# Path Finding

- as used in Rescuing Robots -

Project Report

1-ED3

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:**
Analog Instrumentation

**Project Period:**
Fall Semester 2017

**Project Group:**
1-ED3

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

**Supervisor(s):**
Akbar Hussain

**Copies:** 1

**Page Numbers:** 36

**Date of Completion:**
December 13, 2017

**Abstract:**

This project 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 refers to the three co-authors listed below.

Aalborg University, December 13, 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

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,

how to reference to another chapter: Read more about path finding in Chapter 5.

## 1.1 Examples

You can also have examples in your document such as in example 1.1.

**Example 1.1 (An Example of an Example)**
Here is an example with some math

$$0 = \exp(i\pi) + 1 \ . \tag{1.1}$$

You can adjust the colour and the line width in the `macros.tex` file.

## 1.2 How Does Sections, Subsections, and Subsections Look?

Well, like this

### 1.2.1 This is a Subsection

and this

### This is a Subsubsection

and this.

**A Paragraph** You can also use paragraph titles which look like this.

**A Subparagraph** Moreover, you can also use subparagraph titles which look like this. They have a small indentation as opposed to the paragraph titles.

Is it possible to add a subsubparagraph?

I think that a summary of this exciting chapter should be added.

# 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.

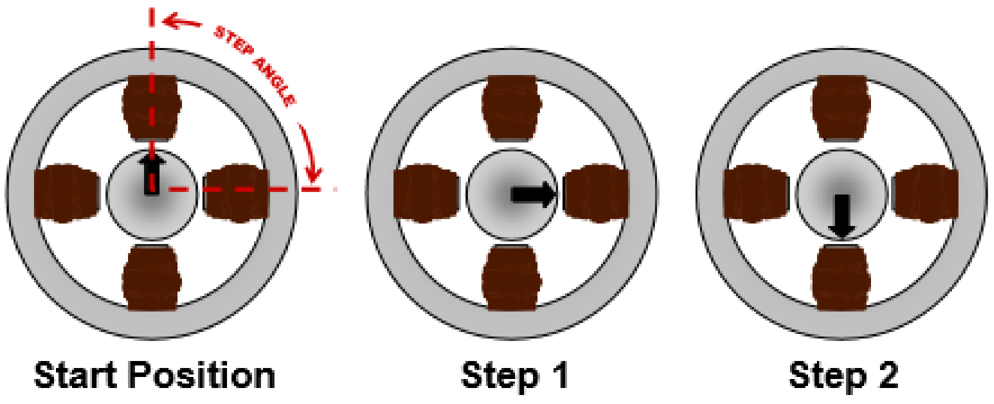


**Figure 2.1:** Step Angle

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.

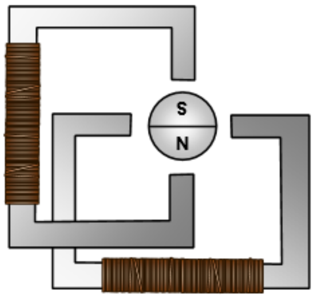
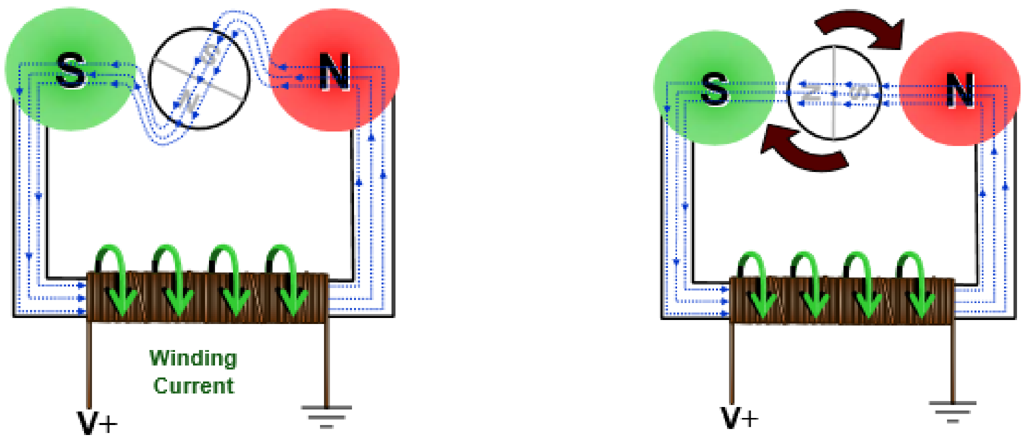


**Figure 2.2:** Main Components

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

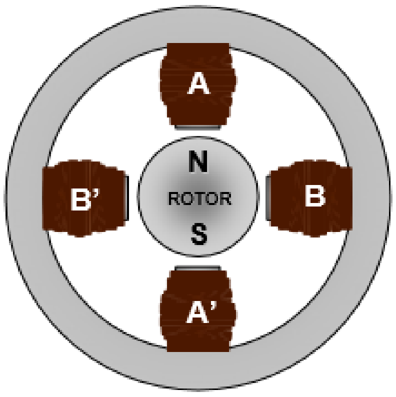
**(b)** Low Resistance

**Figure 2.3:** Direction of Magnetic Flux

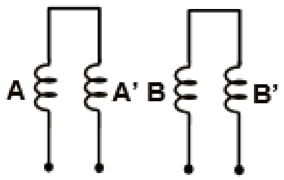
### 2.1.1 Types of Stepper Motors

#### 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.



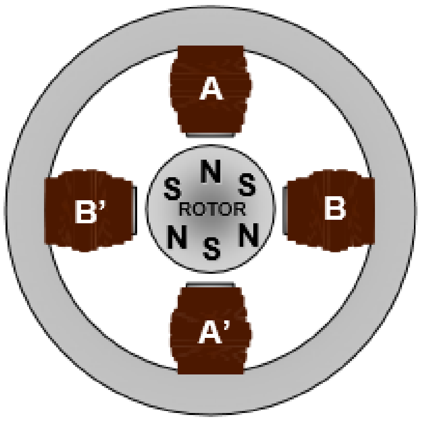
**(a)** Rotor



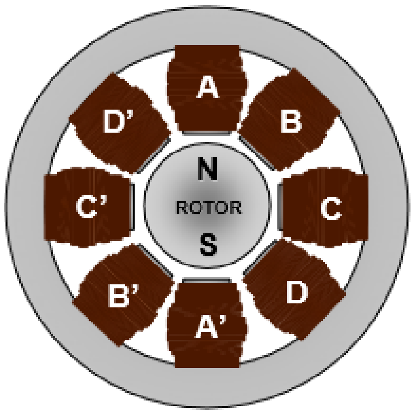
**(b)** Winding

**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.



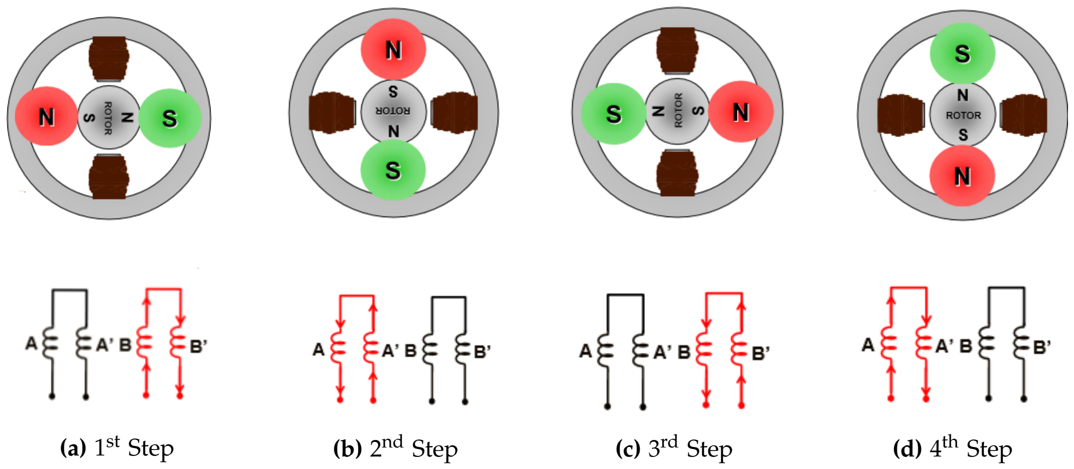
**(a)** Increased Pole Pairs



**(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.

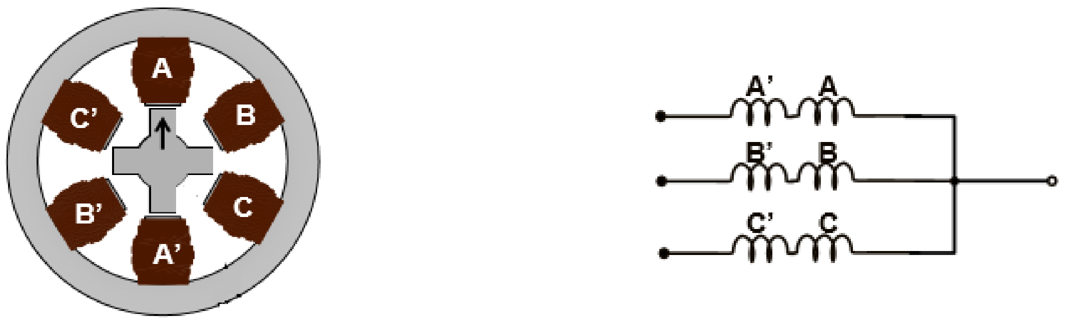


**(a)** 1st Step **(b)** 2nd Step **(c)** 3rd Step **(d)** 4th Step

**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.

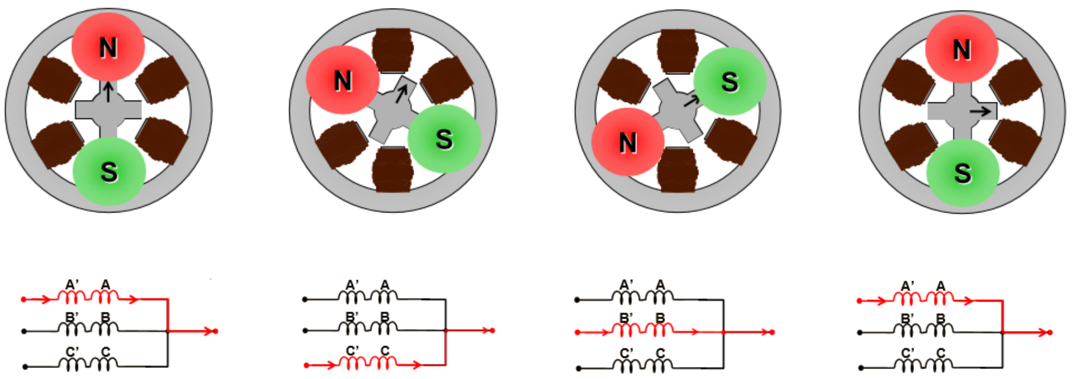


**(a)** Non Magnetized Rotor **(b)** Windings

**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.

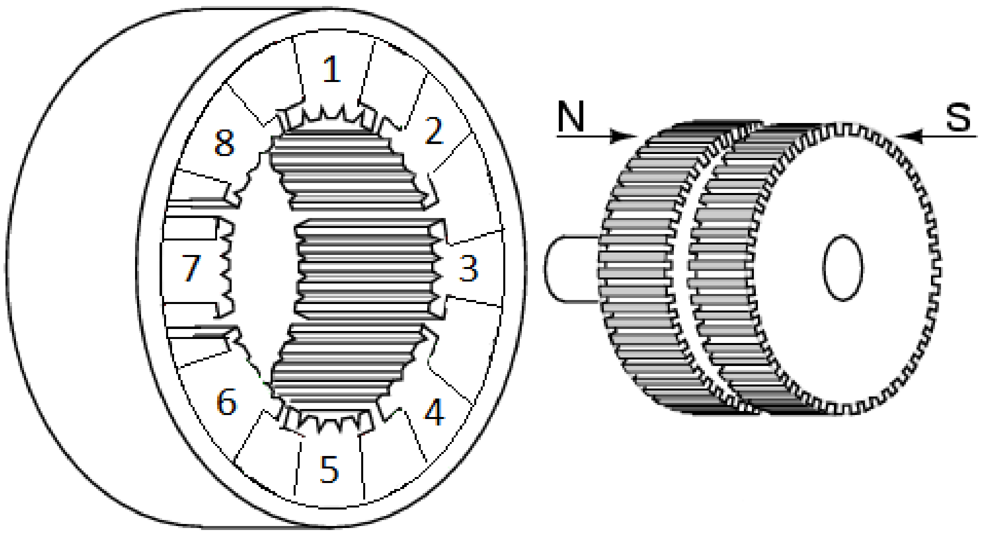
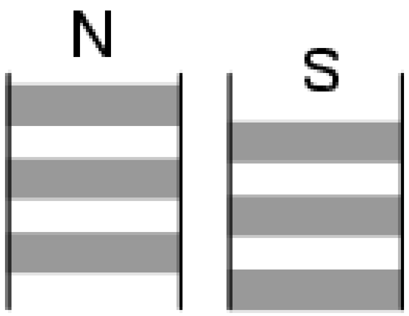
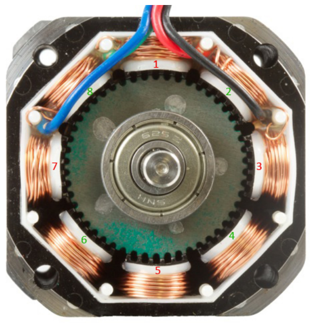


**Figure 2.9:** Stator and Rotor

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.

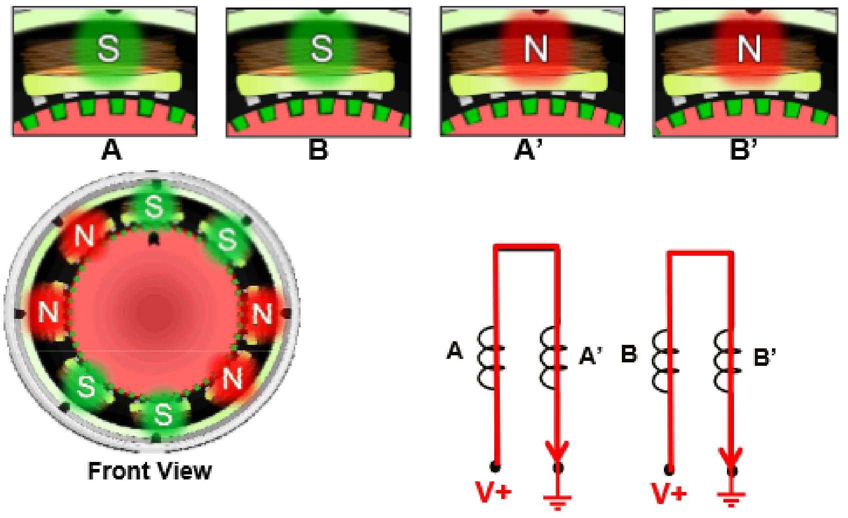


**Figure 2.11:** First Step

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.

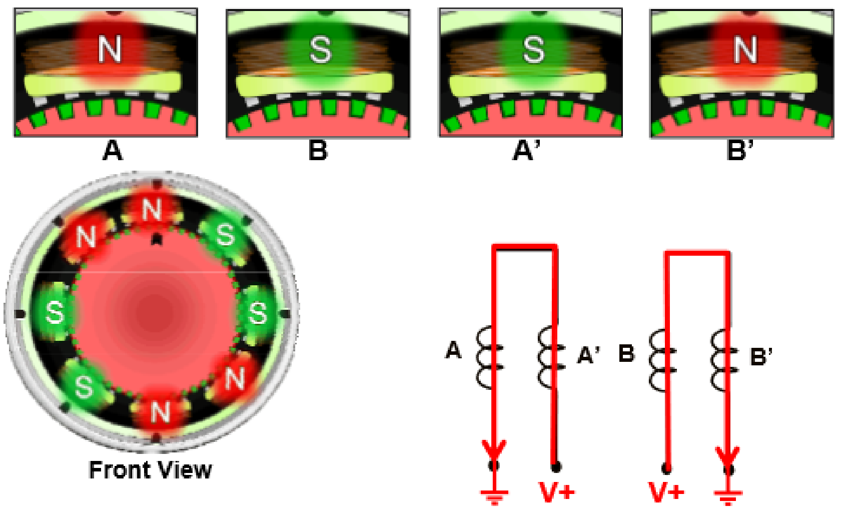


**Figure 2.12:** Second Step

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.

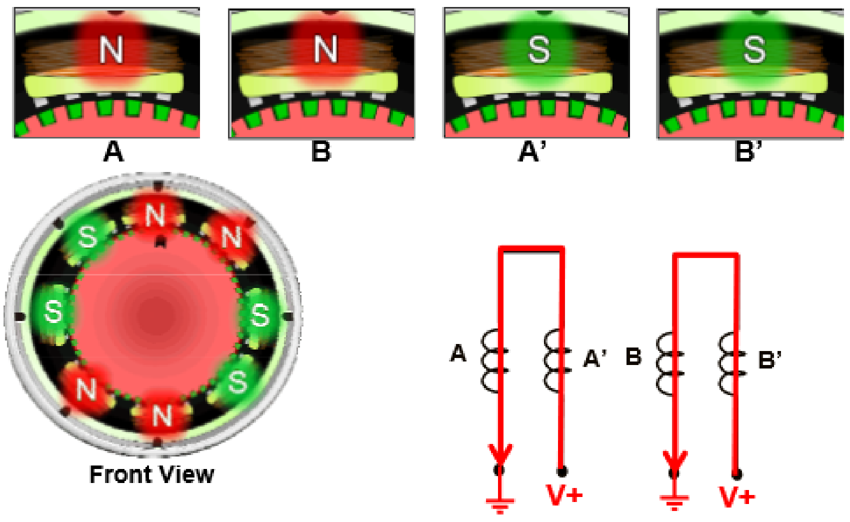


**Figure 2.13:** Third Step

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.

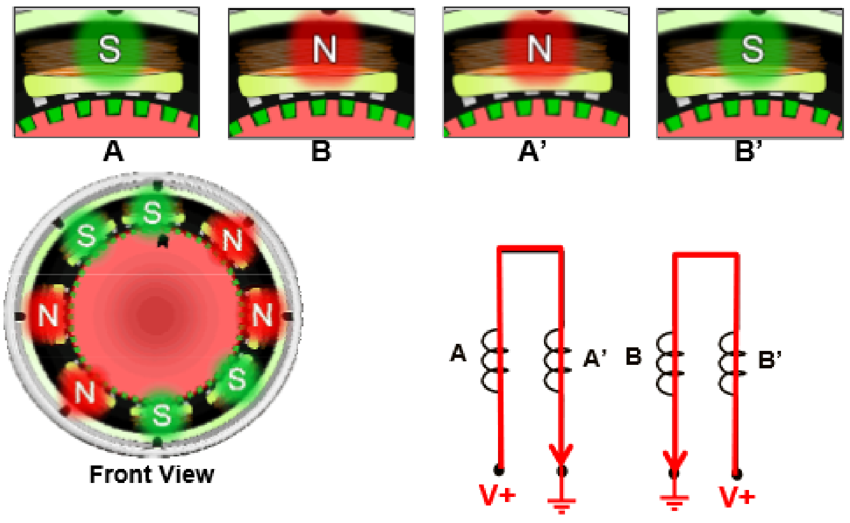


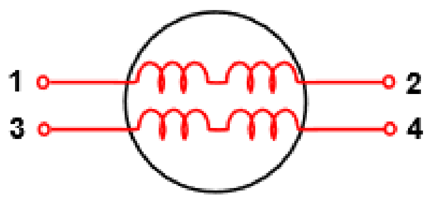
**Figure 2.14:** Forth Step

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.

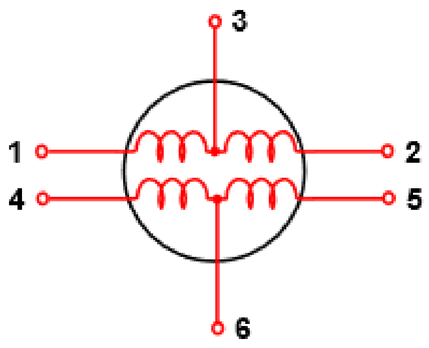
### 2.1.2 Unipolar And Bipolar Stepper Motors

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



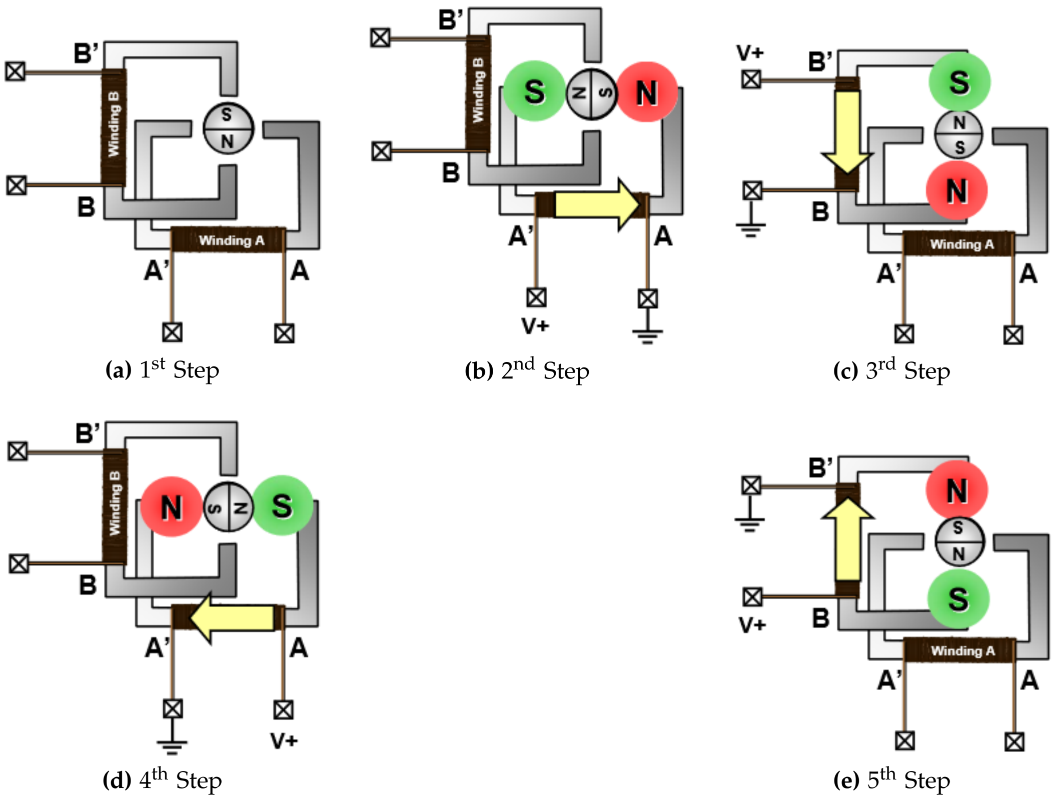
**(b)** Unipolar

**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.

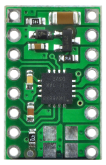
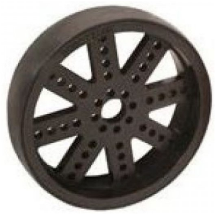


**Figure 2.17:** Driver Board

## 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

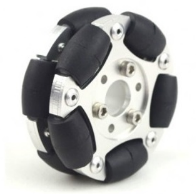
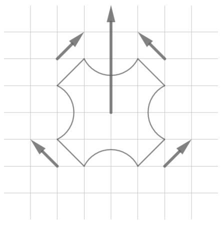
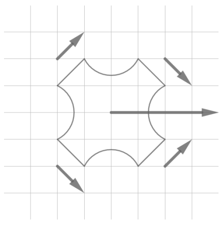
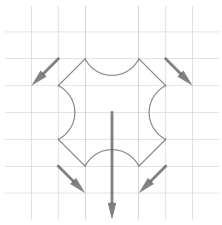

**(b)** Omni-Wheel

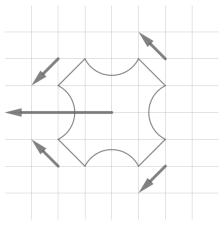
**Figure 2.18:** Wheels

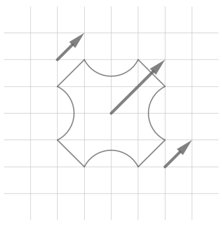
By mounting the wheels in pairs, with the shafts crossing at a 90° angle, we are able to move the robot in any direction without needing to change the orientation of the robot. This is achieved through rotating the pairs as shown in figure 2.19.

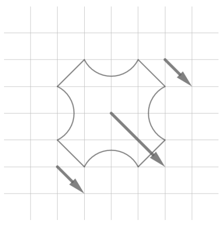

**(a)** North

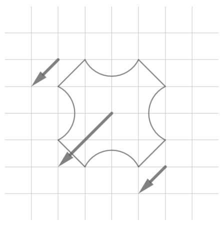

**(b)** East


**(c)** South


**(d)** West


**(e)** North-East


**(f)** South-East


**(g)** South-West

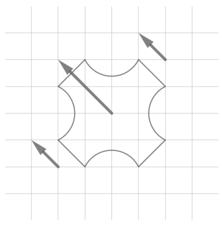

**(h)** North-West

**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.

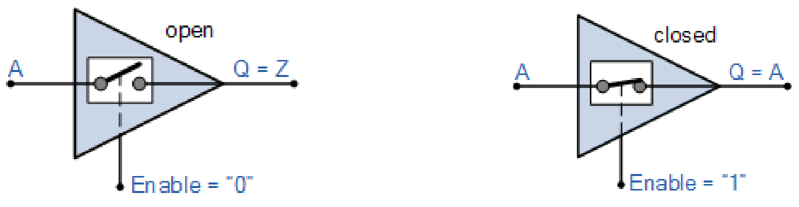
We decided to build our own circuit using tri-state buffers instead. The circuit will be explained after a short explanation of tri-state buffers.

### 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 $\mu$C 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. Giving the motors needed for moving in one direction, the stepping sequence at the same time was a must, and that was also a reason for needing 16 pins.

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 a different microcontroller in the beginning, specifically the MSP430, which had fewer pins available.

Figure 2.21 represents a schematic of the movement circuit. We have used 6 Tri-State buffers, 4 active high, and 2 active low.

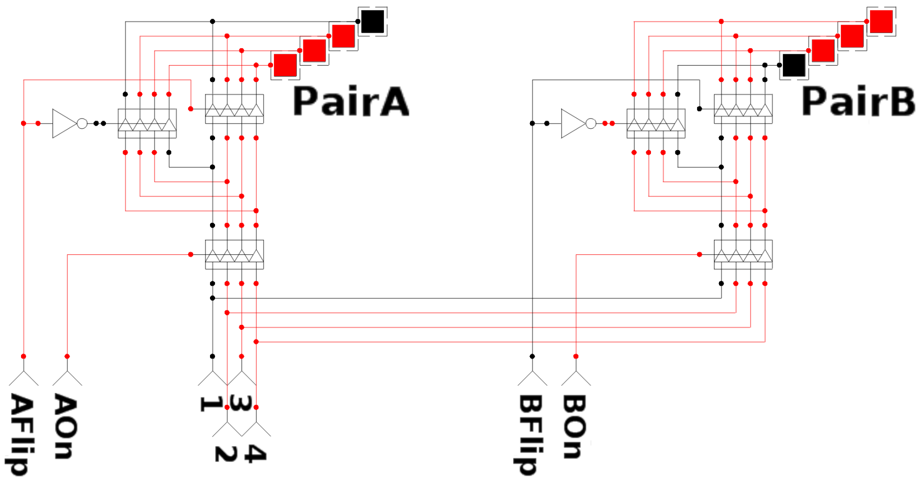


**Figure 2.21:** Motor Control Circuit

The 4 wires coming from the microcontroller, labeled ’1’, ’2’, ’3’ and ’4’ correspond to the stepping sequence. They are each connected to 2 Tri-State Buffer corresponding to each pair of motors. One pair of motors consists of 2 motors. The pairs are represented in Figure 2.22.

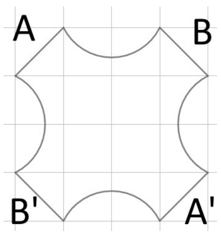


**Figure 2.22:** Motor Pairs

The first two Tri-State buffers, decide whether the pair is needed for moving in a specific direction. When moving diagonally, only when pair of motors is needed at a time. Deciding which direction the motors should spin in, is up to the following 2 Tri-State buffers, one active-low and one active-high. Giving a 'High' signal, activates the active-high buffer, and deactivates the active-low buffer. Giving a 'Low' signal, activates the active-low buffer, and deactivates the active-high buffer.

When spinning in one direction, the two opposite motors that make a pair, spin in opposite directions. For example, when moving North-East, only Pair A moves. Motor labeled A spins counter-clockwise, and motor labeled A' moves clockwise. This was a necessity, considering the placement of the wheels on the robot, and it was done by wiring one motor the other way around in respect to the other.

When spinning in one direction, the two opposite motors that make a pair, spin in opposite directions. This is achieved by wiring the motors in reverse. Figure 2.23 represents the wiring of the motors.

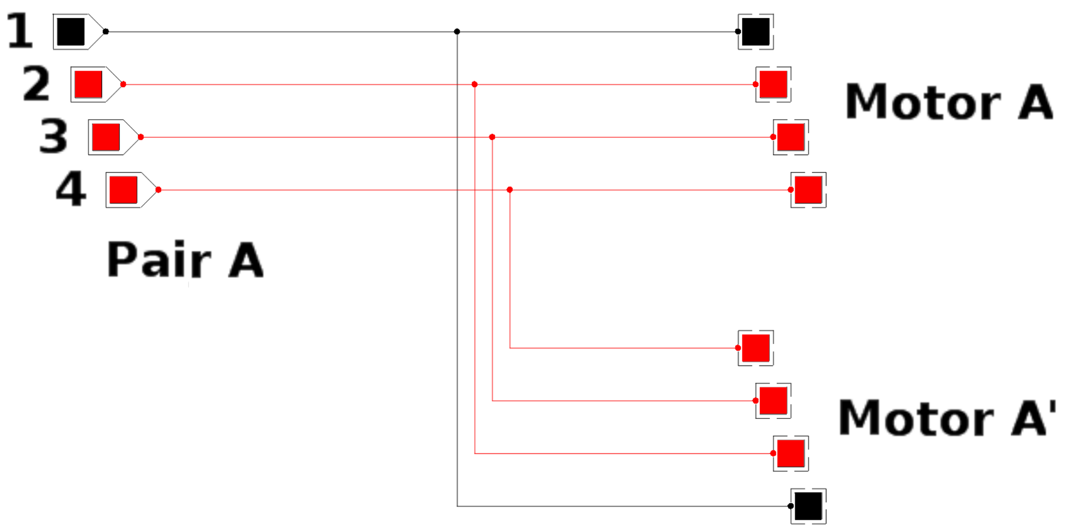


**Figure 2.23:** Motor Wiring in Reverse

| 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 |

| Direction | Aon | Aflip | Bon | Bflip |
|---|---|---|---|---|
| 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 | 1 |
| South-East | 0 | 1 | 1 | 0 |
| South-West | 1 | 1 | 0 | 1 |
| North-West | 0 | 1 | 1 | 1 |

**Table 2.1:** Directions

# Chapter 3

# Map handling

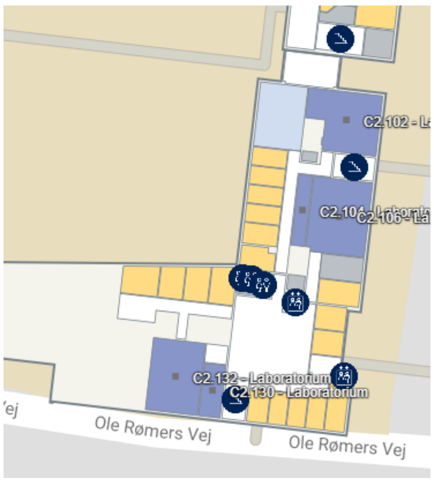
The rescue robot should be able follow a given path from start to finish, based on a predefined map. A map provides useful information about whether areas of the map are accessible or not. This is an important element in path-finding. Map data can be given to the robot prior to its physical presence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

A structured way of storing the required map data for different maps was designed. The goal was to not only make it readable by the microcontroller, but also allow easy user input.

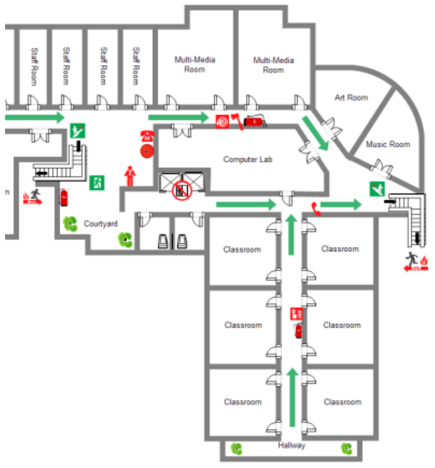
## 3.1 Map requirements

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

Such 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

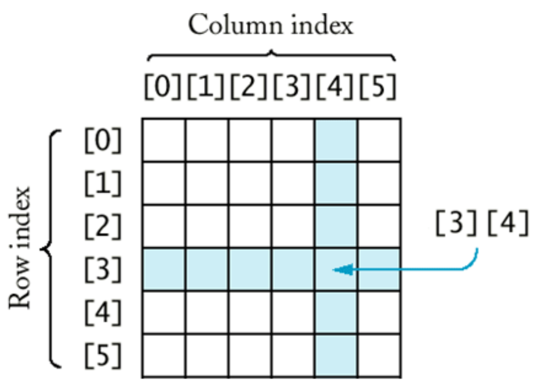
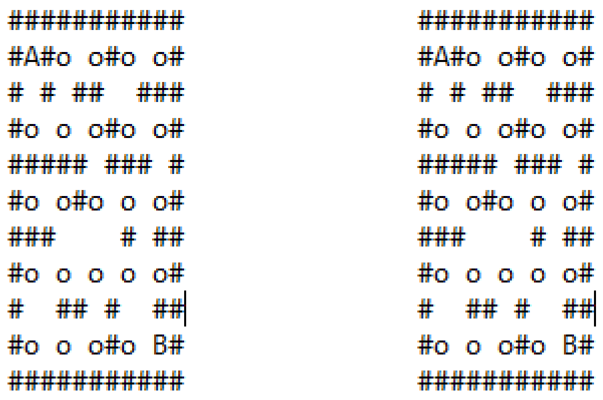


**Figure 3.2:** 2D array

## 3.3 Map design

Show the map + wiki
Explain we made the map size dynamic to handle any map size Explain how to store wall as hex value?



```
###########
#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

```
###########
#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#
###########
```

**(b)** UTF8

**Figure 3.3:** 5x5 map

## 3.4 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. Explain how map is updated.

## 3.5 Implementation

Code here, or parts of code under each section?

Like with many others, is the first step in Dijkstra's algorithm to reduce the map to the necessary minimum. After this reduction, the map only consists of *nodes* and *edges*. An edge connects two nodes together and has one integer *travel cost*. 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).

# Chapter 4

# Scan

Pathfinding is generally the process of finding a path from a given starting point ('A') to a given destination ('B'), on a given map.

## 4.1 Section

Like with many others, is the first step in Dijkstra's algorithm to reduce the map to the necessary minimum. After this reduction, the map only consists of *nodes* and *edges*. An edge connects two nodes together and has one integer *travel cost*. 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).

# 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 3.

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. [1]

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 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.

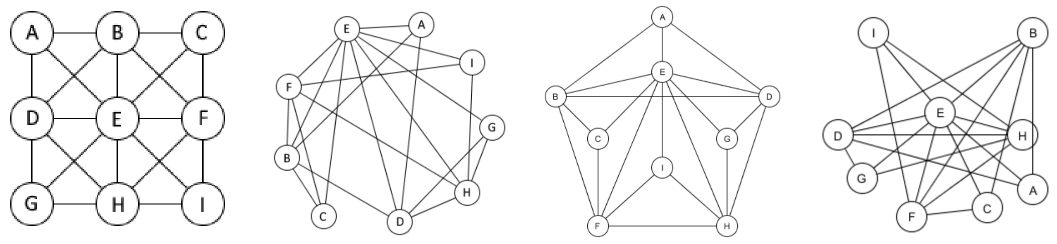
## 5.1 Graphs

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. [2]

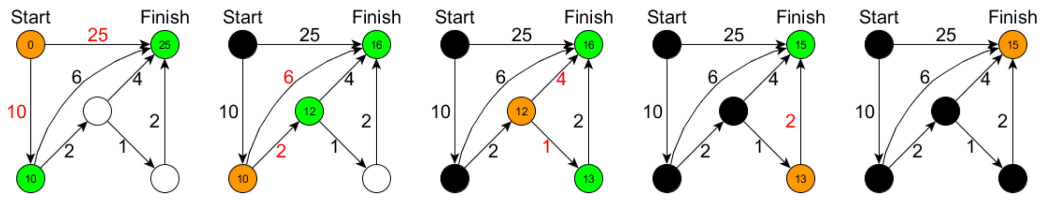
## 5.4 Dijkstra

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 the travel cost 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.



**(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 another value to each *node*. This value often gets called *heuristic*. Just like in Dijkstra, the *cost to reach* gets calculated iteratively, with the same iteration method. But in A* there is also another

## 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 costs. Figure 5.3 shows the similarities between several connections on a grid-based graph. because of this, the Dijkstra algorithm is losing its major advantage over a simple flood fill.

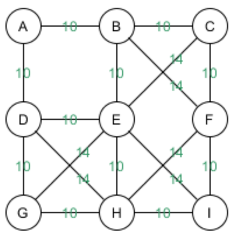


**Figure 5.3:** 3x3 grid with edge costs

## 5.7 Our implementation

does this belong into sec:map-handle?

For our grid we chose to allow vertical, horizontal and diagonal movement, giving us 8 possible directions to move in from every *node*. We decided to store those eight directions in one single byte, with the least significant nibble (LSN) corresponding to the four main directions (N,E,S,W), and the MSN corresponding to NE, SE, SW and NW.

| N | E | S | W | NE | SE | SW | NW | byte |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0x01 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0x02 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0x03 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0x0E |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0x10 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0x20 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0x42 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0xFF |

Just some random cites, to see how it works. [3], [4] and [5].

# 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] 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`.

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

[3] Lars Madsen. *Introduktion til LaTeX*. `http://www.imf.au.dk/system/latex/bog/`. 2010.

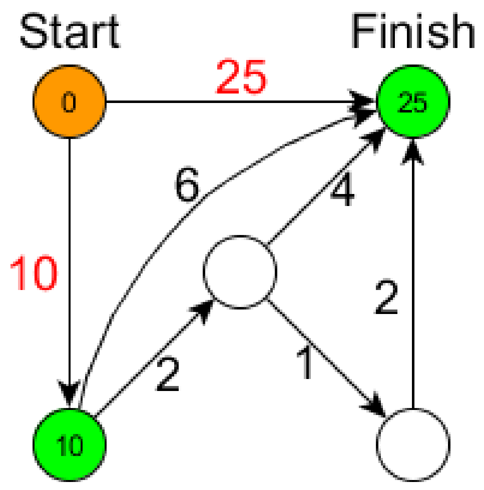
[4] Tobias Oetiker. *The Not So Short A Introduction to LaTeX2e*. `http://tobi.oetiker.ch/lshort/lshort.pdf`. 2010.

[5] Frank Mittelbach. *The LATEX companion*. 2. ed. Addison-Wesley, 2005.

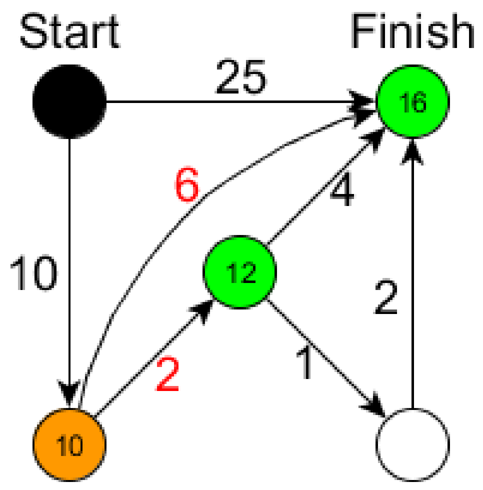
# Appendix A

# Pathfinding

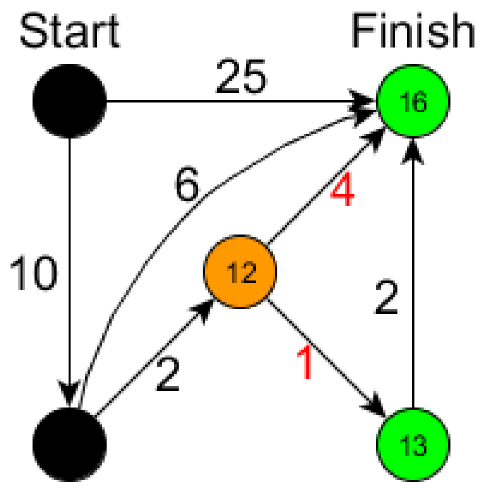
### Dijkstra



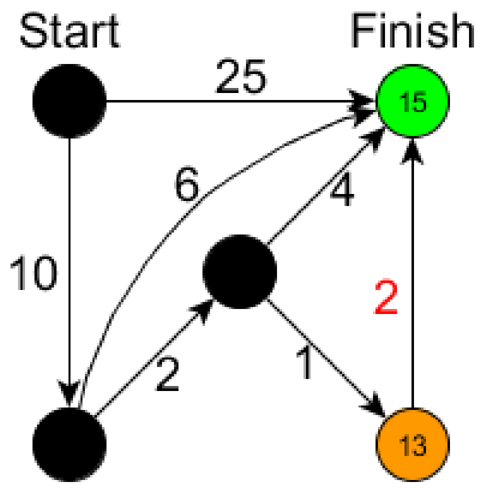
**(a)** 1. Step



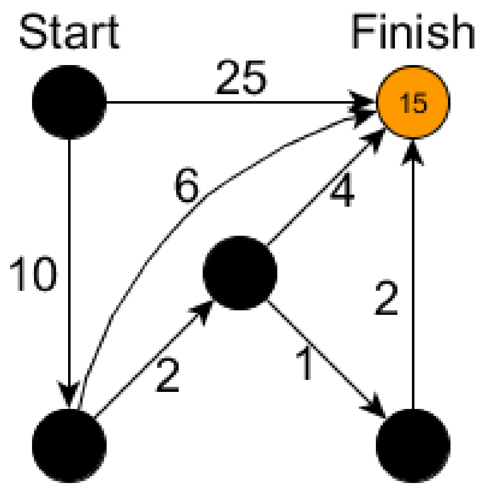
**(b)** 2. Step



**(c)** 3. Step



**(d)** 4. Step



**(e)** 5. Step