Path Finding

- as used in Rescuing Robots -

Project Report

ED3-1-E17

Aalborg University
Electronics and Computer Engineering

We used LaTeXfor typesetting this report, Code::Blocks for prototyping the code and IAR-Workbench for programming the microcrontroller.

AALBORG UNIVERSITY
STUDENT REPORT

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

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: 47

Date of Completion:
December 18, 2017

Abstract:

This report This report is about path finding on a grid based map. To test our theoretical work, we decided to build a small four wheeled robot. Therefore we needed to think of a way to move the robot, a way to observe our surroundings and a way to manage the collected data.

Contents

Todo list

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 and the GitHub repository [1].

Aalborg University, December 18, 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 path-finding robot, with the purpose of bettering the groups 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 limits of this project, we adapted the RoboCup Junior Rescue rules [2]. Those limitations are mainly about the map and surface, and have been altered slightly, to fit the semesters requirements and available time.

After analysing the problem and our options, we came to the conclusion, that

analysis

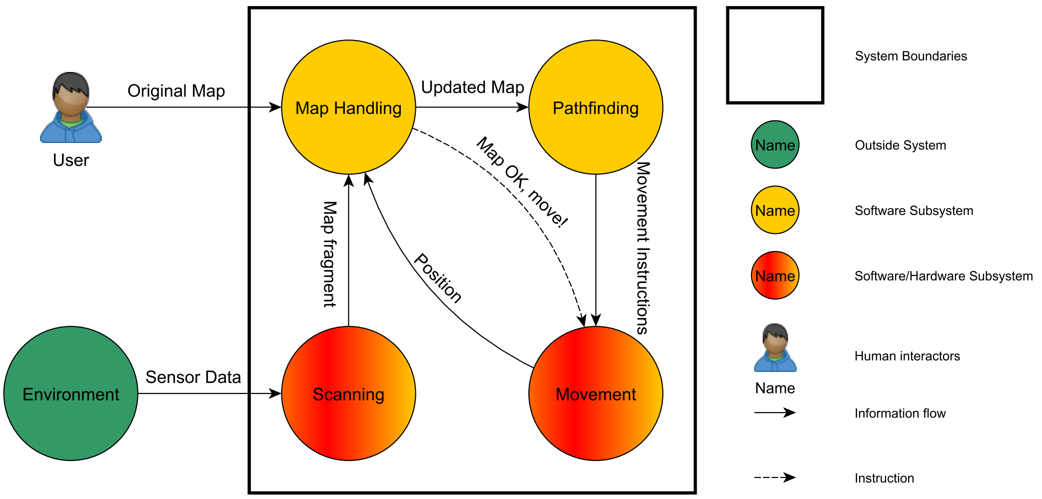


Figure 1.1: Overview over the Robot with Subsystems

a modular system, as described in Figure 1.1, would fit our needs best.

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

We were able to test our software solutions independent from any hardware,

something delivery problems

In the course of our project work, we encountered some problems with hardware delivery. Because we modularized our system, we were still able to work on it without having most of our hardware.

explain sectioning, based on the modules

this is how far I am with writing the introduction

The next paragraphs may contain some more information, don't know how to put them right now

In this project we want to talk about path finding algorithms, with the main focus of building an example implementation on a small scale.

We expect the reader to have a basic understanding of math, programming and simple physics. But will explain the applied topics.

write introduction, should include Problem statement, general idea,

Autonomous movement can be useful in rescue missions, where the area is too dangerous to send a human rescuer. Such environments could be for example: Burned down/burning houses, buildings struck by natural disaster or any other building with unknown structural integrity.

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.

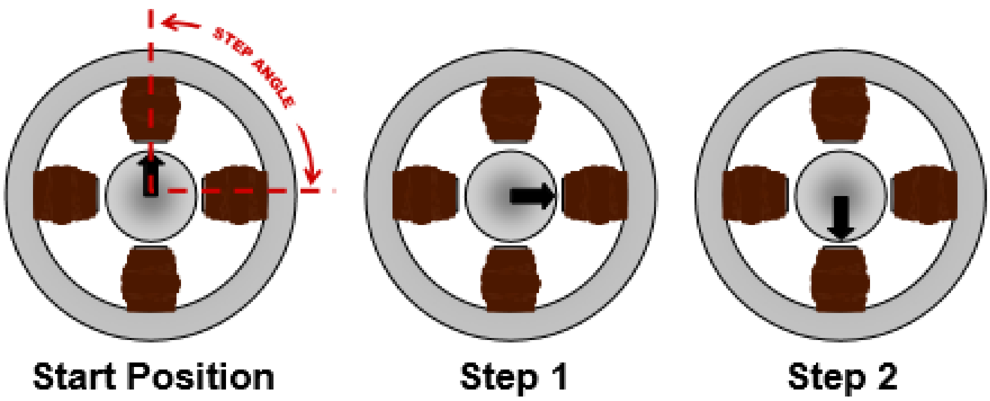


Figure 2.1: Step Angle

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.

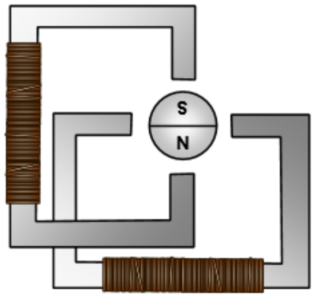
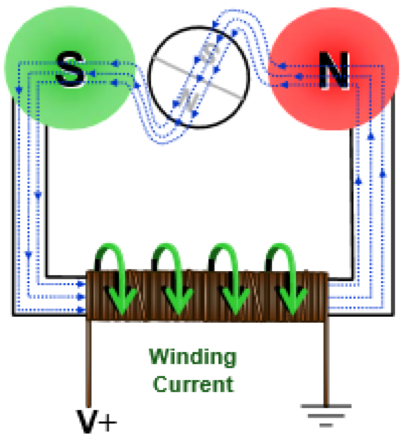
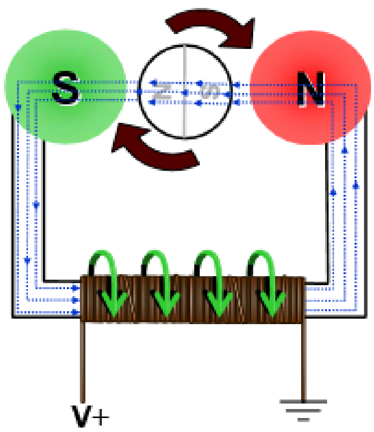


Figure 2.2: Main Components

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



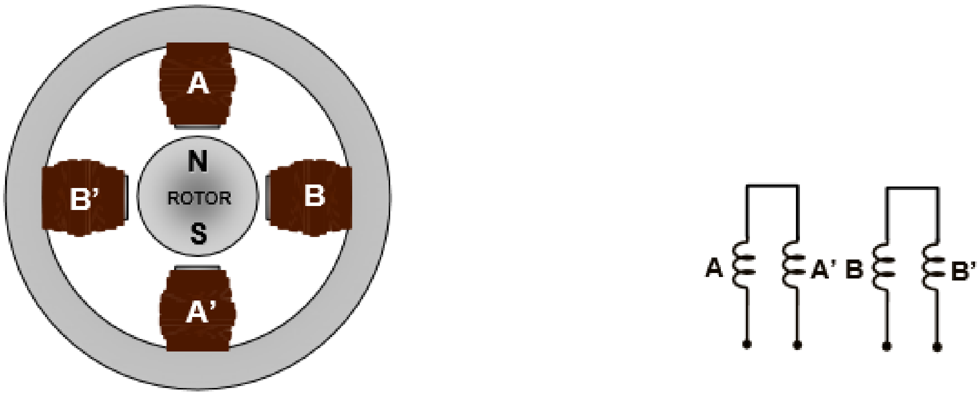
(b) Low Resistance

Figure 2.3: Direction of Magnetic Flux

2.1.1 Types of Stepper Motors

Permanent Magnet Motor

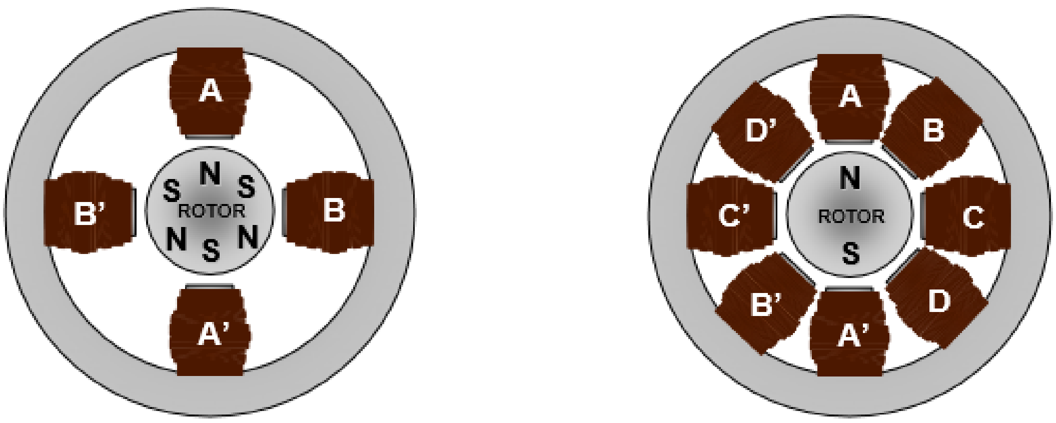
This type of stepper motor has a magnetized rotor. Each winding will be subdivided into two, to better understand how to motor functions. Figure 2.4 represents the windings, and how they are distributed inside a stepper motor.



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

Figure 2.4: Basic Structure of a Motor

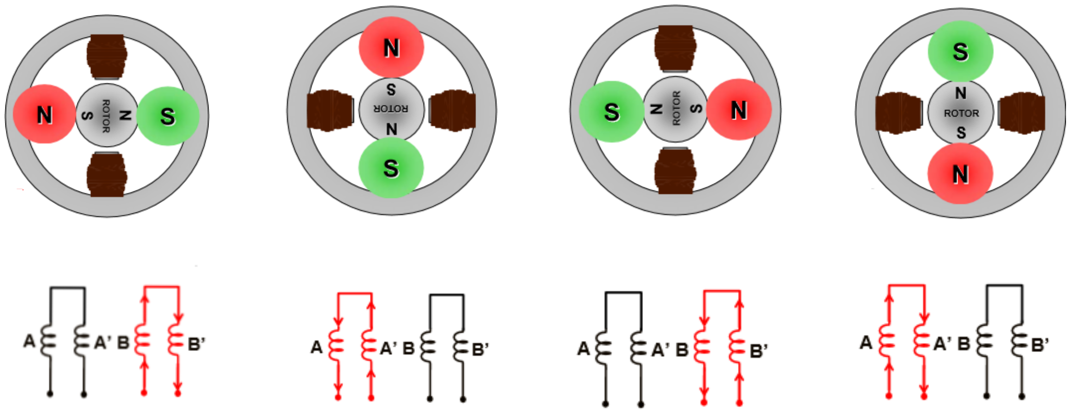
The resolution of the motor can be improved in two ways, either by increasing the number of pole pairs in the rotor itself, or by increasing the number of phases as shown in Figure 2.5.



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

Figure 2.5: Increased resolution

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

Figure 2.6: Stepping a Permanent Magnet Motor

Variable Reluctance Motor

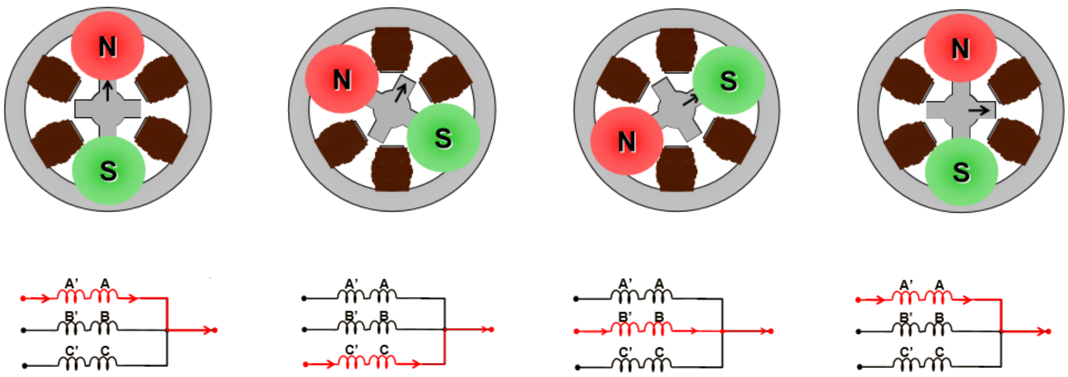
This type of motor uses a rotor that is not magnetized, and has a number of teeth as seen in Figure 2.7. The windings are configured differently, as depicted in 2.7(b), all having a common voltage source but separate ground connections. They usually have 3 or 5 windings. Greater precision can be achieved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

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

Figure 2.7: Variable Reluctance Motor Components

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

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.

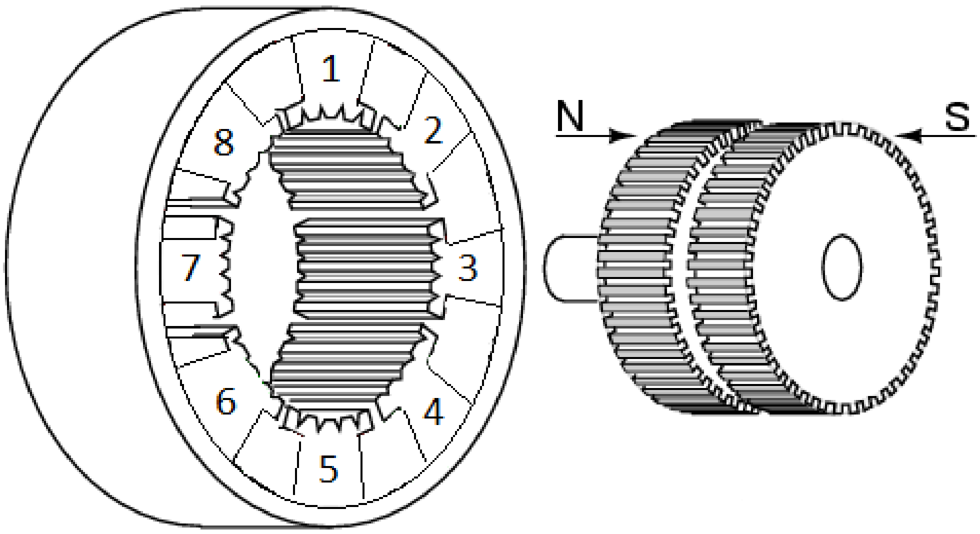
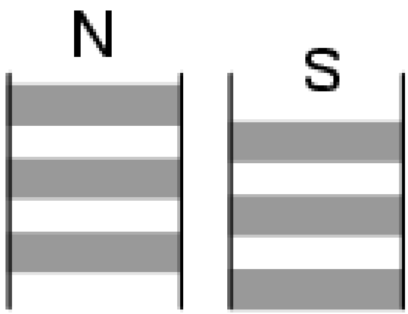
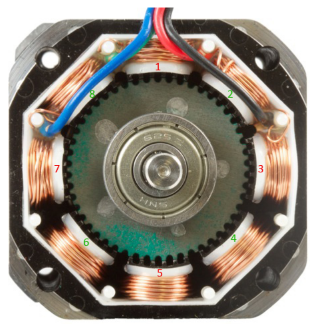


Figure 2.9: Stator and Rotor

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

Figure 2.10, the windings with numbers 1 and 5 are completely aligned with the teeth of the rotor. Windings number 3 and 7 are completely unaligned, 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.

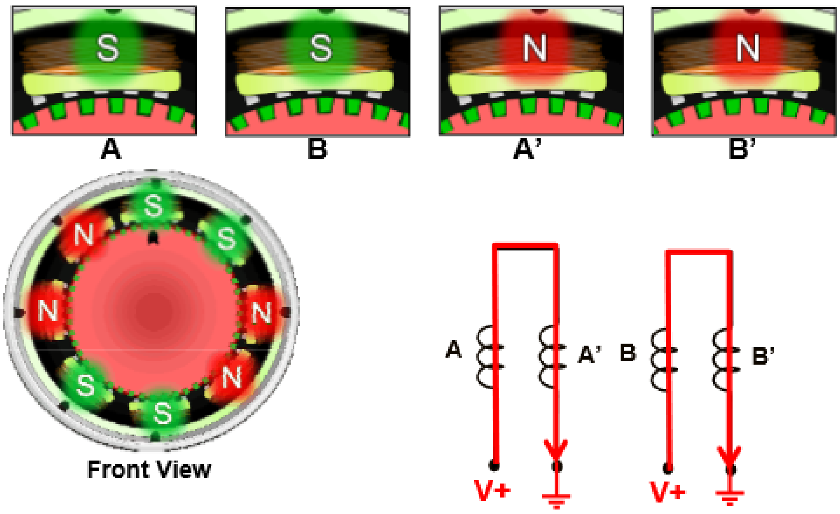


Figure 2.11: First Step

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.

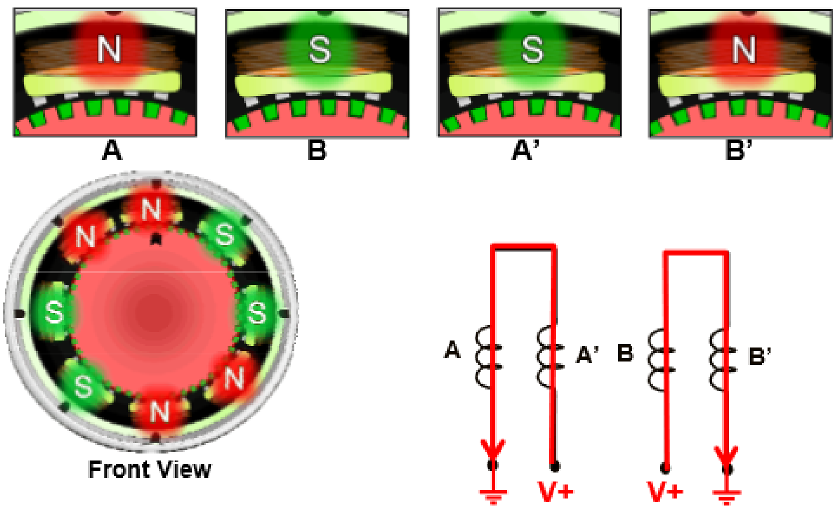


Figure 2.12: Second Step

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.

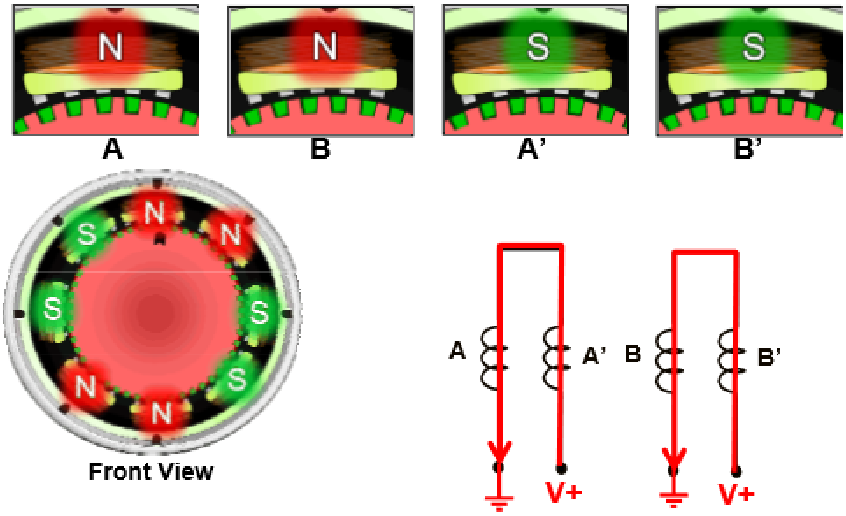


Figure 2.13: Third Step

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.

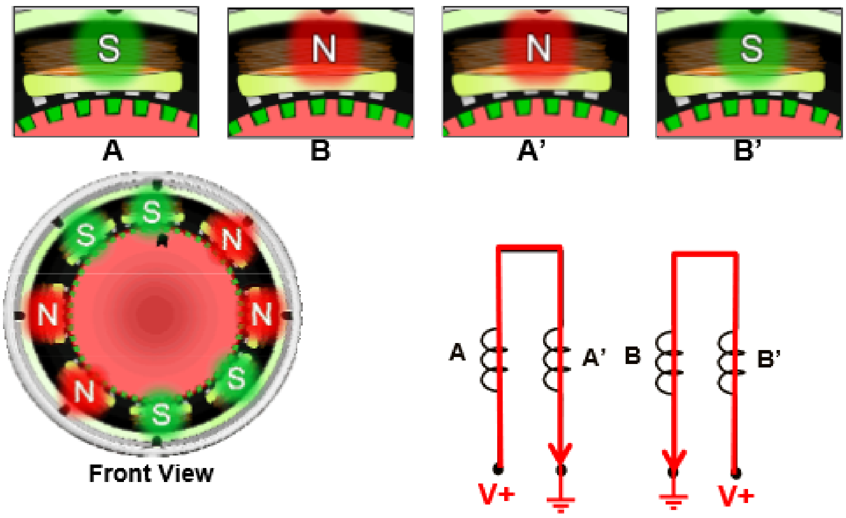


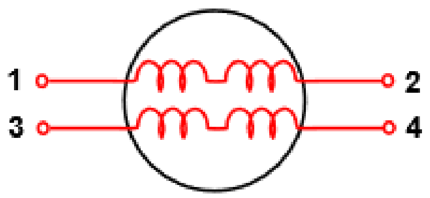
Figure 2.14: Forth Step

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.

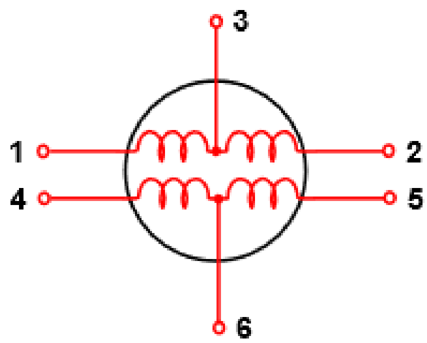
2.1.2 Unipolar And Bipolar Stepper Motors

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



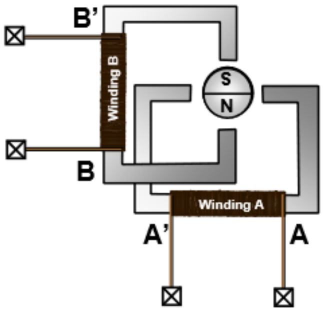
(b) Unipolar

Figure 2.15: Winding Configuration

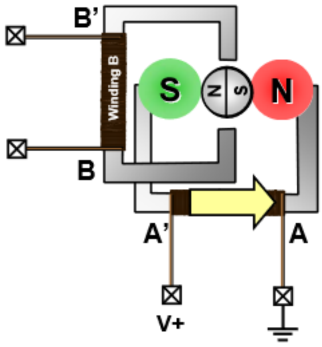
Bipolar stepper motors have one winding per phase. To energize the first phase, voltage needs to be applied to lead 1, and lead 2 needs to be connected to ground. Stepping the motor, involves energizing one phase, then the second, then the first phase again with reverse polarity, meaning the voltage 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.

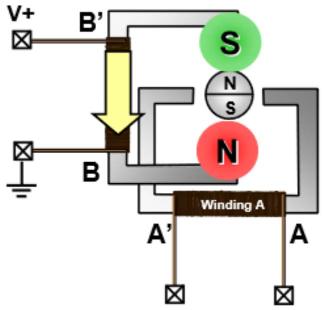
Note that the bipolar configuration as shown in Figure 2.16 allows the current to flow in both directions, but the voltage and ground continuously switch positions. This makes bipolar stepper motors a bit more complicated to drive, but as previously stated, motor driver boards simplify the task.



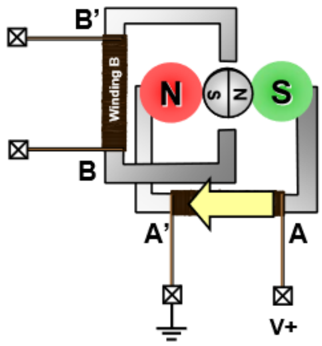
(a) 1st Step



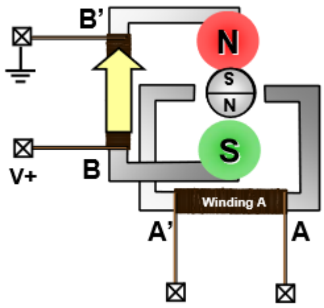
(b) 2nd Step



(c) 3rd Step



(d) 4th Step



(e) 5th Step

Figure 2.16: Bipolar Motor Spinning

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.

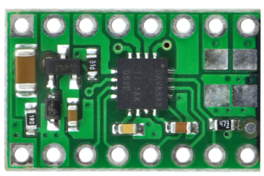


Figure 2.17: Driver Board

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.

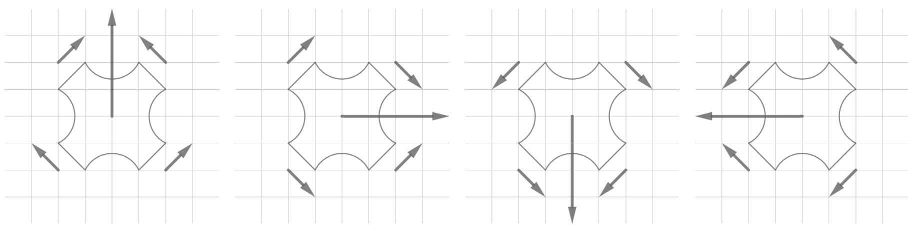
The key difference between omni-wheels and traditional wheels is their contact area. For omni-wheels it consists of smaller wheels that are able to move freely sideways, thus not generating any friction.

(a) Standard Wheel

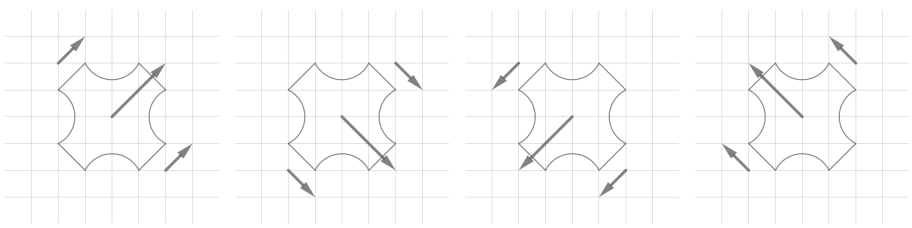
(b) Omni-Wheel

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.

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

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.

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 9 usable pins.

reference to msp

Figure 2.21 shows a schematic of the motor control circuit.

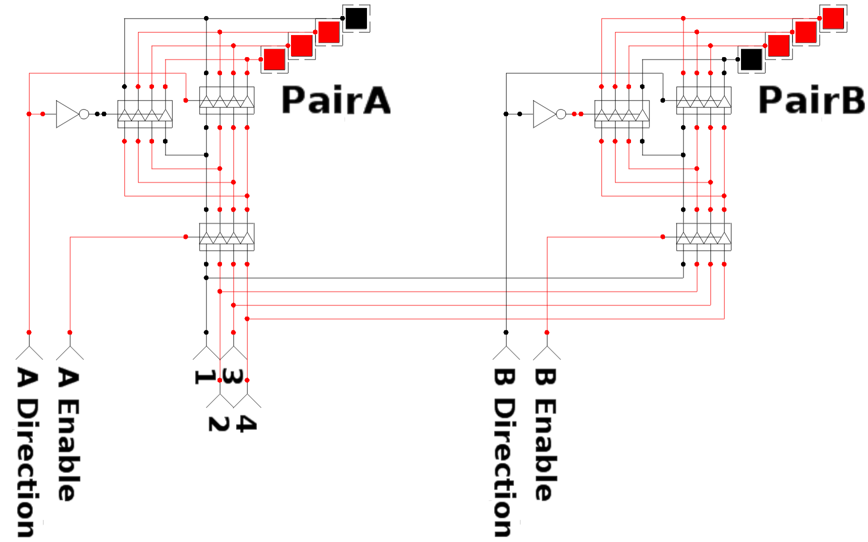


Figure 2.21: Motor Control Circuit

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.

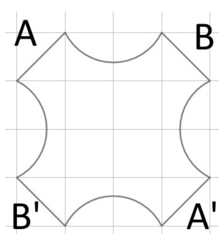


Figure 2.22: Motor Pairs

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.

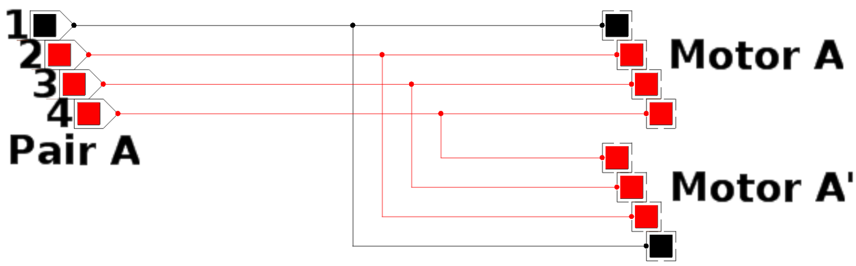


Figure 2.23: Motor Wiring

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

Chapter 3

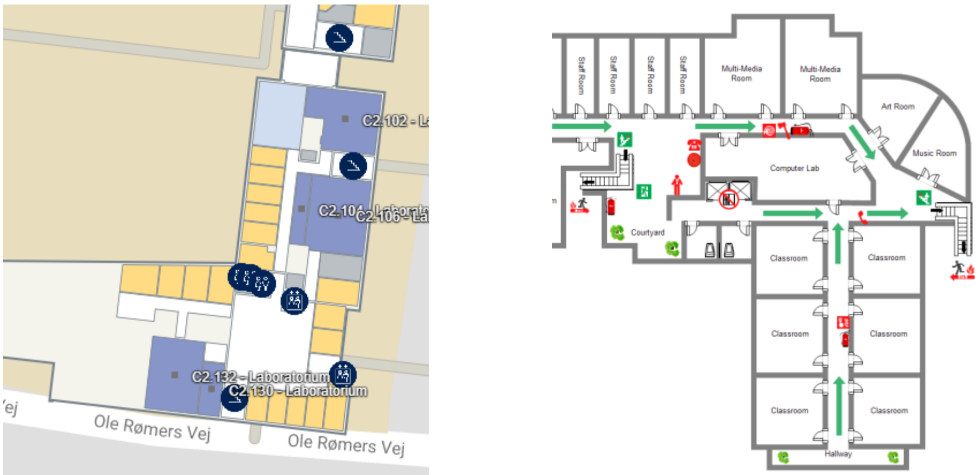
Map Handling

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 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 informations. 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 to be interpreted automatically. 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 the data the map should include, as well as some areas that have been delimited from.

Table 3.1: Map data

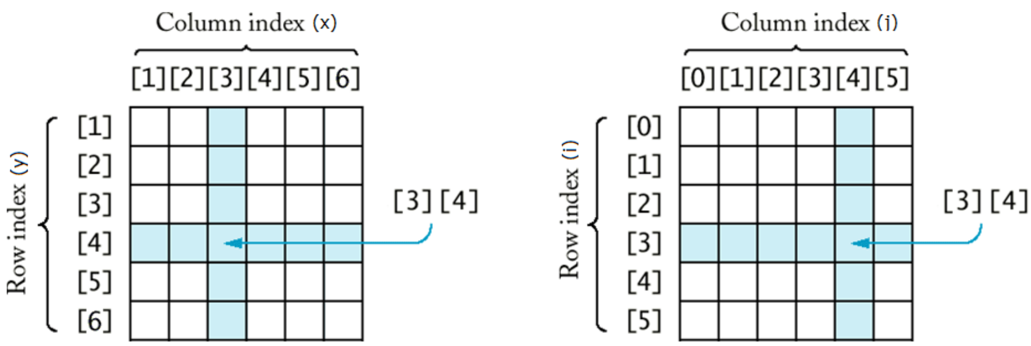
Data to be included	Data to delimit from
Map dimensions	Differences in height (levels, stairs etc.)
Start position	Door openings
Finish position	Ground surface (slipping, traction)
Walls	Objects

3.2 Map Coordinates

During the theoretical development we often used hand-drawn 2D maps with grids as depicted in Figure 3.2a. The map can have a certain size and allows for an object 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 2D arrays 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 3.2a and 3.2b.



(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 coordinate. To allow for a more logical access to segments of the map in the actual programming, we chose to start map coordinates at zero. Rows and columns could be switched when needed. This could be either be done in the syntax, by switching i and j, or by switching the x and y axes when storing the map in the first place.

Example of map segment before rows and columns are switched

,

```
Array: map[4][3], would give map coordinate (3,4)
```

Example of map segment after switching rows and columns

,

```
Array: map[3][4], would give map coordinate (3,4)
```

This lead to the final implementation in our program using 2D arrays, where the value of any given map segment easily could be accessed.
Example of implementation in the final code

,

```
Array: robot->map->segments[3][4], would return the value for map
    coordinate (3,4)
```

Examples of this can be seen implemented in the final code in appendix XX (some specific place?). The same approach was used for handling node maps, which is further explained in Chapter 5.

fix referring to code in appendix, and code language should not be set to Python.

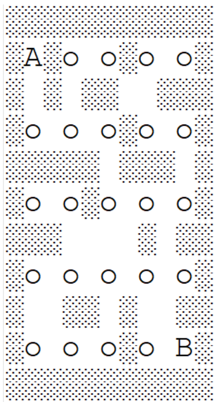
3.3 Map Design

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 5x5 map can be seen in Figure 3.3a, where # being walls, A being the start, B being the finish, o being nodes, and the white spaces being open spaces. Maps where also created using the UTF8 character encoding, which has a more characters to choose from, see Figure 3.3b.

Nodes represent positions on the map where the robot can move between. The idea is to allow movement in 8 directions, 4 straight and 4 diagonal, unless a direction is blocked by a wall. Nodes are explained further in Chapter 5.

```
###########
#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: 5x5 map

We found that UTF8 would require a bit more work, since text editors often 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 1-4 bytes [https://en.wikipedia.org/wiki/UTF-8]. 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 input file

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

The first idea was to use a fixed array large enough to hold maps of a predefined size. On the other hand, we thought this was as an excellent opportunity to try and work with pointers and dynamic memory allocation.

The required size of the array can be calculated by counting the lines and characters of the map, which is equivalent to the rows and columns. For a map with 5x5 nodes, there will be 11 characters in each line and a total of 11 lines. To store the values an 11x11 2D array is required.

Bunch of code could be inserted here

- open file and read content
- count map size (rows and cols)
- allocating 2d array
- saving data in 2d array
- saving start and finish position

3.5 Check Map

Remember we have a chapter dedicated to scan.
Based on scenario things might have dramatically changed, even to the point of map being useless or no map at all. Explain how map is updated.

3.6 Map Save

3.7 Walls / Neighbours hex

Maybe this should be part of map check

3.8 Node Map

Chapter 4

Scan

In an attempt to take it a step further, we decided to use IR distance sensors. Initially, our robot is supposed to find the shortest path from A to B, on a map. In rescue situations, often, the map provided does not match the actual outline, possibly due to different events that altered the outline, such as wall collapses, that might obstruct the movement in a certain direction.

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.

4.1 Sensor

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. [3]

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.

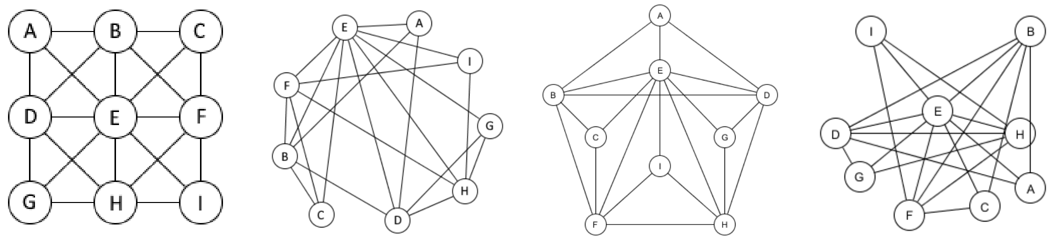
5.1 Graphs

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. [4]

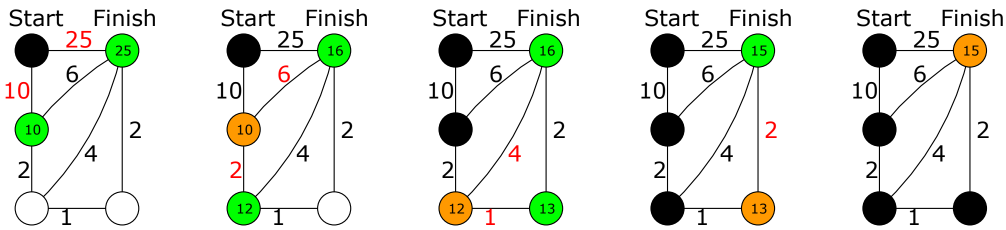
5.4 Dijkstra

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.

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. [5]



(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. [6]

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.

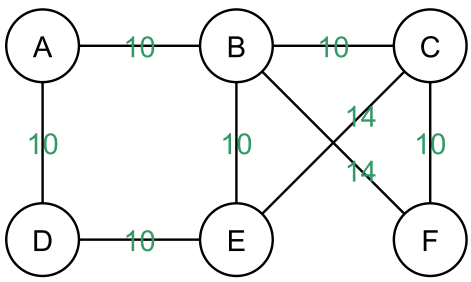


Figure 5.3: Graph with Labelled Edges

talk about A* and lead into heuristics

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 A.

Since our map consists of *nodes* with up to eight neighbours, as explained in Chapter **??**, 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.

does Troels explain this?

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

NE	SE	SW	NW	N	E	S	W	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`

```
,
#define N    0x01
#define E    0x02
```

```
#define S    0x04
#define W    0x08
#define NE   0x10
#define SE   0x20
#define SW   0x40
#define NW   0x80
```

does Troels describe this in ch:map?

After reading the map from the input file, as explained in Section **??**, 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`

,

```
    if (!(robot->map.node[i][j].walls & N))   //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 & E))   //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 & S))   //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 & W))   //WEST exists
      {robot->map.node[i][j].w=&robot->map.node[i][j-1];}
    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`

,

```
    robot->map.node[i][j].movecost = 0xFFF; //set movecost high
    robot->map.node[i][j].parent = NULL;    //point parent to NULL
  }
}
//------------------------END 2D for loop------------------------//
//set movecost for the current position to 0
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*. overwrite?

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`

```
while ((curx!=robot->map.finish.y)||(cury!=robot->map.finish.x)){
  if (currNode==NULL){
    printf("[WARN]\tsomething went wrong, current node is NULL\n");
    break;
  } //catches NULL pointers
  if (deadcount++>=(robot->map.nSize.x)*(robot->map.nSize.y)){
      printf("[ERROR]\tEVERYTHING WENT WRONG, looping\n"); return;
  } //catching infinite loops
  curx = currNode->position.x;
  cury = currNode->position.y;

  /*explaining the ifs below
  checks whether the neighbor exists, then if movecost is smaller
    updates neighbor movecost
    updates neighbor parent
    remove parent as neighbor
    pushes neighbor on queue
  */

  //------------------------------STRAIGHTS--------------------//
  if(currNode->n&&(currNode->n->movecost > currNode->movecost+10))
  { currNode->n->movecost = currNode->movecost+10;
    currNode->n->parent = currNode;
    currNode->n->s = NULL;
    push_queue(&robot->unchecked, currNode->n); }
```

After looking through all neighbours of a *node*, it can be pushed onto the `checked` stack. And the next node has 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`

```
  push_stack(&robot->checked, currNode);  //mark current as checked
  printf("[INFO]\tNode [%2d][%2d] computed!\n",curx,cury);

  currNode = pop(&robot->unchecked);  //get a new node from queue
}//---------------------------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`

```
 ,
  //-----------------------LOOP THROUGH STACK-----------------------//
  currNode=pop(&robot->checked);
  while(currNode->movecost!=0){ //start has movecost 0
    ownX = currNode->position.x;
    ownY = currNode->position.y;
    parX = currNode->parent->position.x;
    parY = currNode->parent->position.y;
    //----------------------Generate Movement---------------------//
    move=0;   //resets move, then adds all movements together
    if (ownX<parX) move+=N;          //Something North
    else if (ownX>parX) move+=S;     //Something South
    if (ownY>parY) move+=E;          //Something East
    else if (ownY<parY) move+=W;     //Something West

    //------------------Detect Diagonal Movement------------------//
    if (((move!=N)&&(move!=E)&&(move!=S)&&(move!=W))){
      if       (move==N+E) move=NE;  //North and East
      else if (move==S+E) move=SE;  //South and East
      else if (move==S+W) move=SW;  //South and West
      else if (move==N+W) move=NW;  //North and West
    }  //now we know the exact moving direction!
    //-------------------Save To Movement Stack-------------------//
    push_move_stack(&robot->movement, move);
    //------------------------ Find Parent------------------------//
    while (((parNode=pop(&robot->checked))->position.x!=parX)||
           (parNode->position.y!=parY)){}
    currNode=parNode;
  }
}
```

5.8 Perspective

5.8.1 Efficiency

Path-finding is the most resource-heavy calculation in the whole system, and will likely be executed several times while running. Because of that we wanted to at least think about and discuss efficiency.

Our current approach takes use of pointers a lot, often iterating through several substructs. Listing 5.7 shows code with multiple pointer access per line, varying from 2 to 4 for a single line. Similar code 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 5.7: 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 5.7, 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 explained in 5.8, uses the aforementioned `walls` byte, to compute in what directions to link.

Listing 5.8: Accessing several Fields through multiple Pointers in `path.c`

,

```
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 & N))   //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 & E))   //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 & S))   //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 & W))   //WEST exists
      {robot->map.node[i][j].w=&robot->map.node[i][j-1];}
    else {robot->map.node[i][j].w=NULL;}
```

Because our *nodes* are organized in a 2D-array, this includes array and pointer lookups, simple arithmetic and bitwise comparison to find both *nodes* and link them.

5.8.2 Design flaws

edges are defined twice, from both nodes

finish this sector

Chapter 6

Conclusion

In case you have questions, comments, suggestions or have found a bug, please do not hesitate to contact me. You can find my contact details below.

Jesper Kjær Nielsen
jkn@es.aau.dk
http://kom.aau.dk/~jkn
Fredrik Bajers Vej 7
9220 Aalborg Ø

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] 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.

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

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

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

Appendix A

Pathfinding

Dijkstra

Code

Listing A.1: `defs.h`

,

```c
/******************
 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 N   0x01
#define E   0x02
#define S   0x04
#define W   0x08
#define NE  0x10
#define SE  0x20
#define SW  0x40
#define NW  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;           // Hex value for the 8 walls
  int movecost;                  // Steps needed to get here
} Nodes;

typedef struct {
  Point start;                   //starting position
  Point finish;                  //finish position
  Point size;                    //amount of segments in the map
  //Point num_nodes;             // number of nodes in the map
  unsigned char **segments;      // 2D array of the map data from file
  Nodes **node;                  // 2D array of nodes
} Maps;

typedef struct element {
  Nodes *node;                   // Pointer to the map node
  struct element *next;          // next element in queue
} Queue, Stack;

typedef struct {
  Point pos;
  Maps map;
  Queue *unchecked;              // Head of queue for unchecked nodes
  Queue *checked;                // Head of stack for checked nodes
} Robot;

//--------------------------- FUNCTIONS ---------------------------//

// Robot
void go();
Robot *init_robot();

// Map
void map_load(Robot *robot);
void map_save(Robot *robot);
void map_check(Robot *robot);
void map_update(Robot *robot, char hex);
void node_map_load(Robot *robot); // node/map?
int robot_finished(Robot *robot);
void test_node_array(Robot *robot);

// Scan
unsigned char scan();

// Move
void move_next(Robot *robot);

// Priority queue
void pushQ(Queue **HoQ, Nodes *new_node); //add element on the stack
void pop(Queue *tq);                      //pops element from stack
void printQueue(Queue *tq);               //prints stack
```

```c
void emptyQueue(Queue *tq);                //pops whole stack
//void push(int queue, Nodes *node); // temp

// Pathfinding
void path_test(Robot *robot);
void path_set_neighbors(Robot *robot);
void path_calculate(Robot *robot);

// Debugging and print to screen
void robot_print(Robot *robot);
```

Listing A.2: `path.c`

,

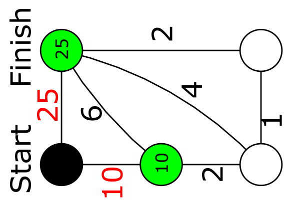
```c
#include "defs.h"

///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 & N))   //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 & E))   //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 & S))   //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 & W))   //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 & NE))  //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 & SE))  //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 & SW))  //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 & NW))  //NW exists
        {robot->map.node[i][j].nw=&robot->map.node[i-1][j-1];}
      else {robot->map.node[i][j].nw=NULL;}

      robot->map.node[i][j].movecost = 0xFFF; //set movecost high
      robot->map.node[i][j].parent = NULL;    //point parent to NULL
    }
```

```
  }
  //-------------------------END 2D for loop------------------------//
  //set movecost for the current position to 0
  robot->map.node[robot->pos.x][robot->pos.y].movecost = 0;
  printf("[INFO]\tDone linking nodes to neighbors\n");
}

///calculates the path from the current position
void path_calculate(Robot *robot) {
  //---------------------------- SETUP -----------------------------//
  //declare all variables needed in scope
  int curx,cury;           //keeps track of position on the map
  Nodes *currNode;         //the Node currently looked at

  path_set_neighbors(robot);  //make sure that all nodes are set up

  //-------------------------SETUP FOR CALC------------------------//
  curx = robot->pos.x;     //start calculating from current position
  cury = robot->pos.y;
  push_queue(&robot->unchecked, &robot->map.node[curx][cury]);
  currNode =pop(&robot->unchecked);

  //------------------------------CALC------------------------------//
  int deadcount=0;
  printf("\n\n[INFO]\tstarted path calculation\n\n");
  //----------------------WHILE not at finish----------------------//
  while ((curx!=robot->map.finish.y)||(cury!=robot->map.finish.x)){
    if (currNode==NULL){
      printf("[WARN]\tsomething went wrong, current node is NULL\n");
      break;
    } //catches NULL pointers
    if (deadcount++>=(robot->map.nSize.x)*(robot->map.nSize.y)){
        printf("[ERROR]\tEVERYTHING WENT WRONG, looping\n"); return;
    } //catching infinite loops
    curx = currNode->position.x;
    cury = currNode->position.y;

    /*explaining the ifs below
    checks whether the neighbor exists, then if movecost is smaller
      updates neighbor movecost
      updates neighbor parent
      remove parent as neighbor
      pushes neighbor on queue
    */

    //-------------------------------STRAIGHTS--------------------//
    if(currNode->n&&(currNode->n->movecost > currNode->movecost+10))
    { currNode->n->movecost = currNode->movecost+10;
      currNode->n->parent = currNode;
      currNode->n->s = NULL;
      push_queue(&robot->unchecked, currNode->n); }
```

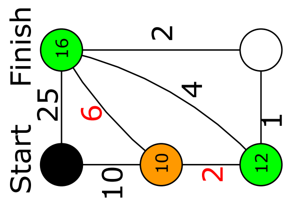
```
    if(currNode->e&&(currNode->e->movecost > currNode->movecost+10))
    { currNode->e->movecost = currNode->movecost+10;
      currNode->e->parent = currNode;
      currNode->e->w = NULL;
      push_queue(&robot->unchecked, currNode->e); }
    if(currNode->s&&(currNode->s->movecost > currNode->movecost+10))
    { currNode->s->movecost = currNode->movecost+10;
      currNode->s->parent = currNode;
      currNode->s->n = NULL;
      push_queue(&robot->unchecked, currNode->s); }
    if(currNode->w&&(currNode->w->movecost > currNode->movecost+10))
    { currNode->w->movecost = currNode->movecost+10;
      currNode->w->parent = currNode;
      currNode->w->e = NULL;
      push_queue(&robot->unchecked, currNode->w); }

    //------------------------------DIAGONALS--------------------//
    if(currNode->ne&&(currNode->ne->movecost > currNode->movecost+14))
    { currNode->ne->movecost = currNode->movecost+14;
      currNode->ne->parent = currNode;
      currNode->ne->sw = NULL;
      push_queue(&robot->unchecked, currNode->ne); }
    if(currNode->se&&(currNode->se->movecost > currNode->movecost+14))
    { currNode->se->movecost = currNode->movecost+14;
      currNode->se->parent = currNode;
      currNode->se->nw = NULL;
      push_queue(&robot->unchecked, currNode->se); }
    if(currNode->sw&&(currNode->sw->movecost > currNode->movecost+14))
    { currNode->sw->movecost = currNode->movecost+14;
      currNode->sw->parent = currNode;
      currNode->sw->ne = NULL;
      push_queue(&robot->unchecked, currNode->sw); }
    if(currNode->nw&&(currNode->nw->movecost > currNode->movecost+14))
    { currNode->nw->movecost = currNode->movecost+14;
      currNode->nw->parent = currNode;
      currNode->nw->se = NULL;
      push_queue(&robot->unchecked, currNode->nw); }

    push_stack(&robot->checked, currNode);  //mark current as checked
    printf("[INFO]\tNode [%2d][%2d] computed!\n",curx,cury);

    currNode = pop(&robot->unchecked);  //get a new node from queue
  }//---------------------------END WHILE---------------------------//

  printf("\n[INFO]\tDone calculating path!\n");
  path_calculate_movement(robot);
}

///calculates the movement stack out of the checked stack
void path_calculate_movement(Robot *robot){
  //pop from stack until start is reached
```

```
  //--------------------------- VARIABLES---------------------------//
  int parX=0,parY=0;
  int ownX=0,ownY=0;
  char move=0;
  Nodes *currNode = NULL, *parNode = NULL;
  //-----------------------LOOP THROUGH STACK-----------------------//
  currNode=pop(&robot->checked);
  while(currNode->movecost!=0){ //start has movecost 0
    ownX = currNode->position.x;
    ownY = currNode->position.y;
    parX = currNode->parent->position.x;
    parY = currNode->parent->position.y;
    //----------------------Generate Movement----------------------//
    move=0;   //resets move, then adds all movements together
    if (ownX<parX) move+=N;          //Something North
    else if (ownX>parX) move+=S;     //Something South
    if (ownY>parY) move+=E;          //Something East
    else if (ownY<parY) move+=W;     //Something West

    //------------------Detect Diagonal Movement------------------//
    if (((move!=N)&&(move!=E)&&(move!=S)&&(move!=W))){
      if      (move==N+E) move=NE;  //North and East
      else if (move==S+E) move=SE;  //South and East
      else if (move==S+W) move=SW;  //South and West
      else if (move==N+W) move=NW;  //North and West
    }   //now we know the exact moving direction!
    //-------------------Save To Movement Stack-------------------//
    push_move_stack(&robot->movement, move);
    //------------------------ Find Parent------------------------//
    while (((parNode=pop(&robot->checked))->position.x!=parX)||
           (parNode->position.y!=parY)){}
    currNode=parNode;
  }
}
```

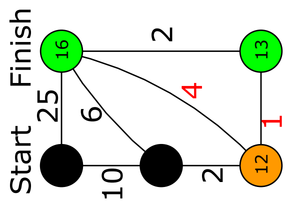
Algorithm



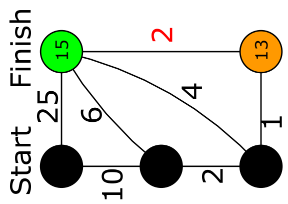
(a) 1. Step



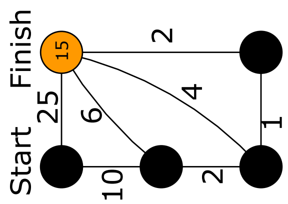
(b) 2. Step



(c) 3. Step



(d) 4. Step



(e) 5. Step