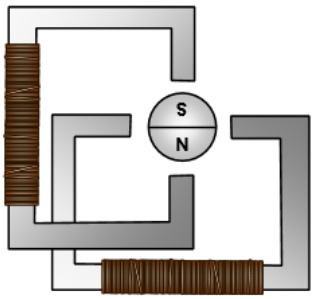
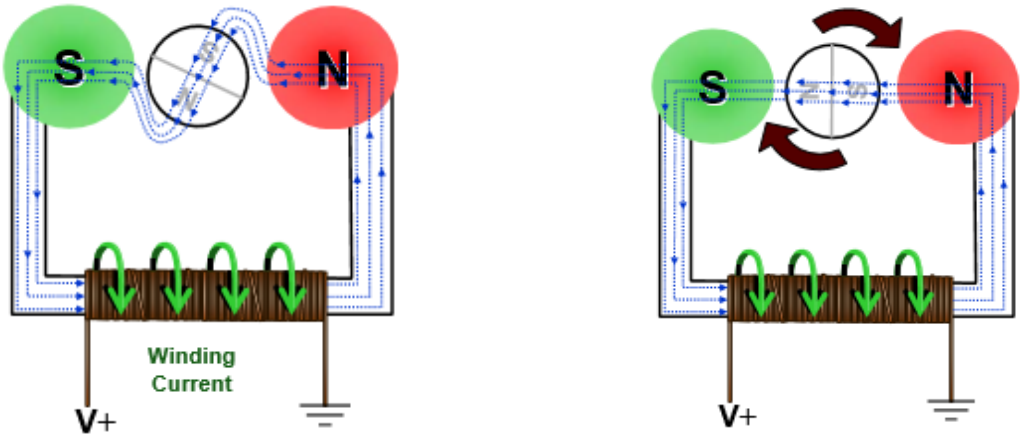


Figure 2.2: Main Components
[4]

By applying a voltage across one of the windings, current will start flowing through it. Using the right-hand rule, the direction of the magnetic flux can be determined. The flux will want to travel through the path that has the least resistance. This determines the rotor to change its position to minimize resistance. This is shown in Figure 2.3.



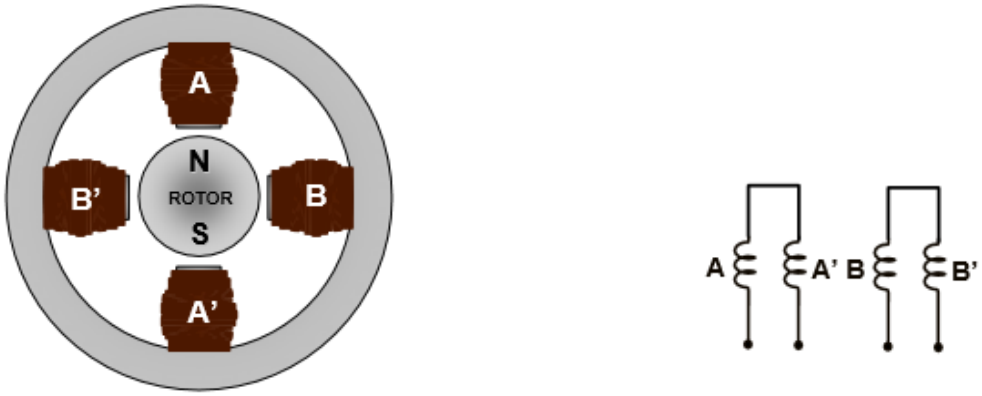
(a) High Resistance **(b)** Low Resistance

Figure 2.3: Direction of Magnetic Flux
[4]

2.1.1 Types of Stepper Motors

Permanent Magnet Motor

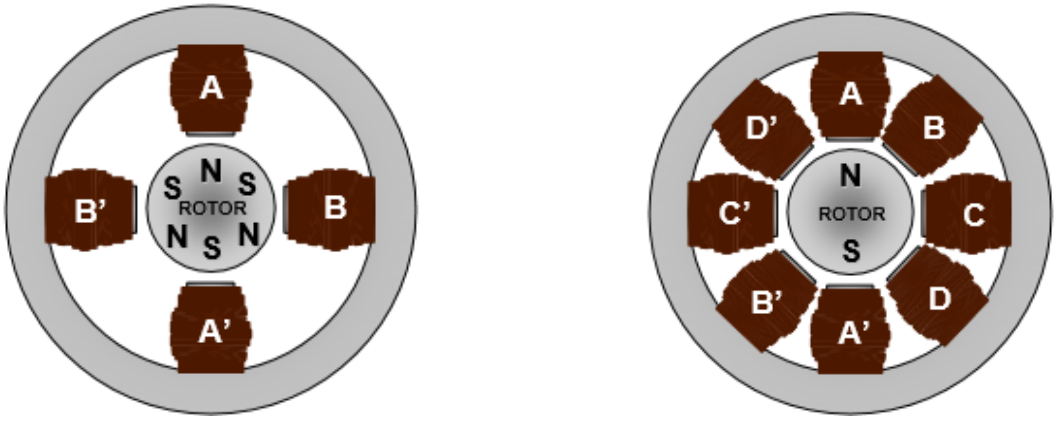
This type of stepper motor has a magnetized rotor. Each winding will be subdivided into two, to better understand how to motor functions. Figure 2.4 represents the windings, and how they are distributed inside a stepper motor.



(a) Rotor **(b)** Winding

Figure 2.4: Basic Structure of a Motor
[4]

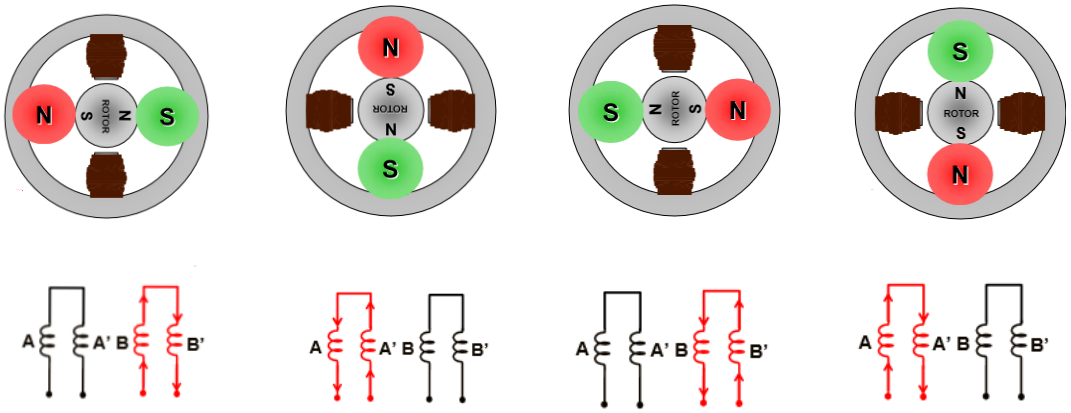
The resolution of the motor can be improved in two ways, either by increasing the number of pole pairs in the rotor itself, or by increasing the number of phases as shown in Figure 2.5.



(a) Increased Pole Pairs **(b)** Increased Number of Winding

Figure 2.5: Increased resolution
[4]

To rotate the motor, simply apply a voltage across the windings in a sequence. A full rotation is shown in Figure 2.6, with the corresponding phases energized.



(a) 1st Step **(b)** 2nd Step **(c)** 3rd Step **(d)** 4th Step

Figure 2.6: Stepping a Permanent Magnet Motor
[4]

Variable Reluctance Motor

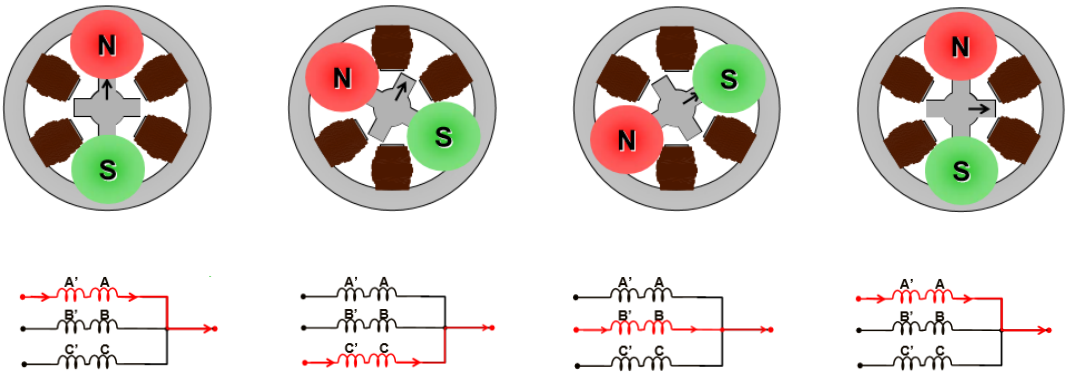
This type of motor uses a rotor that is not magnetized, and has a number of teeth as seen in Figure 2.7. The windings are configured differently, as depicted in 2.7(b), all having a common voltage source but separate ground connections. They usually have 3 or 5 windings. Greater precision can be achieved by adding more teeth to the rotor.

(a) Non Magnetized Rotor **(b)** Windings

Figure 2.7: Variable Reluctance Motor Components
[4]

To spin the motor, each winding is energized one at a time, and the rotor rotates to minimize reluctance as explained before. Some of the differences, between this type of stepper motor and the permanent magnet motor, are that, in order to spin the motor in a direction, the windings have to be energized in a reverse sequence as opposed to the direction of the spin, as depicted in Figure 2.8.

In addition, variable reluctance motors have twice the precision of permanent magnet motors with the same amount of windings.



(a) 1st Step **(b)** 2nd Step **(c)** 3rd Step **(d)** 4th Step

Figure 2.8: Stepping Variable Reluctance Motor [4]

Hybrid Stepper Motor

Hybrid stepper motors borrow characteristics from both previously mentioned types.

Figure 2.9 shows the two the main components of the hybrid stepper motor. On the left side, the stator can be seen consisting of 8 poles. On the right side the rotor. The rotor consists of two sets of teeth, corresponding to the two poles, north and south.

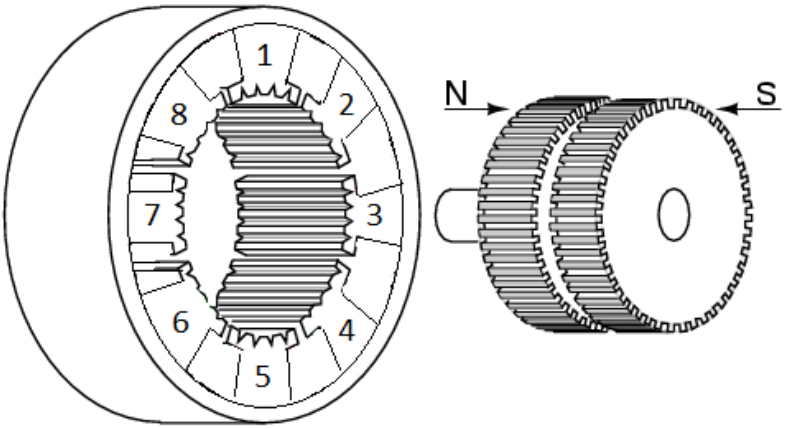


Figure 2.9: Stator and Rotor
[5]

It is important to notice two additional things. The first, is that the teeth on the rotor are not aligned but are interleaved. The second, is the placement of the stator teeth in respect to those of the rotor. Both can be observed in Figure 2.10.

(a) Interleaved Teeth

(b) Stepper Motor Inside

Figure 2.10: Hybrid Stepper Motor
[6]

In Figure 2.10, the orange windings are completely aligned with the teeth of the rotor, while the others are half aligned. This results in higher precision and higher torque offered by the hybrid stepper motor, depending on the stepping method used.

Figures 2.11, 2.12, 2.13, 2.14 represent the way this motor operates.

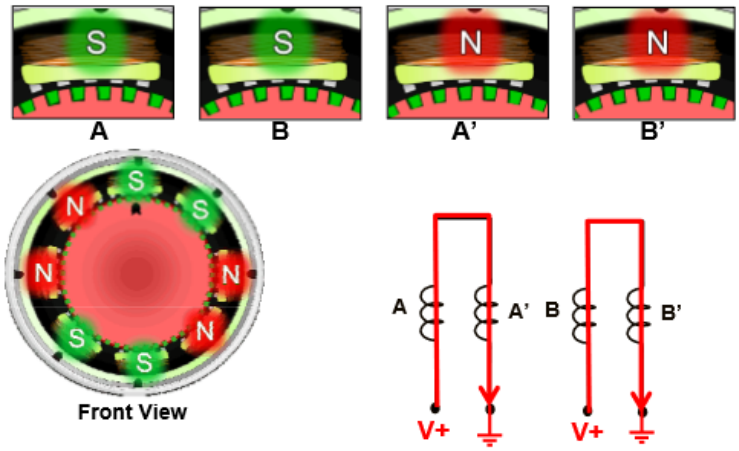


Figure 2.11: First Step
[4]

By applying a voltage to both windings, the current flow can be controlled, thereby controlling the polarity of each stator pole, thus controlling the direction of the motor. Notice that, initially, poles A and A' are completely aligned, and poles B and B' are half aligned.

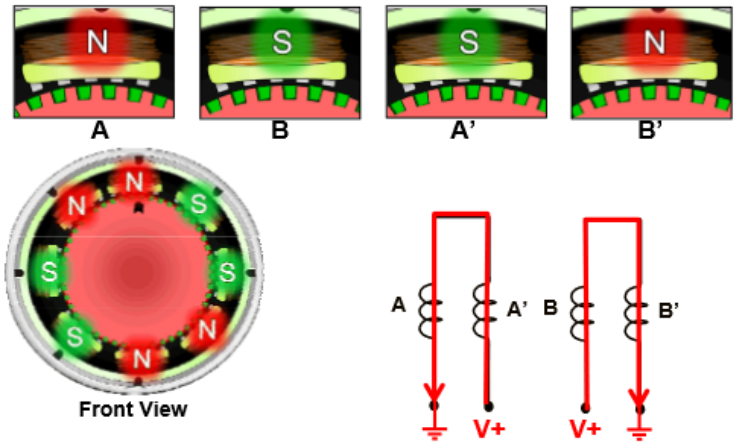


Figure 2.12: Second Step
[4]

Next step involves changing the direction of the current in winding A by applying a voltage at the other end of the winding. Even though only the current in winding A has been changed, all stator poles are aligned differently. Poles A and A' are now half aligned, and poles B and B' are completely aligned.

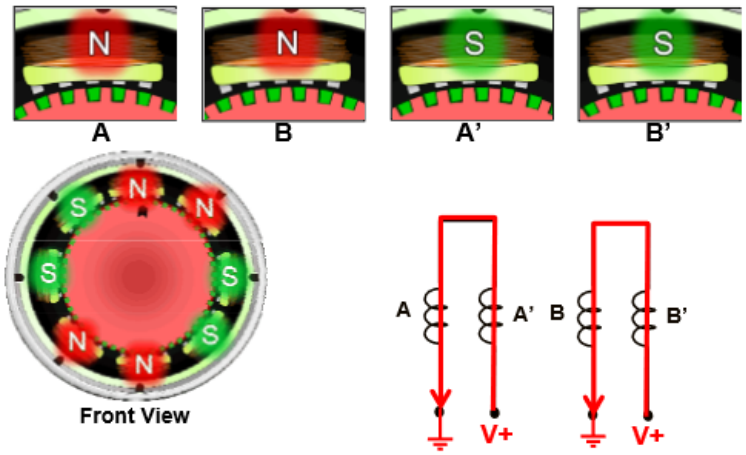


Figure 2.13: Third Step
[4]

Now, changing the direction of the current in winding B, changes the polarity of the stator poles B and B', again, determining a change in the alignment of all stator poles. A and A' are now completely aligned, and stator poles B and B' are half aligned. The positions of the stator poles now correspond to those of the first step.

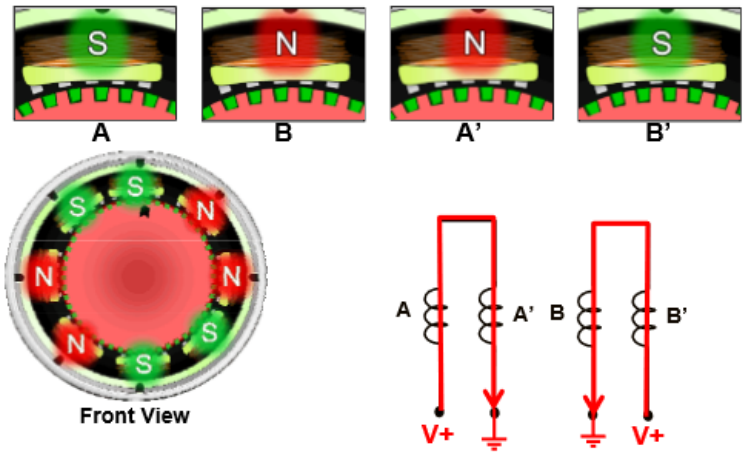


Figure 2.14: Forth Step
[4]

Finally, again changing the direction of the current in winding A, determines the rotor to move another step. Notice the alignment of the stator poles. A and A′ are half aligned, while B and B′ are fully aligned. By changing the direction of the current in winding B, the motor arrives in the initial state, thus repeating the sequence.

2.1.2 Unipolar And Bipolar Stepper Motors

Another classification of stepper motors, is depending on the way the windings are configured. Even though, nowadays, almost every stepper is both. Meaning that unipolar and bipolar, are rather modes in which the stepper motor can be driven. Exception being, stepper motors which have only four wires coming out of them, corresponding to bipolar stepper motors.

Figure 2.15 below represent the configuration of the windings in both unipolar and bipolar stepper motors.

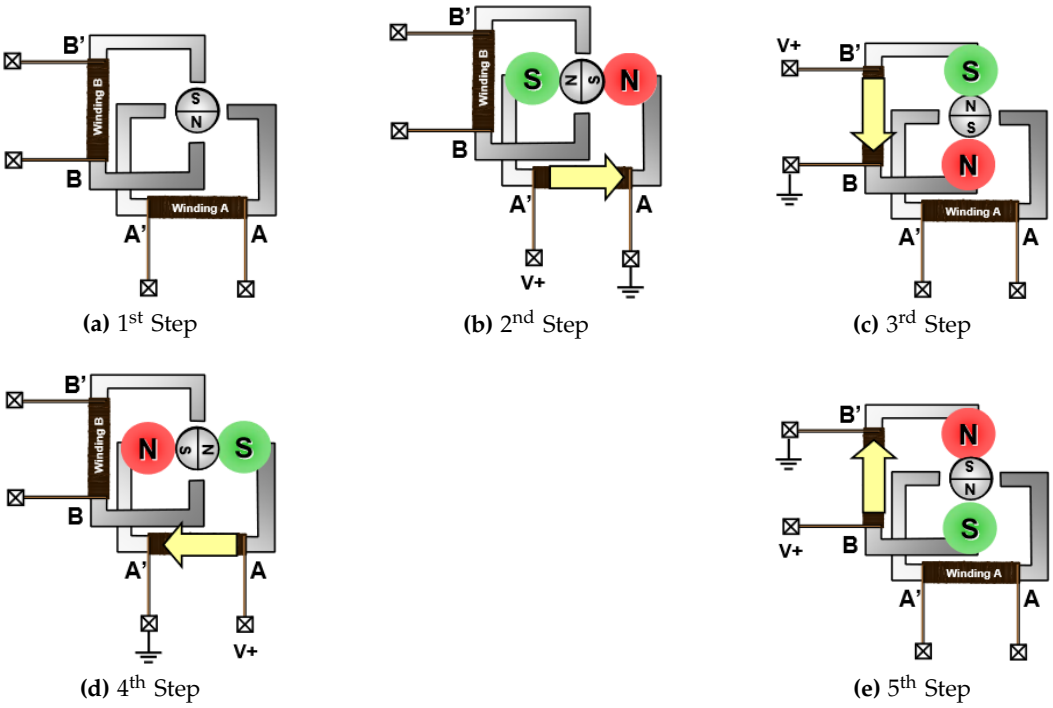
(a) Bipolar **(b)** Unipolar

Figure 2.15: Winding Configuration
[4]

Bipolar stepper motors have one winding per phase. To energize the first phase, voltage needs to be applied to lead 1, and lead 2 needs to be connected to ground. Stepping the motor, involves energizing one phase, then the second, then the first phase again with reverse polarity, meaning the voltage source and the ground must be switched between each other(lead 1 – ground, lead 2 – voltage source). Afterwards, the second winding is energized with reverse polarity. This makes driving them more difficult. We use driver boards to make the task easier.

Unipolar stepper motors allow current flow in only one direction through the winding, unlike bipolar stepper motors. Because of that, a center wire has been added to each winding corresponding to leads 3 and 6. All the other leads are connected to ground. Outside the motor, each lead connected to ground is connected to a transistor first. To step the motor, apply voltage to the corresponding transistor to connect the lead to ground and allow current flow through that winding.

Note that the bipolar configuration as shown in Figure 2.16 allows the current to flow in both directions, but the voltage and ground continuously switch positions. This makes bipolar stepper motors a bit more complicated to drive, but as previously stated, motor driver boards simplify the task.



(a) 1st Step **(b)** 2nd Step **(c)** 3rd Step

(d) 4th Step **(e)** 5th Step

Figure 2.16: Bipolar Motor Spinning
[4]

2.1.3 Motor Driver Boards

We used motor driver boards in order to drive the motors. They provide a simple interface between the microcontroller and the motors, and make for a better alternative than directly driving the motors from the microcontroller.

Figure 2.17: Driver Board
[7]

2.2 Wheels

The robot should be able to move in eight directions from every position. By using traditional wheels, the robot would need to be able to steer to the desired direction, thus changing orientation. This would have been a difficult task raising a number of problems. Our solution is to use omni-wheels instead. A standard wheel and an omni-wheel are shown in Figure 2.18.

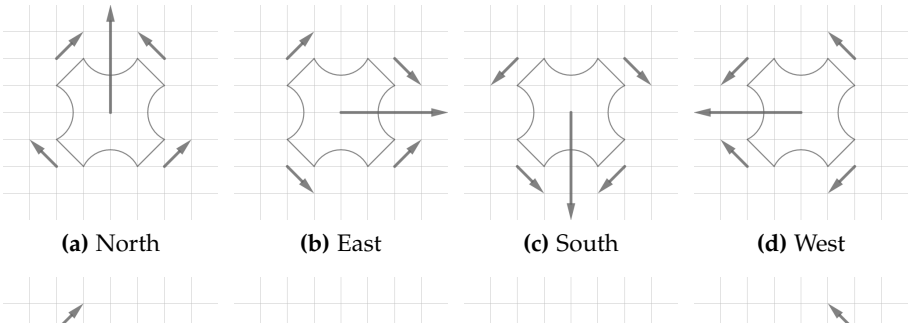
The key difference between omni-wheels and traditional wheels is their contact area. For omni-wheels it consists of smaller wheels that are able to move freely sideways, thus not generating any friction.

(a) Standard Wheel [8]

(b) Omni Wheel [9]

Figure 2.18: Wheels

By mounting the wheels in pairs, with the shafts crossing at a 90° angle, we are able to move the robot in any direction without needing to change the orientation of the robot. This is achieved through rotating the pairs as shown in figure 2.19.



(a) North **(b)** East **(c)** South **(d)** West

(e) North-East **(f)** South-East **(g)** South-West **(h)** North-West

Figure 2.19: Forces from Multiple Wheels Added Together

It can also be observed that no two opposite motors spin in different directions, because this would lead to a rotation, which is undesired for us. This has also made our task of programming the motors more simple.

2.3 Direction Control

To decide the direction of the robot, we had to control which wheels turn what number of steps.

One option would have been to control each motor individually. This required four pins for each motor to step the motors, and precise timing between the four motors. Imprecise timing could introduce unintended rotation.

We decided to build our own circuit using tri-state buffers instead. The circuit will be explained after a short explanation of tri-state buffers.

2.3.1 Tri-State Buffer

To achieve the desired movement using as few pins as possible, we decided to use Tri-State buffers. Fewer pins make it easier to port this part of the robot to a smaller μC with fewer pins for a final product.

Tri-State buffers provide the possibility of disconnecting parts of the circuit, when not needed. This allowed us to manipulate the input to the motors dynamically.[10]

A Tri-State buffer can be thought of as a switch. Figure 2.20 better illustrates that concept.

(a) Disabled **(b)** Enabled

Figure 2.20: Tri-State Buffer Switch Analogy
[10]

When the buffer is enabled, its output corresponds to its input, either 0 or 1, "High" or "Low". However when the buffer is a in its third state, its output is disabled, opening the circuit between the buffer and the next component. That does not mean its output corresponds to a logic “Low", but instead it is in a state of high impedance in which the output is disconnected from the rest of the circuit. [10]

2.3.2 Control Circuit

Initially, we have thought of a movement system, which required 16 pins to accommodate moving in all directions. In this system we would have controlled each motor individually, requiring 4 pins for each motor. With this method, it would have been necessary to have precise timing between the four stepping sequences.

Having a limited number of pins available, has forced us to think of the movement system very thoroughly. The reason for this was that we wanted to test the movement system on the MSP430 μ-C, which only has 10 I/O pins. [11]

Figure 2.21 shows a schematic of the motor control circuit.

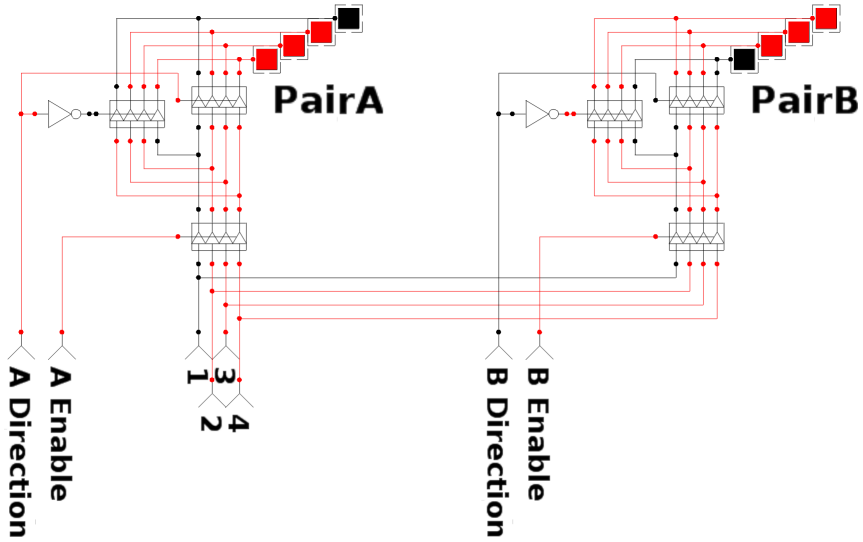


Figure 2.21: Motor Control Circuit

The four wires coming from the microcontroller, labeled '1', '2', '3' and '4' correspond to the stepping sequence. They are connected to two Tri-State buffers corresponding to each pair of motors. The location of the motors is shown in Figure 2.22.

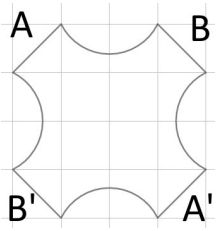


Figure 2.22: Motor Pairs

The first two Tri-State buffers switch the pair on or off, dependent on it being necessary for moving in a specific direction. The input pins, labeled 'A Enable' and 'B Enable' go to the two Tri-State buffers connected to each motor pair.

Afterwards, each enabling buffer is connected to two buffers that decide the direction, one active-high and one active-low. Those two buffers can be controlled by one bit, because of their opposite enable voltage.

For example, when moving North-East, only 'Pair A' is needed. 'A Enable' will have a 'High' signal while 'B Enable' will have a 'Low' signal. Next, applying a 'Low' signal to 'A Direction', reverses the wiring of the motors determining the robot move North-East.

Note that, when moving in a specific direction, the motors that form a pair, have to spin in opposite directions. One spins clockwise, while the other spins counter-clockwise. This was achieved by wiring one motor in reverse as shown in Figure 2.23.

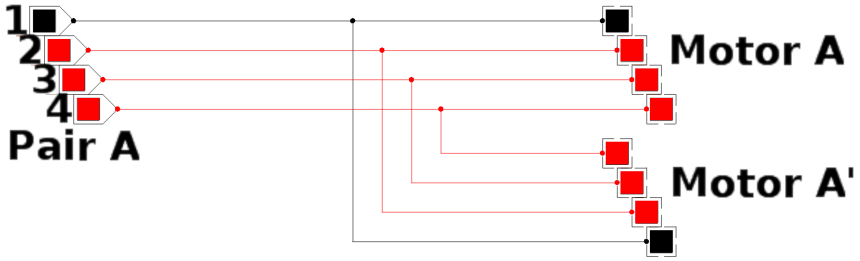


Figure 2.23: Motor Wiring

It is important to specify, that the robot moves in said directions using the same programming only by maintaining the same orientation as in Figure 2.22.

Tables 2.1 and 2.2 show the configurations required for moving in any of the eight directions.

Table 2.1: Directions

Direction	Pair A	Pair B
North	forward	forward
East	forward	backward
South	backward	backward
West	backward	forward
North-East	forward	off
South-East	off	backward
South-West	backward	off
North-West	off	forward

Table 2.2: Direction Signals

Direction	A_{Enable}	$A_{Direction}$	B_{Enable}	$B_{Direction}$
North	1	0	1	0
East	1	0	1	1
South	1	1	1	1
West	1	1	1	0
North-East	1	0	0	x
South-East	0	x	1	0
South-West	1	1	0	x
North-West	0	x	1	1

2.4 Implementation

We have used an MSP430 microcontroller to implement and test the movement system. Parts of the code will be explained in the following pages. The complete code can be read in Appendix B.

Individual functions are called whenever the robot is required to move in one of the eight directions: N, E, S, W, NE, SE, SW, NW. Listing 2.1 and 2.2 show two functions, representing one straight move and one diagonal move. The other behave similarly, the main difference laying in the pairs of wheels used, and their direction.

Listing 2.1: Moving North West

```
74 void moveNorthWest(void){
75   P1OUT |= BIT5                          //enable pair of wheels
76   P1OUT |= BIT6;                         //choose direction
77   stepping();                            //call stepping function
78   P1OUT &= ~BIT5;                        //disable pair of wheels
79 }
```

The function enables the pair of wheels needed for moving North West and selects the rotation of the wheels. Changing `P1OUT |= BIT6` to `P1OUT &= ~BIT6`, would determine the robot to move South East instead. The stepping function is called afterwards, determining the robot to move. Lastly, the pair of wheels is deactivated.

Listing 2.2: Moving North

```
102 void moveNorth(void){
103   P1OUT |= BIT5;                         //enable first pair of wheels
104   P1OUT |= BIT6;                         //choose direction
105   P1OUT |= BIT7;                         //enable second pair of wheels
106   P2OUT |= BIT7;                         //choose direction
107   stepping();                            //call stepping funcion
108   P1OUT &= ~(BIT5 + BIT7);               //disable both pairs of wheels
109 }
```

The function from Listing 2.2 combines both `moveNorthEast` and `moveNorthWest` functions. It enables both pairs of wheels and determines their direction. Setting the direction bits to '0' instead of '1' would determine the robot to move South. Similarly, the stepping function is called and the wheels are deactivated afterwards.

The `move` function is called, whenever the robot should move one direction. The `move` functions further decides which way the robot should move based on the hex value given to the function. The hex values for the directions are predefined. Listing 2.3 shows the `move` function.

Listing 2.3: Move Function

```
138 void move(unsigned int x){
139   switch(x){
140   case 0x01: moveNorth();       break;
141   case 0x02: moveEast();        break;
142   case 0x04: moveSouth();       break;
143   case 0x08: moveWest();        break;
144   case 0x10: moveNorthEast();   break;
145   case 0x20: moveSouthEast();   break;
146   case 0x40: moveSouthWest();   break;
147   case 0x80: moveNorthWest();   break;
148   }
149 }
```

Chapter 3

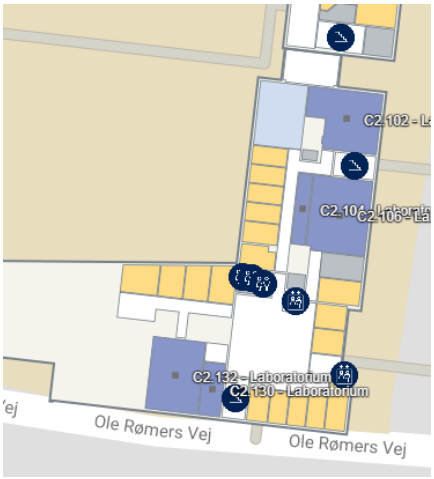
Map Handling

The rescue robot should be able follow a given path from start to finish, based on a predefined map given as input. A map provides useful information about whether areas of the map are accessible or not. Map data can be loaded by the robot prior to its physical presence at a location. Once the robot is at the starting point, it has to rely on its sensors for updated information about the surroundings.

The map itself is a crucial part, that converted into to a graph is used by the path-finding as explained in Chapter 5. Hence a structured way of storing the required map data for different maps was designed. The primary goal was to make it readable by the microcontroller, but also still allow easy user input.

3.1 Map Requirements

Maps can be found in a lot of different styles, varying in how they represent specific information. Those styles often depend on the purpose of the map. Figure 3.1a shows a map for casual orientation purposes, while 3.1b shows a standardized evacuation plan.



(a) Section of AAU Esbjerg

(b) School layout example

Figure 3.1: Examples of different maps

Maps are often very visual, providing a lot of detailed information to the reader. The way the information is represented differently, makes it very hard for a computer to interpret. A map must provide necessary information, in a way that can be interpreted by the micro controller. For this project we decided that a simplified map, would be sufficient.

Table 3.1 shows which data the map should include, as well as some areas that have been delimited from.

Table 3.1: Map data

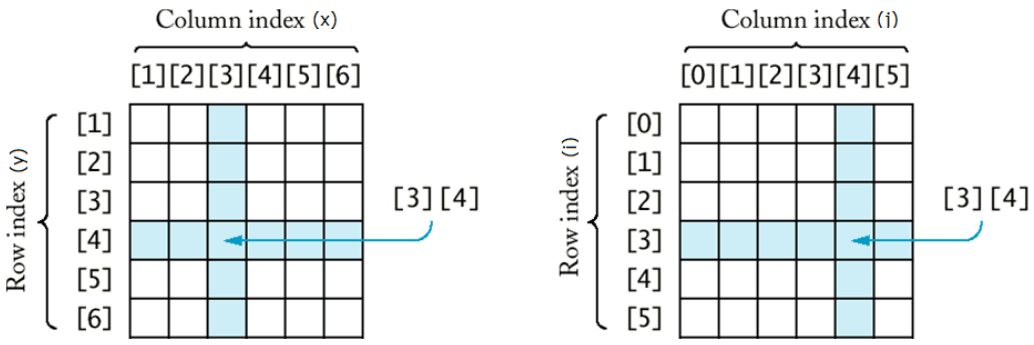
Data to be included	Data to delimit from
Map dimensions	Differences in height (levels, stairs etc.)
Start position	Door openings
Finish position	Ground surface (slipping, traction)
Walls	Objects

3.2 Map Coordinates

During the theoretical development we often used hand-drawn 2D maps with grids, equivalent to the map depicted in Figure 3.2a. The map can have a certain size and allows for an object such as a wall to have a location on the grid. A specific part of the map can easily be referred to by its unique coordinate in the x and y dimensions.

For converting and storing analog maps into a usable digital representation with the same properties, we chose to use a 2D array as data structure. A 2D array can be thought of as a matrix, where a grid of numbers can be arranged in rows and columns. 2D arrays are very similar to matrices, and differs in how elements are indexed.

The result of the different indexing methods can be seen by comparing Figure 3.2a to 3.2b. Given the same index values, the cell referred to would be different, as seen in Figure 3.2.



(a) 2D grid map **(b)** 2D array

Figure 3.2: Difference in indexing for (3,4) in a 2D grid map and a 2D array

Each cell in the map represents a map segment with its own set of coordinates. We chose to start map coordinates at zero, and inf needed rows and columns could still be switched, which is very similar to a matrix transpose.

In the program dynamic 2D arrays were used, which allows to easily read or set the value of any given map segment, as shown in Listing 3.1. The same approach was later used for handling node maps, which is further explained in Chapter 5. Structs are declared in `defs.h` which can be found in Appendix D.

Listing 3.1: Example of reading or setting a value for coordinate (3,4) in the 2D map array, using structs and pointers.

```
1 robot->map.segments[3][4] = 0x0F;          // Setting a value
2
3 char map_segment = robot->map.segments[3][4]; // Reading a value
```

3.3 Map Design

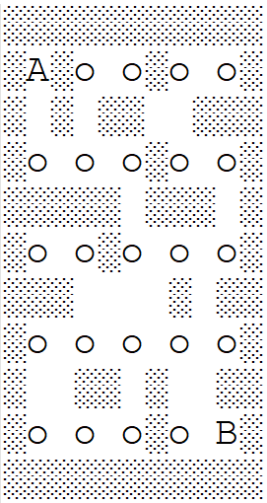
The grid-based map is made up of simple plain-text ASCII characters. This makes it fairly simple and easy-to-understand, and maps can easily be created or changed by a user.

An example of a map with 5x5 nodes can be seen in Figure 3.3a, where # being walls, A being the start, B being the finish, o being nodes, and the whitespaces being open spaces. The same map can be seen using UTF8 encoding in Figure 3.3b. UTF8 has a more characters to choose from, which makes it easier to read for humans , while the plain-text version is easier to read for computers.

Nodes represent positions on the map where the robot can move between, more on nodes in Section 5.1. The robot can move in any of the eight directions unless it is blocked by a wall. The directions consists of four straight directions, N,E,S,W and four diagonal directions NE,SE,SW,NW.

```
###########
#A#o o#o o#
# # ##  ###
#o o o#o o#
##### ### #
#o o#o o o#
###    # ##
#o o o o o#
#  ## #  ##
#o o o#o B#
###########
```

(a) ASCII



(b) UTF8

Figure 3.3: Example of a map with 5x5 nodes, having 11 characters per line and 11 lines.

We found that making the map using UTF8 would require a bit more work, since text editors sometimes add characters to the beginning of the file. This is known as the byte-order-mark (BOM) which indicates the file uses UTF8 encoding. It also uses variable bit-length for characters, between 1-6 bytes. This makes reading and storing the map to a file more complicated, which is why we chose to use plain text ASCII.

3.4 Reading Map from a File

Functions were made for loading maps from a text-file, as well as saving an updated map to a text-file.

Reading and writing maps on a computer locally, was very useful for developing and testing the map handling subsystem. Implementing this functionality on a microprocessor, would require it to have a file system installed. We chose not to implement this functionality, due to the restricted time frame of this project. As an alternative, maps can be stored directly in memory as part of the programming code. This solution is not as user-friendly, and updates to the map will be lost from the memory when the microcontroller is turned off.

In the C programming language the size of an array must be declared at compile time. The array for the map has to be large enough to hold all the map data. Unfortunately the map size and data remains unknown until a map is loaded, which happens during runtime.

One way of dealing with this, is by using a 2D array with a fixed size, large enough to store maps of a convenient size. We chose to use a dynamic 2D array instead. This more advanced approach gave us more experience, by using dynamic memory allocation and pointers to implement this solution in the application.

The required size of the dynamic 2D array can be calculated by counting the the rows and columns of the map, which is equivalent to the lines and characters. A map with 5x5 nodes will have 11 characters in each line and a total of 11 lines, as illustrated in Figure 3.3. Listing 3.2 shows how a map is read from a text-file, and how the lines and characters are counted.

Listing 3.2: Reading map text-file and analysing the map size in `map.c`

```
30 ///Reads user input file, stores map data in struct
31 void map_load(Robot *robot) {
32   // Open file in read mode
33   FILE *myfile = fopen(MAP_FILENAME, "r");
34
35   // Count number of lines and characters
36   int rows = 1;        // Counts newlines
37   int cols = 0;        // Counts characters
38   int c;               // Holds each character as it is read from file
39
40   while ((c = fgetc(myfile)) != EOF) { // Read file characterwise
41     if (c == '\n') {  // end of the line (linebreak)
42       rows++;         // Count the line
43       cols = 0;       // Reset character counter
44     } else cols++;    // Count characters
45   }
46   rewind(myfile);     // reset file pointer
```

To store the map, an 11x11 2D array is required. Now that the required array size is known, it can be allocated in the memory as shown in Listing 3.3.

Listing 3.3: Allocation of 2D array in `map.c`

```
48   unsigned char **array; // Pointer to array
49   array = malloc(rows * sizeof(char*));
50   for (int i=0; i<rows; i++) array[i] = calloc(cols, sizeof(char));
51   // Store the pointer to the 2D array in map struct
52   robot->map.segments = array;
53   // Save map size (rows and cols) in struct
54   robot->map.size.x = cols;
55   robot->map.size.y = rows;
```

Once allocated, the address of 2D array is stored in the `robot` struct as a pointer. The size of the map is stored in the `robot` struct as well, for future reference.

At this point the 2D map array is filled with the actual map data from file. Listing 3.4 shows how this is done using two for loops. The robots start and finish location is read from the map as well, and converted into coordinates for a starting and finish node.

Listing 3.4: Storing values in the 2D array in `map.c`

```
56  // Fill map array with map data from file
57  for (int i = 0; i < rows; i++) {
58    for (int j = 0; j < cols; j++) {
59      c = fgetc(myfile);              // Read next character from file
60      robot->map.segments[i][j] = c;// Store each character in array
61      if (c == 'A') {                 // found starting position
62        robot->map.start.x = (j-1)/2;
63        robot->map.start.y = (i-1)/2;
64        printf("[INFO]\tStart position:  [%2d][%2d]\n",
65              robot->map.start.x, robot->map.start.y);
66      }
67      if (c == 'B') {                 // found finish position
68        robot->map.finish.x = (j-1)/2;
69        robot->map.finish.y = (i-1)/2;
70        printf("[INFO]\tFinish position: [%2d][%2d]\n",
71              robot->map.finish.x, robot->map.finish.y);
72      }
73    }
74    fgetc(myfile);  // don't save newline
75  }
76  fclose(myfile);   // close file
```

Listing 3.5 shows how the coordinates for the start and finish nodes are stored in the `robot` struct for future reference.

Listing 3.5: Storing start and finish position in the `Robot` struct in `map.c`

```
85  // Set robot current position to map start position
86  robot->pos.x = robot->map.start.x;
87  robot->pos.y = robot->map.start.y;
```

3.5 Building a Node Map

Path finding algorithms operate with a special type of map called a graph. A graph consists of only nodes and edges, as explained more in detail in Section 5.1. The function `node_map_load` converts the ordinary map made of ASCII characters, into a map that only consists of nodes. Each `Nodes` struct has pointers to the eight neighbour `Nodes` structs. This is similar to the principle of having a pointer in a linked list, pointing to the next element of the same struct type.

Like with linked lists, structs can be used as a data structure to store data of different types. Listing 3.6 shows the data structure created for the 'Nodes struct' to hold all the necessary data for the individual nodes.

Listing 3.6: data structure of `Nodes` struct in `defs.h`

```
24 typedef struct Node {
25   Point position;                // Nodes own x,y position on node map
26   struct Node *n,*e,*s,*w;       // Pointers to neighbors straight
27   struct Node *nw,*ne,*se,*sw;   // Pointers to neighbors diagonal
28   struct Node *parent;           // Pointer to parent node
29   unsigned char walls;           // Byte value for the 8 walls
30   int movecost;                  // Steps needed to get here
31 } Nodes;
```

Listing 3.7 shows how the dynamic 2D array is declared using malloc. It holds structs of type `Nodes`, one for each node in the map is allocated in memory.

Listing 3.7: Declaration of node map 2D array used for storing `Nodes` structs in `map.c`

```
90 /// loads a node map from the saved map
91 void node_map_load(Robot *robot) {
92   // calculate and store the amount of nodes
93   robot->map.nSize.x = (robot->map.size.x-1)/2;
94   robot->map.nSize.y = (robot->map.size.y-1)/2;
95
96   Nodes **array;  // Declare node array of correct size
97   array = malloc(robot->map.nSize.x * sizeof(Nodes*));
98   for(int i=0; i<robot->map.nSize.x; i++) {
99     array[i] = malloc(robot->map.nSize.y * sizeof(Nodes));
100  }
101  robot->map.node = array;  // store array in struct
```

In the end a pointer to the declared 2D array gets stored in the `Robot` struct. This allows for easy access to all the data for a given node, just by using the nodes position on the map as the array index values. An example of the usage can be seen in Listing 3.8.

Listing 3.8: Example of accessing a node in the node map stored in a 2D array

```
1 robot->map.node[i][j].[name of element in Nodes struct]
```

3.6 Encoding Walls in a Single-byte Value

Each node in the map potentially has up to eight neighbour nodes. The robot can move to any of the neighbour nodes, unless the path is blocked by a wall.

To represent the possible directions to move in, we decided to use the boolean values '0' and '1'. Here '0' would mean it is possible to move in that direction, while '1' means there is a wall. It is possible to store data about the available directions in 8 bits in total, equivalent to a single byte.

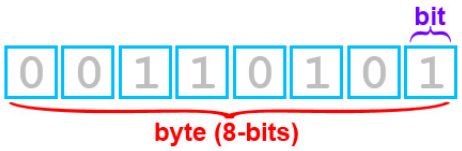


Figure 3.4: A byte where the eight bitflags represents, which of the eight possible directions that are free to move in.

3.7 Map Validation Using Sensors

There might be rescuing scenarios where parts of the map, or even the entire map, would be unknown. External factors such as an explosion in a building, could also change the surroundings dramatically, making the map data invalid.

After the robot has moved to the next node on the map, its current position is updated. At the robots current location, sensors scans the surrounding environment for walls in the eight possible move-directions. The feedback from the sensors is returned as a single-byte value, as described in Chapter 4.

The single-byte from the `scan` function gets compared to the single-byte in the map array, for the robots current location.

Each time the robot has made move, the surroundings at the current position is scanned and compared with the map. This code can be seen in Listing 3.9.

Listing 3.9: Scan surroundings at current position and compare with map segment in `map.c`

```
3  /// Scan surroundings at current position and compare with map segment
4  void map_check(Robot *robot) {
5    unsigned char scan_segment = scan();
6    unsigned char map_segment =
       robot->map.node[robot->pos.x][robot->pos.y].walls;
7    if(scan_segment != map_segment) {   //segment changed
8      map_update(robot, scan_segment);
9      map_save(robot);                  //save to file
10     path_calculate(robot);            //recalculate
11   }
12 }
```

Listing 3.10 shows that if changes to the surroundings where detected, the 2D map array gets updated with the new data.

Each of the eight directions around the node at the robots current position gets updated. The bitwise comparison operator '&' is used to check if the directions in the byte is flagged as '0' or '1', meaning a whitespace ' ' or a wall '#' should be stored.

Listing 3.10: Map gets updated with the byte returned by the scan function in `map.c`

```
126 /// updates the map file at the current position to the given value
127 void map_update(Robot *robot, char hex) {
128
129   // Convert node coordinate to map coordinate
130   int i = robot->pos.y*2+1;
131   int j = robot->pos.x*2+1;
132
133   // Update all 8 neighbors according to the hex wall value
134   robot->map.segments[i-1][j+0] = (hex & North) ? '#' : ' ';
135   robot->map.segments[i-0][j+1] = (hex & East)  ? '#' : ' ';
136   robot->map.segments[i+1][j-0] = (hex & South) ? '#' : ' ';
137   robot->map.segments[i-0][j-1] = (hex & West)  ? '#' : ' ';
138   robot->map.segments[i-1][j+1] = (hex & NorthEast) ? '#' : ' ';
139   robot->map.segments[i+1][j+1] = (hex & SouthEast) ? '#' : ' ';
140   robot->map.segments[i+1][j-1] = (hex & SouthWest) ? '#' : ' ';
141   robot->map.segments[i-1][j-1] = (hex & NorthWest) ? '#' : ' ';
142
143   // Map is now up to date, rebuild nodes based on the updated map
144   free(robot->map.node);  // Free current nodes
145   node_map_load(robot);   // Rebuild nodes
146 }
```

3.8 Writing Updated Map to a File

Sensors scan the surroundings for walls, at the robots current position. If changes are detected, the map gets updated as described in Section 3.7.

If the updated map is kept only in memory, it will be lost once the microcontroller is turned off, since this is not a persistent storage. To prevent this during testing, the map was saved in a text-file.

The individual map segments that consists of ASCII characters, gets written to the map text-file. Listing 3.10 shows how this is done by iterating through all the characters in 2D map array.

Listing 3.11: Map gets written to text-file using two for loops in `map.c`

```
14 /// Write map data to text file
15 void map_save(Robot *robot) {
16   // Open the file in write mode
17   FILE *myfile = fopen(MAP_FILENAME, "w");
18   // Write map segments to text file
19   for (int i = 0; i < robot->map.size.y; ++i) {   // loop rows
20     for (int j = 0; j < robot->map.size.x; ++j) { // loop cols
21       fprintf(myfile, "%c", robot->map.segments[i][j]);
22     }
23     if(i < robot->map.size.y - 1) {               //not last line
24       fprintf(myfile, "\n");                      //ad linebreak
25     }
26   }
27   fclose(myfile);                                 // Close file
28 }
```

Chapter 4

Scan

In order to be able to handle a dynamically changing environment, we decided to use IR distance sensors, to check if the map matches the real physical environment. Initially, our robot is supposed to find the shortest path from A to B, on a map.

In rescue situations the map provided does not necessarily match the actual environment. External factors could for example have collapsed walls, thus obstructing movement.

By using 8 IR sensors, we can detect changes in all directions the robot can move in. If a change is detected, this will be accounted for in the maphandling code, as explained in Chapter 3.

4.1 Sensor

For detecting any changes in the layout of the map, we have used analog IR sensors. Initially, our idea was to use digital sensors, which would have been easier to use, but none were available for testing and implementing. Instead, we have used analog sensors, which were available at the university. Having the chance to implement and test them earlier during the project, we decided to go with them. Figure 4.1 shows the sensors used.

Figure 4.1: Sharp IR GP2Y0A21
[12]

To be able to interpret the voltage received from the sensor, we referred to the datasheet of the sensor. Figure 4.2 shows the approximate voltage output of the sensors in respect to the distance measured. Voltage values below 10cm should be overlooked since the sensor is not able to detect distances closer than 10cm.

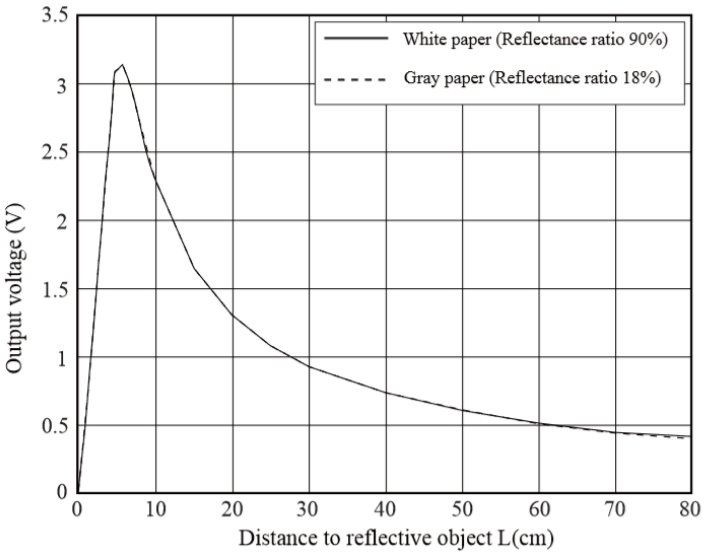


Figure 4.2: Voltage in respect to Distance
[13]

4.2 Scanning

The sensors are able do detect distances up to 80 cm. The grid size is defined in such manner that it should be able to fit the robot in one square. Additionally, the sensors should also be able to detect walls or corners in any of the eight directions. Figure 4.3 represents the robot and the range of the sensors relative to the grid size.

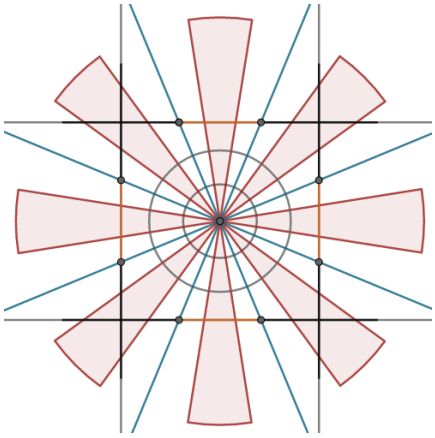


Figure 4.3: Robot and Sensor Range

Table 4.1: Color Explanation for Figure 4.3

Color	Meaning
Blue	Limits of Walls and Corners
Black	Corner
Orange	Wall
Red	Range of IR Sensors

The `scan` function detects changes, and returns a byte with the observed walls. If the byte returned matches the one stored in the map, the robot continues on its predefined path. Otherwise, the byte given by the `scan` function overwrites the one stored in the map and the robot recalculates the shortest path based on the updated map. An example of what a byte stored and a byte scanned might look like is given in Table 4.2.

Table 4.2: Map Byte and Scanned Byte

Directions	NW	SW	SE	NE	W	S	E	N	Byte
Map Byte	1	0	0	1	1	1	1	0	0x9E
Scan Byte	1	1	0	0	1	1	1	0	0xCE

4.3 Implementation

The IR sensor was tested on an Arduino Uno. Listing 4.1 shows the code to test the sensor, and can also be seen in Appendix C.

Listing 4.1: Sensor Tesing

```
1  #define sensor A0                          //define sensor pin
2  int distance;                              //variable to store distance
3  float volts;                               //variable to store reading
4                                             //from sensor
5  bool path;                                 //variable to store path
6  void setup(){
7    Serial.begin(9600);                      //start the serial port
8  }
9  void loop(){
10   volts = analogRead(sensor) * 0.0048828125;  //sensor * (supply
11                                               //voltage/
12                                               //adc resolution)
13   distance = 29.988 * pow(volts , -1.173);    //determined from
14                                               //datasheet graph
15   delay(500);                              //delay between readings
16   if (distance <= 50){                     //no path
17     path = 0;
18     Serial.print("Wall ");
19     Serial.print(distance);
20     Serial.print("cm ");
21     Serial.println(path);
22   }
23   else{                                    //path
24     path = 1;
25     Serial.print("No Wall ");
26     Serial.println(path);
27   }
28 }
```

The value read from the sensor is multiplied with the value given by dividing the supply voltage with the ADC resolution, as shown in line 11. Afterwards, line 13 determines the distance. Lastly, the value is checked and a boolean value is returned.

Figure 4.4 represents the values returned by the distance sensor.

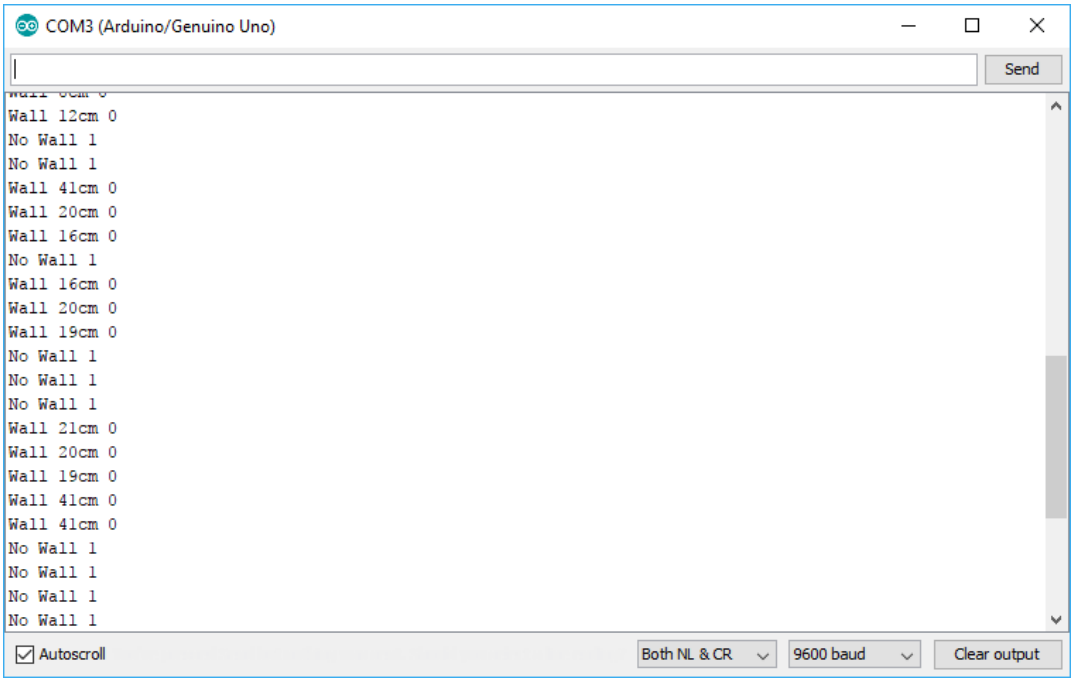


Figure 4.4: Returned Values

Chapter 5

Pathfinding

Pathfinding is generally the process of finding a path from a starting point *A* to a destination *B* on a map. Handling the map is explained in Chapter **??**.

There are different approaches to find the best path, and different ideas what the best path is.

In the case of rescue, where time is very crucial to success, the quickest path has to be considered best. [14]

In other applications 'best' could also mean shortest distance, least expensive (toll roads), most convenient or any number of other qualifiers.

Since our robot has approximately equal movement speed in all used directions, the shortest time path can be approximated as the shortest distance path.

We chose to start with implementing Dijkstra's shortest path algorithm, since it is fairly simple to understand and can be used as a baseline for better, more complex algorithms, like A*.

This chapter will explain the basics of different path-finding approaches, going more into detail on the ones we chose to implement for testing.

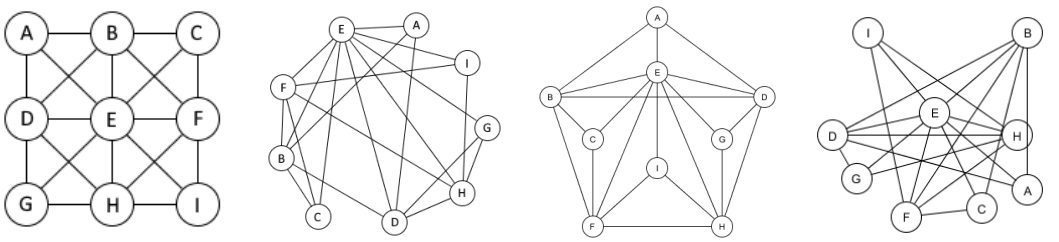
5.1 Graphs

The first step in most algorithms is to reduce the map to the necessary minimum. After this reduction, the map only consists of *nodes* and *edges*, organized in a *graph*.

An *edge* connects two *nodes* together and has a *distance*. In this integer is stored how much it costs to traverse along that *edge*, measured in the metric that should get optimized (in our case distance and approximate time).

A *node* has a *name*, a *cost to reach* and a reference to another *node parent*. The name is used as an identifier, *cost to reach* sums the travelling costs to get here on the currently shortest path from the start. *Parent* refers to what *node* is previous in that path.

There are two special *nodes*, namely the starting *node* and the finish *node*.



(a) Grid Graph **(b)** Circular Graph **(c)** Tree Graph **(d)** Unstructured Graph

Figure 5.1: Different Representations of the same Graph

Such a graph can be represented in any way, as long as none of the described characteristics change. Figure 5.1 shows four equivalent representations of the same graph. We chose to omit any numbers for simplicity.

Since our prototype is running on a grid-like map, the graph shown in Figure 5.1a is our preferred representation, since it is the easiest to relate to the real world for a human. For the algorithm however, it doesn't matter.

5.2 Brute-Force

Brute-force is generally an algorithm, that only relies on computational power, instead of clever design. For path-finding that would mean looking at all possible paths, and evaluating which one is the shortest. Brute-force algorithms can be implemented as a depth first search (DFS), or breadth first search (BFS).

5.3 Flood Fill

Flood fill is looking at all neighbour *nodes* from the start, and looking at all their neighbours. This process then gets repeated until the finish *node* is reached. Because the algorithm expands first in breadth, this is a BFS-algorithm.

The name comes from visualising the algorithm, which looks fairly similar to a liquid being spilled on a map. [15]

5.4 Dijkstra

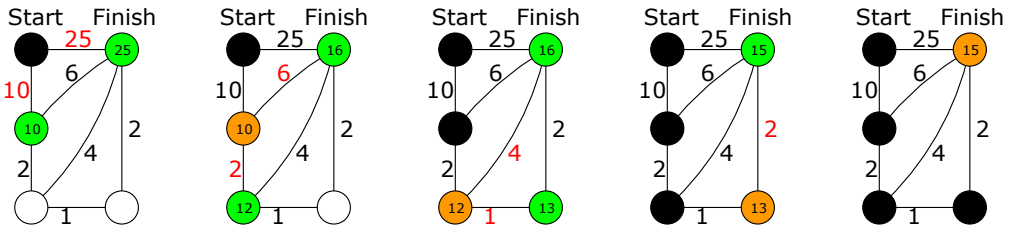
Dijkstra's algorithm is a small improvement on the flood-fill algorithm explained earlier. It takes into account the *distances* between two *nodes*, when deciding which

node to look at next. Thus prioritising the easier to reach *nodes*, when going to the next iteration.

This is done by storing all *nodes* in a priority queue, where they are sorted by their *cost to reach*, in ascending order. Our implementation of the priority queue can be read in Appendix J.

The *cost to reach* gets calculated iteratively, by adding the *cost to reach* of the current *node* together with *distance* to its neighbour. If that value is smaller than the *cost to reach* currently stored in that neighbour, the old value gets overwritten. This process is shown stepwise in Figures 5.2a through 5.2e. Those figures are also available in Appendix A.
Observe how the value for the finish *node* changes in almost every step, until the finish *node* is the current *node*.

Every time the algorithm needs a new *node* to evaluate its neighbours, it takes the first element from that list. [16]



(a) 1. Step **(b)** 2. Step **(c)** 3. Step **(d)** 4. Step **(e)** 5. Step

Figure 5.2: Dijkstra's algorithm on a simple map

Table 5.1: Colour guide for Figure 5.2

Colour	Function	
	Nodes	Edges
Red		Used in current evaluation
Orange	Current Node	
Green	Evaluated Neighbour	
White	Not active yet	
Black	Shortest Path already found	Not used in current step

This approach has a huge benefit for maps, where *distances* between *nodes* vary widely. In our case *distances* are one of two possibilities, either 1 or $\sqrt{2}$. Thus making this effectively one implementation of a flood-fill search, with the benefit, that only one addition needs to be done to implement A*, which gets explained in Section 5.5.

5.5 A*

A* uses Dijkstra's algorithm as a baseline, but adds one more step to the calculation. Just like in Dijkstra, the *cost to reach* gets calculated iteratively, with the same iteration method. But in A* there is also another value added to that sum, this value is often called *heuristic*. It is used to point the algorithm towards the finish node, and is often the *Euclidean Distance* distance from each node to the finish. [17]

The *Euclidean Distance*, is the distance as the crow flies. It can be calculated with help of the Pythagorean Theorem as shown in Example 5.1.

Example 5.1 (Pythagorean Theorem used for Euclidean Distance)

$$\sqrt{|A_X - B_X|^2 + |A_Y - B_Y|^2} \tag{5.1}$$

5.6 Pathfinding on a Grid

Pathfinding on a grid is slightly different to pathfinding on a regular map, because all *nodes* tend to have the same amount of neighbours, and all *edges* have the same or similar *distances*.

In our case *distances* vary between 1 and $\sqrt{2}$, the former in the case of vertical or horizontal movement, the latter for diagonal movement. We decided to round the diagonal *distances* to $14 \simeq 10 * \sqrt{2}$ and the straight *distances* to $10 = 10 * 1$.

Figure 5.3 shows the similarities between connections on a grid-based graph. Because of the similar *distances*, the Dijkstra algorithm is losing its major advantage over a simple flood fill. It is therefore generally a good idea, to implement a more

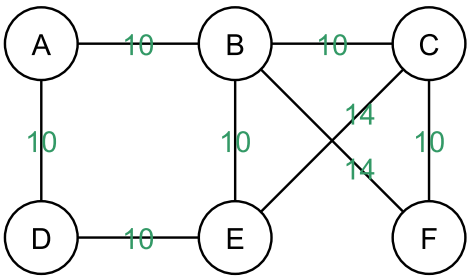


Figure 5.3: Graph with Labelled Edges

sophisticated approach, like A*, when doing pathfinding on a grid-like map.

But also the implementation of A* can be a big factor on how effective it is. This gets mainly determined by the chosen heuristic, because that has big influence on sorting the nodes [18].

On a grid, it makes little sense to use the *euclidean distance*. Because movement is very restricted, and can only be a multiple of the possible *distances*. Here it could make more sense to use the *Manhattan Distance*, because that actually represents a possible path. The *Manhattan Distance* is obtained by the formula shown in Example 5.2.

Example 5.2 (Manhattan Distance)

$$|A_X - B_X| + |A_Y - B_{Y1}| \tag{5.2}$$

5.7 Implementation

Our explanation of the implementation relies heavily on our program written in C. Every function explained in this section can also be read in Appendix H.

Since our map consists of *nodes* with up to eight neighbours, as explained in Section 3.6, we decided to store this information in a single byte per *node*. In this byte, the *Least Significant Nibble (LSN)* represents the straight directions N,E,S,W. The *Most Significant Nibble (MSN)* represents the diagonal directions NE,SE,SW,NW. Some example bytes can be seen in Table 5.2.

Table 5.2: Examples of Bytes representing Walls
path gets stored as 0, WALL as 1

NW	SW	SE	NE	W	S	E	N	byte
path	path	path	path	path	path	path	WALL	0x01
path	path	path	path	path	path	WALL	path	0x02
path	path	path	path	path	WALL	WALL	WALL	0x07
path	path	path	path	WALL	WALL	WALL	path	0x0E
path	path	path	WALL	path	path	path	path	0x10
path	path	WALL	path	path	path	path	path	0x20
path	WALL	path	path	path	path	WALL	path	0x42
WALL	WALL	WALL	WALL	WALL	WALL	WALL	WALL	0xFF

Listing 5.1 shows how we define the different directions in our program.

Listing 5.1: Definition of Directions in `defs.h`

```
11 #define North      0x01
12 #define East       0x02
13 #define South      0x04
14 #define West       0x08
15 #define NorthEast  0x10
16 #define SouthEast  0x20
17 #define SouthWest  0x40
18 #define NorthWest  0x80
```

After reading the map from the input file, as explained in Section 3.4, all *nodes* have to be set up for the pathfinding algorithm. We do this in the function `path_set_neighbors`. Every *node* gets linked to its neighbours by comparing the `walls` value with the defined directions as shown in Listing 5.2. Every direction that does not have a wall, gets linked as a pointer. If it has a wall, the pointer is set to `NULL`. We only show the linking of the straight neighbours, because the diagonal neighbours work basically the same.

Listing 5.2: Linking of straight Neighbours in `path.c`

```
9       if (!(robot->map.node[i][j].walls & North))   //NORTH exists
10        {robot->map.node[i][j].n=&robot->map.node[i-1][j];}
11      else {robot->map.node[i][j].n=NULL;}
12      if (!(robot->map.node[i][j].walls & East))    //EAST exists
13        {robot->map.node[i][j].e=&robot->map.node[i][j+1];}
14      else {robot->map.node[i][j].e=NULL;}
15      if (!(robot->map.node[i][j].walls & South))   //SOUTH exists
16        {robot->map.node[i][j].s=&robot->map.node[i+1][j];}
17      else {robot->map.node[i][j].s=NULL;}
18      if (!(robot->map.node[i][j].walls & West))    //WEST exists
19        {robot->map.node[i][j].w=&robot->map.node[i][j-1];}
20      else {robot->map.node[i][j].w=NULL;}
```

The same function also initially sets the `movecost` (*cost to reach*) to 4095 (`0xFFF`) for all *nodes*, except for the starting *node*, and points `parent` to `NULL`. This can be seen in Listing 5.3.

Listing 5.3: Setting `movecost` and `parent` in `path.c`

```
35      robot->map.node[i][j].movecost = 0xFFF; //set movecost high
36      robot->map.node[i][j].parent = NULL;    //point parent to NULL
37    }
38  }
39  //-------------------------END 2D for loop-----------------------//
40  //set movecost for the current position to 0
41  robot->map.node[robot->pos.x][robot->pos.y].movecost = 0;
```

After everything is set up, the path-finding algorithm can start.

We implemented Dijkstra's algorithm in the function `path_calculate`, shown in Listing 5.4. Here we have two integers `curx` and `cury` to keep track of the position and a `currNode` as our current *node*. The algorithm loops through all *nodes*, until it reaches the finish *node*. We implemented this as a `while` loop.

To prevent the algorithm from looping infinitely, in case of a subtle error, we increment a counter `deadcount` on every iteration and check whether it surpasses the amount of *nodes*.

Inside the loop we check for every *neighbour*, whether a pointer to it exists and whether its *cost to reach* is higher than it would be from the current *node*. In the case that both statements are true, we update the neighbours *cost to reach* and *parent*.

We also remove the current *node* as a neighbour, because we know the path back and forth cannot be shorter. This removes one lookup going through three pointers and involving one addition.

Listing 5.4: Calculating the Path in `path.c`

```
64  while ((curx!=robot->map.finish.y)||(cury!=robot->map.finish.x)){
65    if (currNode==NULL){
66      printf("[WARN]\tsomething went wrong, current node is NULL\n");
67      break;
68    } //catches NULL pointers
69    if (deadcount++>=(robot->map.nSize.x)*(robot->map.nSize.y)){
70        printf("[ERROR]\tEVERYTHING WENT WRONG, looping\n"); return;
71    } //catching infinite loops
72    curx = currNode->position.x;
73    cury = currNode->position.y;
74
75    /*explaining the ifs below
76    checks whether the neighbor exists, then if movecost is smaller
77      updates neighbor movecost
78      updates neighbor parent
79      remove parent as neighbor
80      pushes neighbor on queue
81    */
82
83    //-------------------------------STRAIGHTS--------------------//
84    if(currNode->n&&(currNode->n->movecost > currNode->movecost+10))
85    { currNode->n->movecost = currNode->movecost+10;
86      currNode->n->parent = currNode;
87      currNode->n->s = NULL;
88      push_queue(&robot->unchecked, currNode->n); }
```

After looking through all neighbours of a *node*, it can be pushed onto the `checked` stack. Our implementation of the stack is documented in Appendix I.

The next node has now to be popped from the `unchecked` queue, before looping to the start of the `while` again. This is done in Listing 5.5.

Listing 5.5: Pushing and Popping Nodes in `path.c`

```
127   push_stack(&robot->checked, currNode);  //mark current as checked
128   printf("[INFO]\tNode [%2d][%2d] computed!\n",curx,cury);
129
130   currNode = pop(&robot->unchecked);  //get a new node from queue
131 }//--------------------------END WHILE--------------------------//
```

When all *nodes* leading to the finish *node* have been checked, the full path is in the `checked` stack. But also every other *node* with a smaller *cost to reach*. The top element on the stack is also the finish *node*, so the first movement gets described in the bottom of the stack.

This means it is necessary to sort through the stack, and to only keep the necessary *nodes*. While doing this we can immediately calculate the movements to be done and put them on a new stack. Since the *node*-stack had the finish *node* at the top, the movement stack will have the finish movement at the bottom.

We calculate the movements out of the coordinates of the *nodes*, in function `path_calculate_movement` as can be seen in listing 5.6. Movement can be seen as a vector, where a change in X coordinate shows movement along the North-South axis, and change in Y coordinate shows movement along the East-West axis. If only one axis changes, the resulting movement is straight, if change happens on both axes, it is diagonal.

Listing 5.6: Pushing and Popping Nodes in `path.c`

```
145   //----------------------LOOP THROUGH STACK----------------------//
146   currNode=pop(&robot->checked);
147   while(currNode->movecost!=0){ //start has movecost 0
148     ownX = currNode->position.x;
149     ownY = currNode->position.y;
150     parX = currNode->parent->position.x;
151     parY = currNode->parent->position.y;
152     //----------------------Generate Movement----------------------//
153     move=0;   //resets move, then adds all movements together
154     if (ownX<parX) move+=North;          //Something North
155     else if (ownX>parX) move+=South;     //Something South
156     if (ownY>parY) move+=East;           //Something East
157     else if (ownY<parY) move+=West;      //Something West
158
159     //------------------Detect Diagonal Movement------------------//
160     if (((move!=North)&&(move!=East)&&(move!=South)&&(move!=West))){
161       if       (move==North+East) move=NorthEast;  //North and East
162       else if (move==South+East) move=SouthEast;  //South and East
163       else if (move==South+West) move=SouthWest;  //South and West
164       else if (move==North+West) move=NorthWest;  //North and West
165     }   //now we know the exact moving direction!
166     //-------------------Save To Movement Stack-------------------//
167     push_move_stack(&robot->movement, move);
168     //------------------------ Find Parent-----------------------//
169     while (((parNode=pop(&robot->checked))->position.x!=parX)||
170            (parNode->position.y!=parY)){}
171     currNode=parNode;
172   }
173 }
```

Chapter 6

Perspective on Efficiency

The code written for this project was not designed to be efficient, but to be understandable to the reader and the authors, because efficiency was outside the scope of the project.

In a real world scenario, efficiency both in time and power consumption is crucial to the effectiveness of a rescue robot.

Because of this we want to at least formulate our thoughts on the topic.

6.1 Pathfinding

Pathfinding is probably the most resource-heavy calculation in our system, and will likely be executed several times while running. Every small change in efficiency is therefore multiplied by the amount of recalculations needed.

Most pathfinding algorithms need several variables to be stored and accessed multiple times, for each node. We decided to store them in structs for our implementation.

Listing 6.1: Declaring a Struct in `defs.h`

```
24 typedef struct Node {
25   Point position;               // Nodes own x,y position on node map
26   struct Node *n,*e,*s,*w;      // Pointers to neighbors straight
27   struct Node *nw,*ne,*se,*sw;  // Pointers to neighbors diagonal
28   struct Node *parent;          // Pointer to parent node
29   unsigned char walls;          // Byte value for the 8 walls
30   int movecost;                 // Steps needed to get here
31 } Nodes;
```

An example of a struct declaration can be seen in Listing 6.1. A struct stores all declared variables as an individual variable, with pointers to them. This makes it easy to access them for the programmer, but creates a notable overhead when compiling.

Depending on the word size of the used processor, it could be possible to store all information necessary for a node in one variable. This would necessitate inventing a custom encoding, but would possibly improve time consumption on runtime.

One idea for this would be to reserve sections of the variables bits for specific information, our thoughts on how that might look can be seen in Table 6.1.

Table 6.1: Example of a possible custom Encoding for Nodes

position								walls								parent								movecost							
1	1	0	1	1	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	0	0	1	0	1	0	1	1	1	1	1	1

Single values could be retrieved through simple bitwise operations, implemented either in hardware or in software. To retrieve the position for example, the whole value could be shifted to the right 24 times.

To select the parent position, one could take the value `AND 0xFF00` and afterwards shift to the right 8 times.

This would make it possible to store the newly invented variables in a 2D array, like we do now with our `node struct`. But since we would refer to every node as its position, we would only need to make single array lookups, instead of multiple pointer lookups.

Our current approach takes use of pointers a lot, often iterating through several substructs. Listing 6.2 shows code with multiple pointer accesses per line. Code similar to this listing gets executed up to 8 times per *node*, for every *node* with a lower *cost to reach* than the finish *node*, to check for every direction whether moving there would be shorter.

Listing 6.2: Accessing several Fields through multiple Pointers in `path.c`

```
84    if(currNode->n&&(currNode->n->movecost > currNode->movecost+10))
85    { currNode->n->movecost = currNode->movecost+10;
86      currNode->n->parent = currNode;
87      currNode->n->s = NULL;
88      push_queue(&robot->unchecked, currNode->n); }
```

When implementing Dijkstra's algorithm or A*, there is no need to look at an already checked *node*. It could therefore be an option to remove all pointers to a node, when pushing it to the checked stack.

In our current implementation, those pointers can only be set to `NULL` after finding them through another pointer lookup. We are already doing this in line 87 in Listing 6.2, when removing the reference to the parent as a neighbour, to remove redundant information and ease the lookup procedure.

The whole part of linking the *nodes* to their neighbours, as shown in 6.3, uses the aforementioned `walls` byte, to compute in what directions to link. This means that basically the neighbours are stored redundantly, once in the `walls` byte and once in the eight pointers.

Listing 6.3: Linking Nodes based on `walls` variable in `path.c`

```
7   for (int i = 0; i<(robot->map.size.x-1)/2; i++){
8     for (int j=0; j<(robot->map.size.y-1)/2; j++){
9       if (!(robot->map.node[i][j].walls & North))    //NORTH exists
10        {robot->map.node[i][j].n=&robot->map.node[i-1][j];}
11      else {robot->map.node[i][j].n=NULL;}
12      if (!(robot->map.node[i][j].walls & East))     //EAST exists
13        {robot->map.node[i][j].e=&robot->map.node[i][j+1];}
14      else {robot->map.node[i][j].e=NULL;}
15      if (!(robot->map.node[i][j].walls & South))    //SOUTH exists
16        {robot->map.node[i][j].s=&robot->map.node[i+1][j];}
17      else {robot->map.node[i][j].s=NULL;}
18      if (!(robot->map.node[i][j].walls & West))     //WEST exists
19        {robot->map.node[i][j].w=&robot->map.node[i][j-1];}
20      else {robot->map.node[i][j].w=NULL;}
```

6.2 Scan

When developing the `scan` function, we had already thought about efficiency to some extend. Our robot was designed specifically to move in eight directions, so we could store the information about each *node* in one byte.

We therefore designed the function to return one byte value, representing the environment at the current position.

We didn't have the opportunity to finalise our idea, since we had troubles acquiring functioning sensors, and only used hardcoded values for testing.

6.3 Map Handling

Our approach to map handling was also mainly focused on usability, not on efficiency.

We also think, that this specific subsystem needs to be very fail-safe, and therefore persistent. Without the map handling, the robot can't keep track of its current position, which it also should be able to find again after losing power.

Listing 6.4 shows our current approach to knowing the map size.

Listing 6.4: Loading the Map from a File in `map.c`

```
31 void map_load(Robot *robot) {
32   // Open file in read mode
33   FILE *myfile = fopen(MAP_FILENAME, "r");
34
35   // Count number of lines and characters
36   int rows = 1;        // Counts newlines
37   int cols = 0;        // Counts characters
38   int c;               // Holds each character as it is read from file
39
```

```
40  while ((c = fgetc(myfile)) != EOF) { // Read file characterwise
41    if (c == '\n') {  // end of the line (linebreak)
42      rows++;         // Count the line
43      cols = 0;       // Reset character counter
44    } else cols++;    // Count characters
45  }
46  rewind(myfile);     // reset file pointer
```

We are reading the file twice, counting the characters and the lines first and storing the map content in the second iteration.

This could for example be prevented through the user inputting the map size manually, or storing the dimensions in the header of the map file.

Listing 6.5: Loading the Node Map from the stored Map in `map.c`

```
105 for (int i=1; i<robot->map.size.y; i+=2) {
106   for (int j=1; j<robot->map.size.x; j+=2) {
107     // For each node check in 8 directions and build 8-bit value
108     // # means a wall exists, space means a path exists
109     hex = 0;    //reset for each node
110     hex += (robot->map.segments[i-1][j+0] == '#') ? North : 0;
111     hex += (robot->map.segments[i-0][j+1] == '#') ? East  : 0;
112     hex += (robot->map.segments[i+1][j-0] == '#') ? South : 0;
113     hex += (robot->map.segments[i-0][j-1] == '#') ? West  : 0;
114     hex += (robot->map.segments[i-1][j+1] == '#') ? NorthEast : 0;
115     hex += (robot->map.segments[i+1][j+1] == '#') ? SouthEast : 0;
116     hex += (robot->map.segments[i+1][j-1] == '#') ? SouthWest : 0;
117     hex += (robot->map.segments[i-1][j-1] == '#') ? NorthWest : 0;
118     robot->map.node[(i-1)/2][(j-1)/2].walls = hex;  //save in struct
119     // Save node position on node map
120     robot->map.node[(i-1)/2][(j-1)/2].position.x = (i-1)/2;
121     robot->map.node[(i-1)/2][(j-1)/2].position.y = (j-1)/2;
122   }
123 }
```

After loading the map into memory, we have to convert the text into a node map. This process happens in two nested `for` loops, as can be seen in Listing 6.5.

For a final, more efficient solution, we would want to store the map in a format ready for import, and have all conversion happen on the users computer.

The scope of the project didn't allow us to develop a file format and external program to do this, but we strongly believe, that it would make the map handling more efficient.

6.4 Movement

Because the movement subsystem was developed very close to hardware level, we believe that it already is fairly efficient. We therefore want to highlight the measures we took, to ensure this.

All four motors get controlled by a single stepping sequence, not only making them run synchronously, but also necessitating less interrupts than stepping them individually. Subsection 2.3.2 explains the circuit we developed, to make this possible.

Listing 6.6: Part of the `stepping` function in `movement.c`

```
13 void stepping(void){
14   unsigned int step = 1, counter = 0;    //defining step and counter
15                                          //variable
16   CCR0 = 8000;                           //upper limit for the timer
17   TACTL = MC_1 | ID_0 | TASSEL_2 |TACLR;//initializing timer, up mode
18                                          //divided by 1, source select,
19                                          //clear
20   while(counter < 1000){                 //number of steps
21     while((TACTL & 0x0001) == 0){}       //halfstepping from here down
22     TACTL &= ~0x0001;                    //resetting the interrupt flag
23     if(step == 1){                       //A1
24       P1OUT &= ~(BIT1 + BIT2 + BIT3);
25       P1OUT |= BIT4;
26       step++;
27       counter++;
28     }
```

The stepping shown in Listing 6.6 mainly relies on the timer circuits in the microcontroller, which are accessible also in most powered down modes.

We also believe, that it would be possible and fairly easy, to implement this behaviour using sequential logic, in which way it could run directly off an external clock source. Our thoughts on what that might look like are depicted in Figure 6.1.

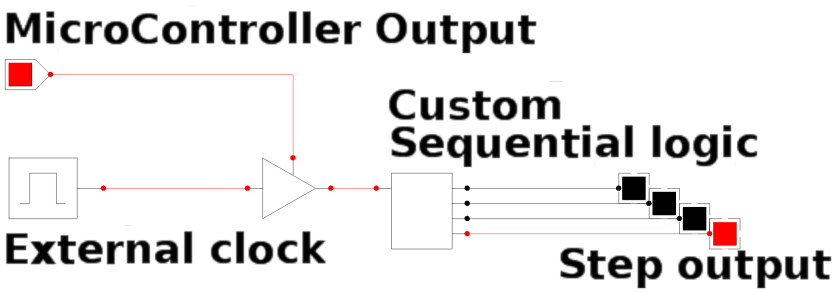


Figure 6.1: Mock-up of a sequential stepping Solution

Chapter 7

Conclusion

Based on the discussion in the previous chapter, we conclude that the goal of researching pathfinding algorithms and their implementation on a microprocessor was met to great extend.

The previous discussion about efficiency shows, that the topic is not researched completely in this report, but gives pointers to where future research should be headed.

We initially set out to develop a system subdivided into smaller parts, so we could better accomplish a project fitting real live scenarios.

This goal was met to our full satisfaction in the subsystems, but due to unforeseeable events during the development of this project, we needed to compromise on combining all subsystems together.

Subsystems were developed and tested separately by the group members, either individually or in teams, with every member having influence on each of the subsystems during the development phase.

Unfortunately, at the time of writing the report, we did not manage to implement a full solution. However, having tested all subsystems individually, checking their outputs and inputs to fit the defined interfaces, we have a strong belief that the system will work as a whole in the near future.

All together we would say that the research going into this report was successful, giving the authors and hopefully readers better knowledge about pathfinding and microprocessors.

Bibliography

[1] Daniel Busemann, Troels Klein, and Razvan Bucur. *Pathfinding -as used in Rescuing Robots- (a 3rd Semester Project)*. https://github.com/BlackAndWhiteSnaable/testLTC. 2017.

[2] Jennifer Krieger et al. *RoboCupJunior Rescue Maze – Rules 2017*. https://www.robocupgermanopen.de/sites/default/files/rescue_maze_2017.pdf. Rules available from https://www.robocupgermanopen.de/de/junior/rescue Accessed: 2017-12-16. 2017.

[3] Simon Bennett and Ray Farmer. *Object-Oriented Systems Analysis and Design Using UML*. Maidenhead, Berkshire: McGraw-Hill Education, 2010. ISBN: 0077125363.

[4] Microchip. *Introduction to Stepper Motors*. 2007. URL: http://www.microchip.com/stellent/groups/SiteComm_sg/documents/DeviceDoc/en543047.pdf.

[5] JColvin91. *How to Use a Stepper Motor*. URL: http://www.instructables.com/id/How-to-use-a-Stepper-Motor/.

[6] Manuel Jimenez, Rogelio Palomera, and Isidoro Couvertier. *Introduction to Embeddd Systems*. Springer, 2014.

[7] *DRV8833 Dual Motor Driver Carrier*. URL: https://www.mgsuperlabs.co.in/estore/DRV8833-Dual-Motor-Driver-Carrier.

[8] *Actobotics 4" Heavy Duty Wheel*. URL: https://www.robotshop.com/en/actobotics-4-heavy-duty-wheel.html.

[9] *60mm Aluminum Omni Wheel*. URL: https://www.robotshop.com/en/60mm-aluminum-omni-wheel.html.

[10] *Digital Buffer Tutorials*. URL: http://www.electronics-tutorials.ws/logic/logic_9.html.

[11] Texas Instruments. *MSP430 USB Stick Development Tool*. 2010. URL: http://www.ti.com/tool/EZ430-F2013.

[12] *Sharp IR Range Sensor - 10cm to 80cm W/ Cablel*. URL: https://www.robotshop.com/en/sharp-gp2d12-ir-range-sensor-cable.html.

[13] SHARP. *General Purpose Type Distance Measuring Sensors*. URL: https://www.sparkfun.com/datasheets/Components/GP2Y0A21YK.pdf.

[14] Douglas P. Zipes. "Saving Time Saves Lives". In: *Circulation* 104.21 (2001), pp. 2506–2508. ISSN: 0009-7322. eprint: http://circ.ahajournals.org/content/104/21/2506.full.pdf. URL: http://circ.ahajournals.org/content/104/21/2506.

[15] Vaibhav Jaimini. *Flood-fill Algorithm*. https://www.hackerearth.com/practice/algorithms/graphs/flood-fill-algorithm/tutorial/. Accessed: 2017-12-11. 2017.

[16] Dr. Mike Pound. *Dijkstra's Algorithm*. https://youtu.be/GazC3A4OQTE. Accessed: 2017-12-13. 2017.

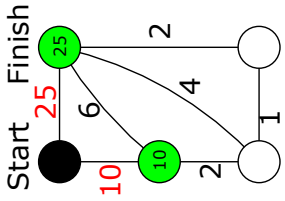
[17] Dr. Mike Pound. *A* (A Star) Search Algorithm*. https://youtu.be/ySN5Wnu88nE. Accessed: 2017-12-13. 2017.

[18] Nathan Sturtevant. *A* Tie Breaking*. http://movingai.com/astar.html. Accessed: 2017-12-14. 2017.

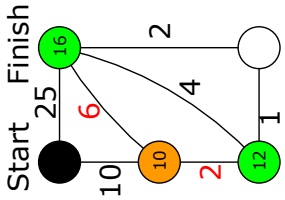
Appendix A

Dijkstra Pathfinding Algorithm

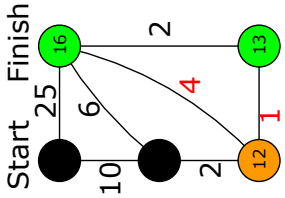
Algorithm



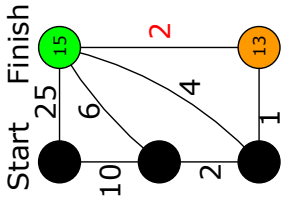
(a) 1. Step



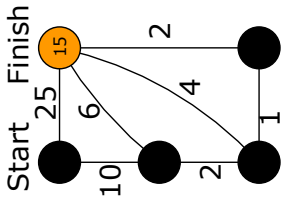
(b) 2. Step



(c) 3. Step



(d) 4. Step



(e) 5. Step

Appendix B

Code: movement.c

Listing B.1: `movement.c`

```
#include <msp430x20x3.h>

#define North           0x01
#define East            0x02
#define South           0x04
#define West            0x08

#define NorthEast       0x10
#define SouthEast       0x20
#define SouthWest       0x40
#define NorthWest       0x80

void stepping(void){
  unsigned int step = 1, counter = 0;    //defining step and counter
                                         //variable
  CCR0 = 8000;                           //upper limit for the timer
  TACTL = MC_1 | ID_0 | TASSEL_2 |TACLR;//initializing timer, up mode
                                         //divided by 1, source select,
                                         //clear
  while(counter < 1000){                 //number of steps
    while((TACTL & 0x0001) == 0){}       //halfstepping from here down
    TACTL &= ~0x0001;                    //resetting the interrupt flag
    if(step == 1){                       //A1
      P1OUT &= ~(BIT1 + BIT2 + BIT3);
      P1OUT |= BIT4;
      step++;
      counter++;
    }
    else if(step == 2){                  //A1 B1
      P1OUT |= BIT1 + BIT4;
      P1OUT &= ~(BIT2 + BIT3);
      step++;
      counter++;
```

```
34      }
35     else if(step == 3){                  //B1
36       P1OUT |= BIT1;
37       P1OUT &= ~(BIT2 + BIT3 + BIT4);
38       step++;
39       counter++;
40      }
41     else if(step == 4){                  //B1 A2
42       P1OUT |= BIT1 + BIT3;
43       P1OUT &= ~(BIT2 + BIT4);
44       step++;
45       counter++;
46     }
47     else if(step == 5){                  //A2
48       P1OUT &= ~(BIT1 + BIT2 + BIT4);
49       P1OUT |= BIT3;
50       step++;
51       counter++;
52     }
53     else if(step == 6){                  //A2 B2
54       P1OUT &= ~(BIT1 + BIT4);
55       P1OUT |= BIT2 + BIT3;
56       step++;
57       counter++;
58     }
59     else if(step == 7){                  //B2
60       P1OUT &= ~(BIT1 + BIT3 + BIT4);
61       P1OUT |= BIT2;
62       step++;
63       counter++;
64     }
65     else if(step == 8){                  //B2 A1
66       P1OUT &= ~(BIT1 + BIT3);
67       P1OUT |= BIT2 + BIT4;
68       step = 1;
69       counter++;
70     }
71   }
72 }
73
74 void moveNorthWest(void){
75   P1OUT |= BIT5                         //enable pair of wheels
76   P1OUT |= BIT6;                        //choose direction
77   stepping();                           //call stepping function
78   P1OUT &= ~BIT5;                       //disable pair of wheels
79 }
80
81 void moveSouthEast(void){
82   P1OUT |= BIT5;                        //enable pair of wheels
83   P1OUT &= ~BIT6;                       //choose direction
84   stepping();                           //call stepping funtion
```

```
85    P1OUT &= ~BIT5;                          //disable pair of wheels
86  }
87
88  void moveNorthEast(void){
89    P1OUT |= BIT7;                           //enable pair of wheels
90    P2OUT |= BIT7;                           //choose direction
91    stepping();                              //call stepping funtcion
92    P1OUT &= ~BIT7;                          //disable pair of wheels
93  }
94
95  void moveSouthWest(void){
96    P1OUT |= BIT7;                           //enable the pair of wheels
97    P2OUT &= ~BIT7;                          //choose direction
98    stepping();                              //call stepping function
99    P2OUT &= ~BIT7;                          //disable pair of wheels
100 }
101
102 void moveNorth(void){
103   P1OUT |= BIT5;                           //enable first pair of wheels
104   P1OUT |= BIT6;                           //choose direction
105   P1OUT |= BIT7;                           //enable second pair of wheels
106   P2OUT |= BIT7;                           //choose direction
107   stepping();                              //call stepping funcion
108   P1OUT &= ~(BIT5 + BIT7);                 //disable both pairs of wheels
109 }
110
111 void moveWest(void){
112   P1OUT |= BIT5;                           //enable first pair of wheels
113   P1OUT |= BIT6;                           //choose direction
114   P1OUT |= BIT7;                           //enable second pair of wheels
115   P2OUT &= ~BIT7;                          //choose direction
116   stepping();                              //call stepping function
117   P1OUT &= ~(BIT5 + BIT7);                 //disable both pairs of wheels
118 }
119
120 void moveSouth(void){
121   P1OUT |= BIT5 + BIT7;                    //enable first pair of wheels
122   P1OUT &= ~BIT6;                          //choose direction
123   P1OUT |= BIT7;                           //enable second pair of wheels
124   P2OUT &= ~BIT7;                          //choose direction
125   stepping();                              //call stepping function
126   P1OUT &= ~(BIT5 + BIT7);                 //disable both pairs of wheels
127 }
128
129 void moveEast(void){
130   P1OUT |= BIT5 + BIT7;                    //enable first pair of wheels
131   P1OUT &= ~BIT6;                          //choose direction
132   P2OUT |= BIT7;                           //enable second pair of wheels
133   P1OUT |= BIT7;                           //choose direction
134   stepping();                              //call stepping function
135   P1OUT &= ~(BIT5 + BIT7);                 //disable both pairs of wheels
```

```
136 }
137
138 void move(unsigned int x){
139   switch(x){
140   case 0x01: moveNorth();       break;
141   case 0x02: moveEast();        break;
142   case 0x04: moveSouth();       break;
143   case 0x08: moveWest();        break;
144   case 0x10: moveNorthEast();   break;
145   case 0x20: moveSouthEast();   break;
146   case 0x40: moveSouthWest();   break;
147   case 0x80: moveNorthWest();   break;
148   }
149 }
```

Appendix C

Code: scan.ino

Listing C.1: `scan.ino`

```
#define sensor A0                          //define sensor pin
int distance;                              //variable to store distance
float volts;                               //variable to store reading
                                           //from sensor
bool path;                                 //variable to store path
void setup(){
  Serial.begin(9600);                      //start the serial port
}
void loop(){
  volts = analogRead(sensor) * 0.0048828125;  //sensor * (supply
                                              //voltage/
                                              //adc resolution)
  distance = 29.988 * pow(volts , -1.173);    //determined from
                                              //datasheet graph
  delay(500);                              //delay between readings
  if (distance <= 50){                     //no path
    path = 0;
    Serial.print("Wall ");
    Serial.print(distance);
    Serial.print("cm ");
    Serial.println(path);
  }
  else{                                    //path
    path = 1;
    Serial.print("No Wall ");
    Serial.println(path);
  }
}
```

Appendix D

Code: defs.h

Listing D.1: `defs.h`

```
/********************
 defs.h
********************/
#include <stdio.h>      // Needed for printf
#include <stdlib.h>     // Needed for malloc

#define TRUE 1
#define FALSE 0
#define MAP_FILENAME "testmap.txt"  // Map to load

#define North     0x01
#define East      0x02
#define South     0x04
#define West      0x08
#define NorthEast 0x10
#define SouthEast 0x20
#define SouthWest 0x40
#define NorthWest 0x80

typedef struct {
  int x,y;
} Point;                        // A point, consisting of two integers

typedef struct Node {
  Point position;               // Nodes own x,y position on node map
  struct Node *n,*e,*s,*w;      // Pointers to neighbors straight
  struct Node *nw,*ne,*se,*sw;  // Pointers to neighbors diagonal
  struct Node *parent;          // Pointer to parent node
  unsigned char walls;          // Byte value for the 8 walls
  int movecost;                 // Steps needed to get here
} Nodes;

typedef struct {
```

```
34   Point start;                    // Starting position
35   Point finish;                   // Finish position
36   Point size;                     // Amount of segments in the map
37   //Point num_nodes;              // Number of nodes in the map
38   unsigned char **segments;       // 2D array of the map data from file
39   Nodes **node;                   // 2D array of nodes
40 } Maps;
41
42 typedef struct element {
43   Nodes *node;                    // Pointer to the map node
44   struct element *next;          // Next element in queue
45 } Queue, Stack;
46
47 typedef struct {
48   Point pos;
49   Maps map;
50   Queue *unchecked;               // Head of queue for unchecked nodes
51   Queue *checked;                 // Head of stack for checked nodes
52 } Robot;
53
54 //---------------------------- FUNCTIONS ----------------------------//
55
56 // Robot
57 void go();
58 Robot *init_robot();
59
60 // Map
61 void map_load(Robot *robot);
62 void map_save(Robot *robot);
63 void map_check(Robot *robot);
64 void map_update(Robot *robot, char hex);
65 void node_map_load(Robot *robot);
66 int robot_finished(Robot *robot);
67 void test_node_array(Robot *robot);
68
69 // Scan
70 unsigned char scan();
71
72 // Move
73 void move_next(Robot *robot);
74
75 // Priority queue
76 void pushQ(Queue **HoQ, Nodes *new_node); // Add element on the stack
77 void pop(Queue *tq);                      // Pops element from stack
78 void printQueue(Queue *tq);               // Prints stack
79 void emptyQueue(Queue *tq);               // Pops whole stack
80
81 // Pathfinding
82 void path_test(Robot *robot);
83 void path_set_neighbors(Robot *robot);
84 void path_calculate(Robot *robot);
```

Appendix E

Code: main.c

Listing E.1: `main.c`

```c
#include "defs.h"

int main() {
  go(); // Calls all other functions
  return 0;
}

// Executes robot behavior instructions
void go() {
  // Setup
  Robot *robot;           // Declare empty pointer to a struct of
    type Robot
  robot = robot_init();   // Allocate structs and return the address
    to pointer

  // Load maps
  map_load(robot);        // Load map data from text-file
  node_map_load(robot);   // Convert text map into node map

  path_calculate(robot);  // Calculate path

  // While robot has not reached the finish position
  while (!robot_finished(robot)) {
    robot_print(robot);

    // Scan surroundings at current position and compare with map
    segment
    // if scan and map differs update map segment save file and
    recalculate path
    map_check(robot);

    // Move to next position
    move_next(robot);
```

```
30   }
31
32   // While loop has ended so robot must be at finish coordinates
33
34   // Print info to screen for debugging
35   printf("\nRobot current pos: %d.%d\n", robot->pos.x, robot->pos.y);
36   printf("### Finishhas been reached ###\n\n");
37
38   printf("Map array size: %dx%d\n", robot->map.size.x,
       robot->map.size.y);
39   printf("Node array size: %dx%d\n\n", (robot->map.size.x-1)/2,
       (robot->map.size.y-1)/2);
40
41   // Print wall byte for each nodes in the map
42   for (int i = 0; i < (robot->map.size.y-1)/2; ++i) { // loop rows
43     for (int j = 0; j < (robot->map.size.x-1)/2; ++j) { // loop cols
44       printf("Node in robot->map.node[%d][%d].walls = 0x%02X\n", i,
       j, robot->map.node[i][j].walls);
45     }
46   }
47 }
```

Appendix F

Code: robot.c

Listing F.1: `robot.c`

```c
#include "defs.h"

/// Returns TRUE if robot current position is identical to map finish position
int robot_finished(Robot *robot) {
  // Compares robot current position with map finish position
  if(robot->pos.x == robot->map.finish.x &&
    robot->pos.y == robot->map.finish.y) {
    return TRUE;
  }
  return FALSE;
}

/// Allocate robot struct in memory using malloc, store address in *robot pointer and return it
Robot *robot_init() {
  // Dynamically allocate robot struct in memory and return pointer
  Robot *robot = malloc(sizeof(Robot));

  // First queue and stack element must be initialized to NULL
  robot->unchecked = NULL;
  robot->checked = NULL;
  robot->movement = NULL;

  return robot;
}
```

Appendix G

Code: map.c

Listing G.1: `map.c`

```
#include "defs.h"

/// Scan surroundings at current position and compare with map segment
void map_check(Robot *robot) {
  unsigned char scan_segment = scan();
  unsigned char map_segment =
    robot->map.node[robot->pos.x][robot->pos.y].walls;
  if(scan_segment != map_segment) {   //segment changed
    map_update(robot, scan_segment);
    map_save(robot);                  //save to file
    path_calculate(robot);            //recalculate
  }
}

/// Write map data to text file
void map_save(Robot *robot) {
  // Open the file in write mode
  FILE *myfile = fopen(MAP_FILENAME, "w");
  // Write map segments to text file
  for (int i = 0; i < robot->map.size.y; ++i) {   // loop rows
    for (int j = 0; j < robot->map.size.x; ++j) { // loop cols
      fprintf(myfile, "%c", robot->map.segments[i][j]);
    }
    if(i < robot->map.size.y - 1) {               //not last line
      fprintf(myfile, "\n");                      //ad linebreak
    }
  }
  fclose(myfile);                                 // Close file
}

///Reads user input file, stores map data in struct
void map_load(Robot *robot) {
  // Open file in read mode
```

```
33  FILE *myfile = fopen(MAP_FILENAME, "r");
34
35  // Count number of lines and characters
36  int rows = 1;        // Counts newlines
37  int cols = 0;        // Counts characters
38  int c;               // Holds each character as it is read from file
39
40  while ((c = fgetc(myfile)) != EOF) { // Read file characterwise
41    if (c == '\n') {  // end of the line (linebreak)
42      rows++;         // Count the line
43      cols = 0;       // Reset character counter
44    } else cols++;    // Count characters
45  }
46  rewind(myfile);     // reset file pointer
47
48  unsigned char **array; // Pointer to array
49  array = malloc(rows * sizeof(char*));
50  for (int i=0; i<rows; i++) array[i] = calloc(cols, sizeof(char));
51  // Store the pointer to the 2D array in map struct
52  robot->map.segments = array;
53  // Save map size (rows and cols) in struct
54  robot->map.size.x = cols;
55  robot->map.size.y = rows;
56  // Fill map array with map data from file
57  for (int i = 0; i < rows; i++) {
58    for (int j = 0; j < cols; j++) {
59      c = fgetc(myfile);            // Read next character from file
60      robot->map.segments[i][j] = c;// Store each character in array
61      if (c == 'A') {               // found starting position
62        robot->map.start.x = (j-1)/2;
63        robot->map.start.y = (i-1)/2;
64        printf("[INFO]\tStart position:  [%2d][%2d]\n",
65              robot->map.start.x, robot->map.start.y);
66      }
67      if (c == 'B') {               // found finish position
68        robot->map.finish.x = (j-1)/2;
69        robot->map.finish.y = (i-1)/2;
70        printf("[INFO]\tFinish position: [%2d][%2d]\n",
71              robot->map.finish.x, robot->map.finish.y);
72      }
73    }
74    fgetc(myfile);  // don't save newline
75  }
76  fclose(myfile);   // close file
77
78  for (int i = 0; i < rows; i++) {
79    printf("[INFO]\t");
80    for (int j = 0; j < cols; j++) {
81      printf("%c", robot->map.segments[i][j]);
82    }
83    printf("\n");
```

```
84    }
85    // Set robot current position to map start position
86    robot->pos.x = robot->map.start.x;
87    robot->pos.y = robot->map.start.y;
88  }
89
90  /// loads a node map from the saved map
91  void node_map_load(Robot *robot) {
92    // calculate and store the amount of nodes
93    robot->map.nSize.x = (robot->map.size.x-1)/2;
94    robot->map.nSize.y = (robot->map.size.y-1)/2;
95
96    Nodes **array;  // Declare node array of correct size
97    array = malloc(robot->map.nSize.x * sizeof(Nodes*));
98    for(int i=0; i<robot->map.nSize.x; i++) {
99      array[i] = malloc(robot->map.nSize.y * sizeof(Nodes));
100   }
101   robot->map.node = array;  // store array in struct
102   //----------------------- Calculate Walls-----------------------//
103   unsigned char hex;
104   // loop through all nodes positions in the textmap
105   for (int i=1; i<robot->map.size.y; i+=2) {
106     for (int j=1; j<robot->map.size.x; j+=2) {
107       // For each node check in 8 directions and build 8-bit value
108       // # means a wall exists, space means a path exists
109       hex = 0;    //reset for each node
110       hex += (robot->map.segments[i-1][j+0] == '#') ? North : 0;
111       hex += (robot->map.segments[i-0][j+1] == '#') ? East  : 0;
112       hex += (robot->map.segments[i+1][j-0] == '#') ? South : 0;
113       hex += (robot->map.segments[i-0][j-1] == '#') ? West  : 0;
114       hex += (robot->map.segments[i-1][j+1] == '#') ? NorthEast : 0;
115       hex += (robot->map.segments[i+1][j+1] == '#') ? SouthEast : 0;
116       hex += (robot->map.segments[i+1][j-1] == '#') ? SouthWest : 0;
117       hex += (robot->map.segments[i-1][j-1] == '#') ? NorthWest : 0;
118       robot->map.node[(i-1)/2][(j-1)/2].walls = hex;  //save in struct
119       // Save node position on node map
120       robot->map.node[(i-1)/2][(j-1)/2].position.x = (i-1)/2;
121       robot->map.node[(i-1)/2][(j-1)/2].position.y = (j-1)/2;
122     }
123   }
124 }
125
126 /// updates the map file at the current position to the given value
127 void map_update(Robot *robot, char hex) {
128
129   // Convert node coordinate to map coordinate
130   int i = robot->pos.y*2+1;
131   int j = robot->pos.x*2+1;
132
133   // Update all 8 neighbors according to the hex wall value
134   robot->map.segments[i-1][j+0] = (hex & North) ? '#' : ' ';
```

```
135   robot->map.segments[i-0][j+1] = (hex & East)  ? '#' : ' ';
136   robot->map.segments[i+1][j-0] = (hex & South) ? '#' : ' ';
137   robot->map.segments[i-0][j-1] = (hex & West)  ? '#' : ' ';
138   robot->map.segments[i-1][j+1] = (hex & NorthEast) ? '#' : ' ';
139   robot->map.segments[i+1][j+1] = (hex & SouthEast) ? '#' : ' ';
140   robot->map.segments[i+1][j-1] = (hex & SouthWest) ? '#' : ' ';
141   robot->map.segments[i-1][j-1] = (hex & NorthWest) ? '#' : ' ';
142
143   // Map is now up to date, rebuild nodes based on the updated map
144   free(robot->map.node);  // Free current nodes
145   node_map_load(robot);   // Rebuild nodes
146 }
```

Appendix H

Code: path.c

Listing H.1: `path.c`

```
#include "defs.h"

///finds all neighbors of a node and sets them as pointers
void path_set_neighbors(Robot *robot) {
  printf("\n[INFO]\tStarted linking nodes to neighbors\n");
  //---------------------------2D for loop--------------------------//
  for (int i = 0; i<(robot->map.size.x-1)/2; i++){
    for (int j=0; j<(robot->map.size.y-1)/2; j++){
      if (!(robot->map.node[i][j].walls & North))   //NORTH exists
        {robot->map.node[i][j].n=&robot->map.node[i-1][j];}
      else {robot->map.node[i][j].n=NULL;}
      if (!(robot->map.node[i][j].walls & East))    //EAST exists
        {robot->map.node[i][j].e=&robot->map.node[i][j+1];}
      else {robot->map.node[i][j].e=NULL;}
      if (!(robot->map.node[i][j].walls & South))   //SOUTH exists
        {robot->map.node[i][j].s=&robot->map.node[i+1][j];}
      else {robot->map.node[i][j].s=NULL;}
      if (!(robot->map.node[i][j].walls & West))    //WEST exists
        {robot->map.node[i][j].w=&robot->map.node[i][j-1];}
      else {robot->map.node[i][j].w=NULL;}
      //------------------------ DIAGONALS-------------------------//
      if (!(robot->map.node[i][j].walls & NorthEast))   //NE exists
        {robot->map.node[i][j].ne=&robot->map.node[i-1][j+1];}
      else {robot->map.node[i][j].ne=NULL;}
      if (!(robot->map.node[i][j].walls & SouthEast))   //SE exists
        {robot->map.node[i][j].se=&robot->map.node[i+1][j+1];}
      else {robot->map.node[i][j].se=NULL;}
      if (!(robot->map.node[i][j].walls & SouthWest))   //SW exists
        {robot->map.node[i][j].sw=&robot->map.node[i+1][j-1];}
      else {robot->map.node[i][j].sw=NULL;}
      if (!(robot->map.node[i][j].walls & NorthWest))   //NW exists
        {robot->map.node[i][j].nw=&robot->map.node[i-1][j-1];}
      else {robot->map.node[i][j].nw=NULL;}
```

```
34
35        robot->map.node[i][j].movecost = 0xFFF; //set movecost high
36        robot->map.node[i][j].parent = NULL;    //point parent to NULL
37      }
38    }
39    //-------------------------END 2D for loop-----------------------//
40    //set movecost for the current position to 0
41    robot->map.node[robot->pos.x][robot->pos.y].movecost = 0;
42    printf("[INFO]\tDone linking nodes to neighbors\n");
43  }
44
45  ///calculates the path from the current position
46  void path_calculate(Robot *robot) {
47    //---------------------------- SETUP -----------------------------//
48    //declare all variables needed in scope
49    int curx,cury;          //keeps track of position on the map
50    Nodes *currNode;        //the Node currently looked at
51
52    path_set_neighbors(robot);  //make sure that all nodes are set up
53
54    //-------------------------SETUP FOR CALC-------------------------//
55    curx = robot->pos.x;    //start calculating from current position
56    cury = robot->pos.y;
57    push_queue(&robot->unchecked, &robot->map.node[curx][cury]);
58    currNode =pop(&robot->unchecked);
59
60    //-----------------------------CALC-----------------------------//
61    int deadcount=0;
62    printf("\n\n[INFO]\tstarted path calculation\n\n");
63    //----------------------WHILE not at finish----------------------//
64    while ((curx!=robot->map.finish.y)||(cury!=robot->map.finish.x)){
65      if (currNode==NULL){
66        printf("[WARN]\tsomething went wrong, current node is NULL\n");
67        break;
68      } //catches NULL pointers
69      if (deadcount++>=(robot->map.nSize.x)*(robot->map.nSize.y)){
70          printf("[ERROR]\tEVERYTHING WENT WRONG, looping\n"); return;
71      } //catching infinite loops
72      curx = currNode->position.x;
73      cury = currNode->position.y;
74
75      /*explaining the ifs below
76      checks whether the neighbor exists, then if movecost is smaller
77        updates neighbor movecost
78        updates neighbor parent
79        remove parent as neighbor
80        pushes neighbor on queue
81      */
82
83      //-------------------------------STRAIGHTS--------------------//
84      if(currNode->n&&(currNode->n->movecost > currNode->movecost+10))
```

```
85      { currNode->n->movecost = currNode->movecost+10;
86        currNode->n->parent = currNode;
87        currNode->n->s = NULL;
88        push_queue(&robot->unchecked, currNode->n); }
89      if(currNode->e&&(currNode->e->movecost > currNode->movecost+10))
90      { currNode->e->movecost = currNode->movecost+10;
91        currNode->e->parent = currNode;
92        currNode->e->w = NULL;
93        push_queue(&robot->unchecked, currNode->e); }
94      if(currNode->s&&(currNode->s->movecost > currNode->movecost+10))
95      { currNode->s->movecost = currNode->movecost+10;
96        currNode->s->parent = currNode;
97        currNode->s->n = NULL;
98        push_queue(&robot->unchecked, currNode->s); }
99      if(currNode->w&&(currNode->w->movecost > currNode->movecost+10))
100     { currNode->w->movecost = currNode->movecost+10;
101       currNode->w->parent = currNode;
102       currNode->w->e = NULL;
103       push_queue(&robot->unchecked, currNode->w); }
104
105     //-------------------------------DIAGONALS--------------------//
106     if(currNode->ne&&(currNode->ne->movecost > currNode->movecost+14))
107     { currNode->ne->movecost = currNode->movecost+14;
108       currNode->ne->parent = currNode;
109       currNode->ne->sw = NULL;
110       push_queue(&robot->unchecked, currNode->ne); }
111     if(currNode->se&&(currNode->se->movecost > currNode->movecost+14))
112     { currNode->se->movecost = currNode->movecost+14;
113       currNode->se->parent = currNode;
114       currNode->se->nw = NULL;
115       push_queue(&robot->unchecked, currNode->se); }
116     if(currNode->sw&&(currNode->sw->movecost > currNode->movecost+14))
117     { currNode->sw->movecost = currNode->movecost+14;
118       currNode->sw->parent = currNode;
119       currNode->sw->ne = NULL;
120       push_queue(&robot->unchecked, currNode->sw); }
121     if(currNode->nw&&(currNode->nw->movecost > currNode->movecost+14))
122     { currNode->nw->movecost = currNode->movecost+14;
123       currNode->nw->parent = currNode;
124       currNode->nw->se = NULL;
125       push_queue(&robot->unchecked, currNode->nw); }
126
127     push_stack(&robot->checked, currNode);  //mark current as checked
128     printf("[INFO]\tNode [%2d][%2d] computed!\n",curx,cury);
129
130     currNode = pop(&robot->unchecked);  //get a new node from queue
131   }//---------------------------END WHILE---------------------------//
132
133   printf("\n[INFO]\tDone calculating path!\n");
134   path_calculate_movement(robot);
135 }
```

```
136
137 ///calculates the movement stack out of the checked stack
138 void path_calculate_movement(Robot *robot){
139   //pop from stack until start is reached
140   //---------------------------- VARIABLES----------------------------//
141   int parX=0,parY=0;
142   int ownX=0,ownY=0;
143   char move=0;
144   Nodes *currNode = NULL, *parNode = NULL;
145   //-----------------------LOOP THROUGH STACK-----------------------//
146   currNode=pop(&robot->checked);
147   while(currNode->movecost!=0){ //start has movecost 0
148     ownX = currNode->position.x;
149     ownY = currNode->position.y;
150     parX = currNode->parent->position.x;
151     parY = currNode->parent->position.y;
152     //----------------------Generate Movement---------------------//
153     move=0;   //resets move, then adds all movements together
154     if (ownX<parX) move+=North;          //Something North
155     else if (ownX>parX) move+=South;     //Something South
156     if (ownY>parY) move+=East;           //Something East
157     else if (ownY<parY) move+=West;      //Something West
158
159     //------------------Detect Diagonal Movement------------------//
160     if (((move!=North)&&(move!=East)&&(move!=South)&&(move!=West))){
161       if      (move==North+East) move=NorthEast;  //North and East
162       else if (move==South+East) move=SouthEast;  //South and East
163       else if (move==South+West) move=SouthWest;  //South and West
164       else if (move==North+West) move=NorthWest;  //North and West
165     }   //now we know the exact moving direction!
166     //--------------------Save To Movement Stack--------------------//
167     push_move_stack(&robot->movement, move);
168     //------------------------ Find Parent-----------------------//
169     while (((parNode=pop(&robot->checked))->position.x!=parX)||
170            (parNode->position.y!=parY)){}
171     currNode=parNode;
172   }
173 }
```

Appendix I

Code: stack.c

Listing I.1: `stack.c`

```
#include "defs.h"

// Add element to stack
void push_stack(Stack **head, Nodes *new_node)
{
  Stack *tmp;               // pointer to a Stack
  tmp = (Stack *)malloc(sizeof(Stack));

  if (!*head)               // add element to empty stack
  {
    *head = tmp;            // set pointer to first element = tmp
    (*head)->next = NULL;   // first element has no next => NULL
  }
  else                      // add element to top of existing stack
  {
    tmp->next = *head;      // in new element set *next to current top
    *head = tmp;            // set head to point to the new element
  }
  (*head)->node = new_node; // set value of new element to val passed
}

// print number of elements in stack and their values
void print_stack(Stack *head)
{
  if(!head)
  {
    printf("[WARNING]\tPrint what? Stack is empty\n");
  }
  else
  {
    Stack *cur; // pointer to node currently being traversed
    int i=1; // count stack element position

```

```
34      for (cur = head; cur != NULL; cur = cur->next)
35      {
36        printf(
37          "//-----------------%2d. Stack element-----------------//\n",
38          i
39        );
40        printf(
41          "[DEV]\tposition [%2d][%2d]",
42          cur->node->position.x,
43          cur->node->position.y
44        );
45        if (cur->node->parent) printf(
46                                 "\tparent [%2d][%2d]\n",
47                                 cur->node->parent->position.x,
48                                 cur->node->parent->position.y
49                               );
50        else printf("\n");
51        i++;
52      }
53    }
54    printf("\n");
55  }
```

Appendix J

Code: queue.c

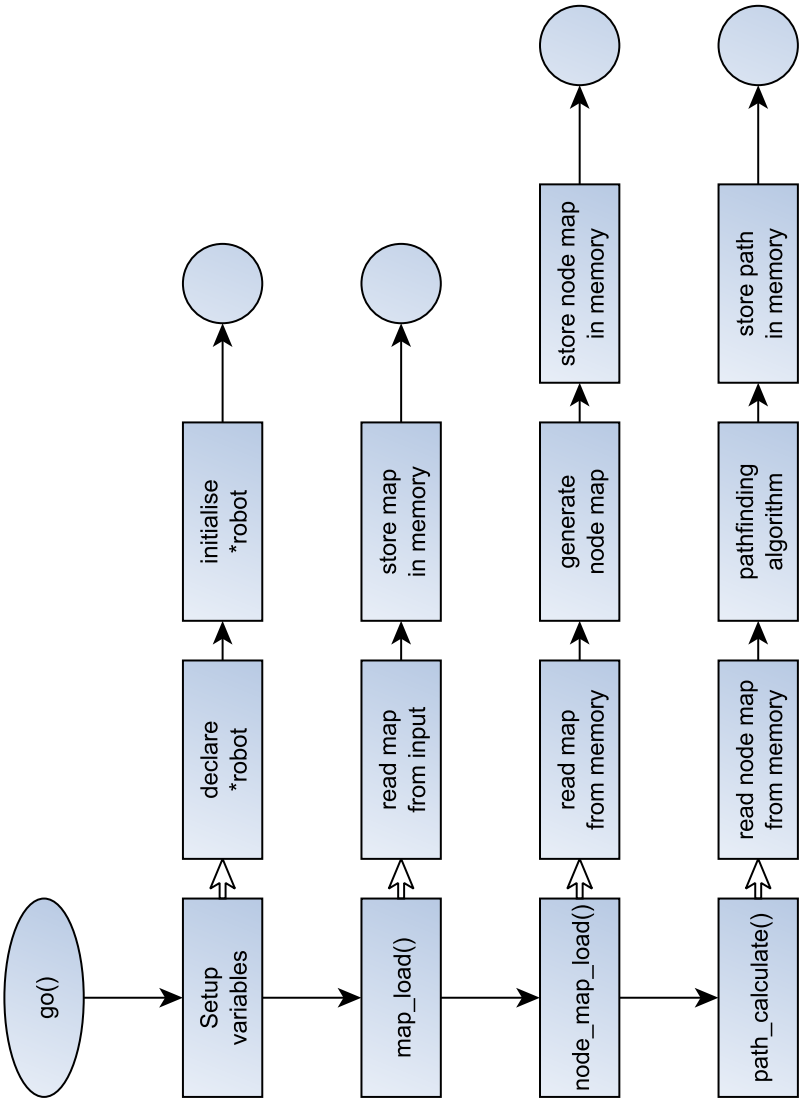
Listing J.1: queue.c

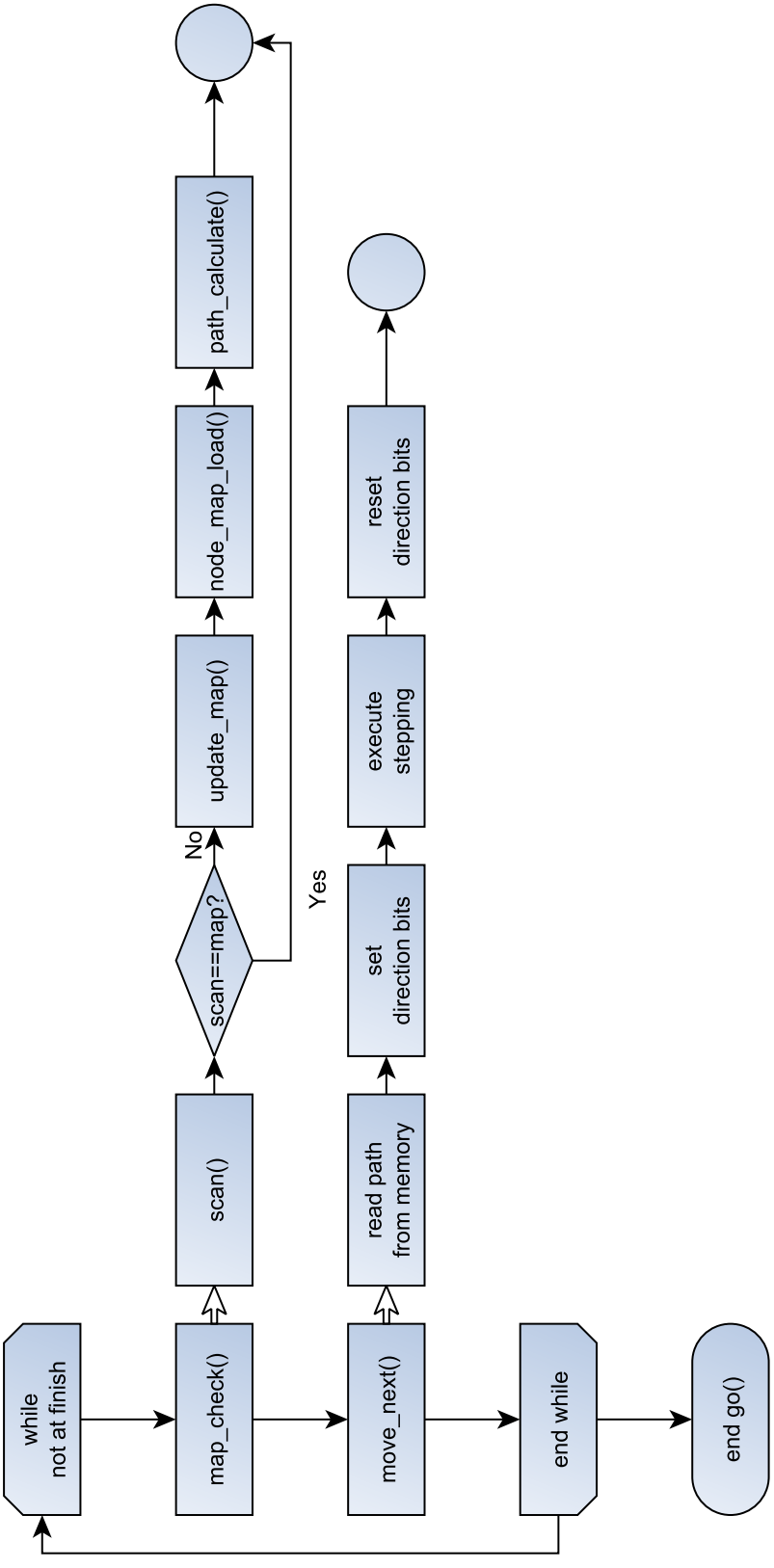
```c
#include "defs.h"

/// Add element to queue
void push_queue(Queue **head, Nodes *new_node)
{
  // Allocate new queue element struct in memory
  Queue *tmp;
  tmp = (Queue *)malloc(sizeof(Queue))

  // queue is empty, insert new Queue element as head
  if (!*head) {
    *head = tmp; // Point head to the new queue element
    (*head)->next = NULL;

  // First element in queue has higher movecost than the new element
  // So insert new element as first element, and update next pointer
  } else if((*head)->node->movecost > new_node->movecost) {
  tmp->next = *head;
  *head = tmp;

  // Insert new element before element with higher movecost value
  } else {
    Queue *cur;
    cur = *head;
    while
    (cur->next != NULL &&
     cur->next->node->movecost <= new_node->movecost) {
    cur = cur->next; // Next node
  }
    tmp->next = cur->next;     // set next pointer to current head
    cur->next = tmp;           // set head queue element to new element
  }
  // Store pointer to node in the new queue element
```

```
34   tmp->node = new_node;
35 //  print_queue(head);
36 }
37
38 /// Print number of elements in queue and their values
39 void print_queue(Queue *head) {
40     if(!head)
41     {
42         printf("[WARN]\tPrint what? Queue is empty\n");
43     }
44     else
45     {
46         Queue *cur; // pointer to node currently being traversed
47         int i=1; // count queue element position
48
49         for (cur = head; cur != NULL; cur = cur->next)
50         {
51             printf(
52               "[INFO]\t%d. Queue element is
53               node[%d][%d] with movecost: %d\n",
54               i,
55               cur->node->position.x,
56               cur->node->position.y,
57               cur->node->movecost
58             );
59             i++;
60         }
61     }
62     printf("\n");
63 }
64
65 /// Remove one element from head of queue
66 Nodes *pop(Queue **head)
67 {
68     if (!*head) {
69         printf("[WARN]\tNothing to pop, queue is empty\n");
70
71         return NULL;
72     } else {
73         Nodes *node;
74         node = (*head)->node;
75
76         Queue *tmp;                // tmp pointer to struct
77         tmp = (*head)->next;       // set pointer to 2nd element
78         free(*head);               // free memory for 1st element
79         *head = tmp;               // set 2nd element to 1st element
80
81         return node;
82     }
83 }
```

Appendix K

Flow Diagram


go()
Setup variables
declare *robot
initialise *robot
map_load()
read map from input
store map in memory
node_map_load()
read map from memory
generate node map
store node map in memory
path_calculate()
read node map from memory
pathfinding algorithm
store path in memory


while
not at finish
map_check()
scan()
scan==map?
No
update_map()
node_map_load()
path_calculate()
Yes
move_next()
read path
from memory
set
direction bits
execute
stepping
reset
direction bits
end while
end go()