Warehouse AI

An embedded system for optimal placement and retrieval of books

Project Report

SW505E17

Aalborg University
Department of Computer Science

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
http://www.aau.dk

Title:
Warehouse AI

Theme:
Embedded Systems

Project Period:
Fall 2017

Project Group:
SW505E17

Participant(s):
Jakob Meldgaard Kjær
Jeppe Wehner Lindberg
Kennet Nørgaard Larsen
Nicklas Højgaard Sneftrup
Rasmus Kjettrup Thomsen
Tobias Klitgaard Sørensen

Supervisor(s):
Giovanni Bacci

Copies: 1

Page Numbers: 81

Date of Completion:
December 19, 2017

Abstract:

This report documents research and development of an embedded system. Warehouse AI is an embedded system on an Android smartphone that communicates with a server. Our project product is a system for employees that works in a warehouse. The server contains book information and handles book placement in the warehouse. The user can scan a book barcode and add the scanned books to the shelves by using our placement algorithm. The user can also request for a route if there is a book order. The server provides an optimal route through the warehouse for the user.

Contents

Todo list

Preface

This report, along with an attached computer program, is a result of a 5th semester Software project at Aalborg University. A thanks goes to our supervisor, Giovanni Bacci, for his help and constructive criticism throughout the project.

Chapter 1

Introduction

Traditionally, the placement of items in warehouses has been designed to be human-understandable. Items have traditionally been ordered by class and ID, to make finding specific items easier for employees. But lately, another tactic has emerged.

This other method of sorting the items in a warehouse instead uses the opportunities that evolving technologies enable to place items in places that allows the most efficient retrieval of items in the warehouse, given the employee can rely on a computerised system to find the locations of the given items.

The new method allows warehouses to be faster and more efficient than ever before, given the items are placed optimally - but what positions are the most optimal to place each given item on?

This question leads us the following initial problem statement:

- *How can an assortment of items be placed in a warehouse in a way that an employee will be able to collect a set of items fast and efficiently in the most probable cases, so that the collection process will be, on average, as quick as possible?*

To accomplish this, certain criteria must be fulfilled. These are described below, in no particular order:

1.1 Evaluation Criteria

The system needs support for some sort of display. The display should be used to display a route for an employee to find items in the warehouse. If the platform does not support this, the entire idea of the system is useless.

It should be possible for the user to add items to the system. The user needs a way to give the system input about which items should be added.

The system must be able to store a high amount of data. Since the system should represent the warehouse, the system needs the ability to contain data for every single item in the system.

It must be usable by warehouse employees. The target group is the employees in a given warehouse. This means that the system must be realistic and easy to carry around and use by these employees.

Part I

Analysis

Chapter 2

Warehouse Representation

The warehouse is a graph of connected nodes, where each node represents a position where a person can walk. How the warehouse is represented is entirely determined by the employees of the warehouse, as the calculations done on the warehouse do not differentiate between different warehouse representations.

The representation can be chosen by the employee to fit whatever they find most logical, or fit their specific warehouse.

In this chapter, the different representations are explored and explained and compared using the plan of a warehouse seen in Figure 2.1. Primarily the chapter makes use of [10] as a source.

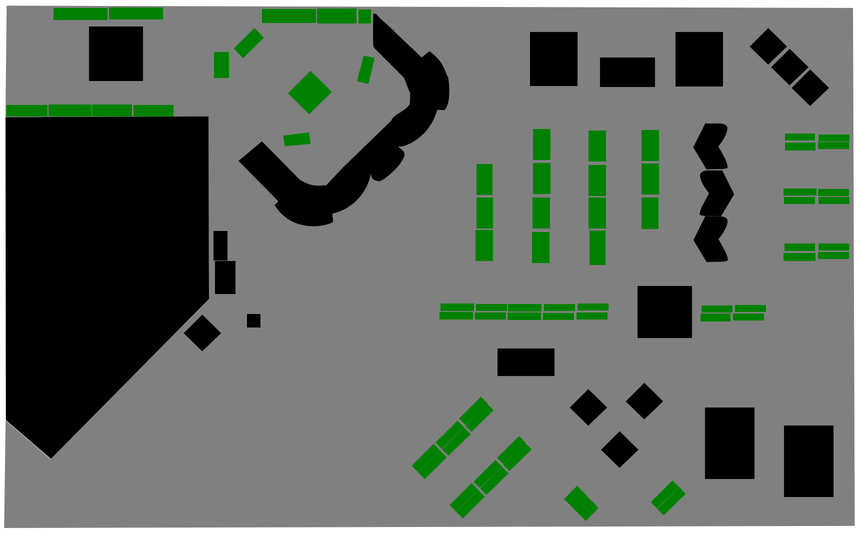


Figure 2.1: The plan of a warehouse, where each green area is a bookshelf, each black area is an area that cannot be walked on and the grey area is where a person can move.

2.1 Grid

A grid map makes use of uniform subdivision to define the world that the problem resides in. This subdivision provide us with a world as small regular shapes which can be squares, triangles or hexagons. When you work with grids you can choose between tile, edge or vertex movement [10].

Figure 2.2 shows Figure 2.1 with a grid as overlay. It is shown that some of the tiles contain different amounts of shelf space. Since tiles are seen as constant distances it would imply that a bookshelf that is four and a half tiles away would be seen as being just four tiles away, when it is in actuality further away. This could be changed by fitting the tile-size to the warehouse, but this could cause the grid to be more complex, since, in the case where an obstacle or a bookshelf is diagonal or circular, the grid would not be able to fit even if we changed the shape of the tiles to triangles or hexagons instead.

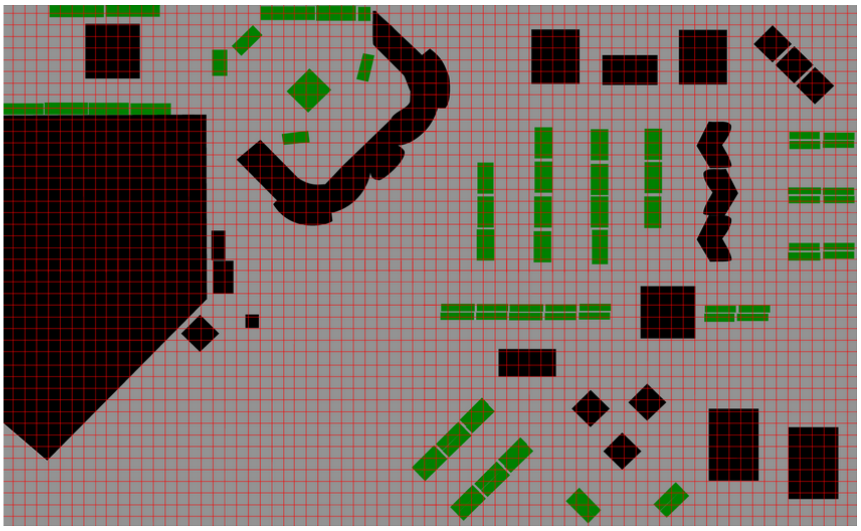


Figure 2.2: The plan of the warehouse, with a grid overlay.

2.2 Polygonal Maps

If grid representations do not meet the requirements, then polygonal maps are another alternative for the choice of representation. If you can move in straight lines on your grid instead of following the grid, and the movement cost are uniform, then you can use a non-grid representation to illustrate a route from start point to an end point as you can see on figure 2.3. The figure contains a pentagon and a triangle that the unit needs to move around to get to the end point [10].

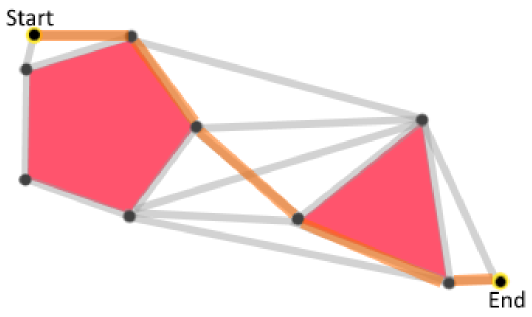


Figure 2.3: Illustration of a polygonal map.[10]

The shortest path will be to follow the obstacles by their corners like it is shown on figure 2.3. The corner points are used as "navigation markers" so that a pathfinding algorithm can find a shortest path between a start and end point.

2.2.1 Managing Complexity

The previous example with polygonal maps was simple. In some maps, the visibility graph can be more complex. The more open spaces and points the space has, the more complex the graph is as seen in figure 2.4.

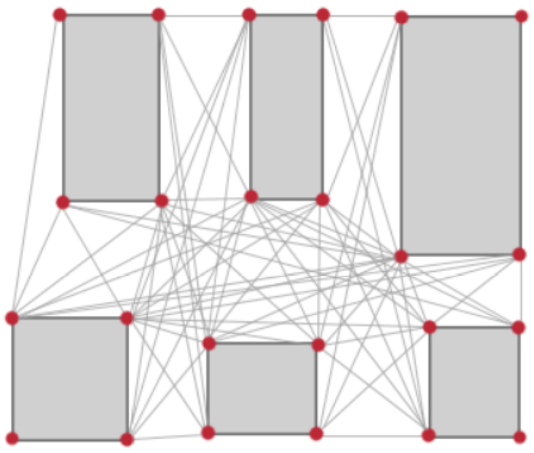


Figure 2.4: Illustration of a problem with visibility on polygonal map when there is too much open space.[10]

When comparing these maps with grids, these edges gives the path finding some "shortcuts" which helps it to find the shortest path faster than grid maps. By using algorithms that can remove unnecessary edges provides a faster way to analyse the shortest path[10].

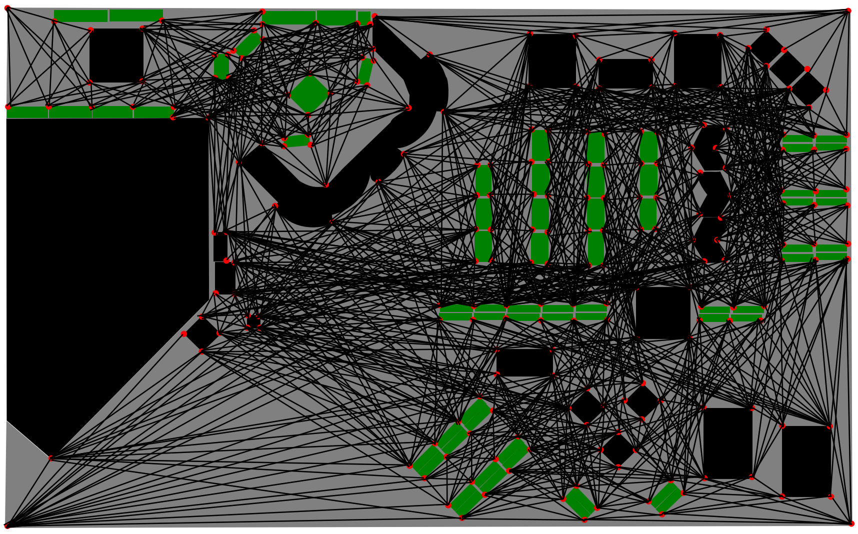


Figure 2.5: Edge movement

Figure 2.6: Polygonal graph representation.[10]

In the case of the warehouse plan seen in Figure 2.6 a polygonal graph would lead to an overly complex graph where the algorithm used for removing edges would be called many times in succession. Though it would be better to use a Polygonal graph as representation than a grid since polygonal graphs can represent diagonal areas, it is still inefficient in the case that the warehouse has large areas of empty space.

2.3 Navigation Meshes

Another way of representing the warehouse could be by representing walkable areas with a navigational mesh. These areas is illustrated as polygons around the obstacles as you can see in Figure 2.7.

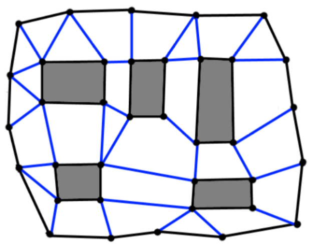


Figure 2.7: Illustration of a mesh.[10]

This way the model will show where one can move in the warehouse instead of where one cannot. Meshes can be treated like grids. Like with grids there is a choice between using polygon centres, edges and vertices as navigation points. There is also a hybrid movement where edges and vertices are combined [10].

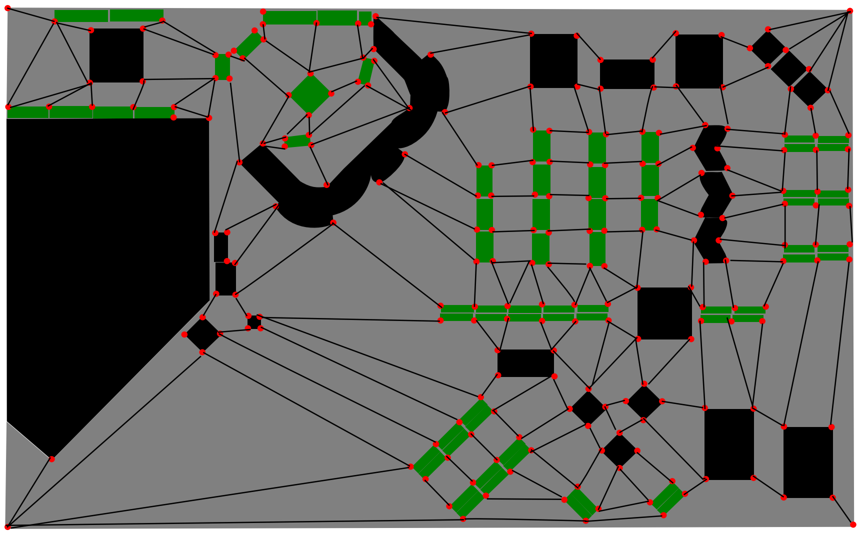


Figure 2.8: The warehouse plan with a mesh overlay.[10]

Compared to the graph in Figure 2.6 the mesh in Figure 2.8 is less complicated in that the polygonal graph has 1103 edges and 226 vertices, while the mesh only has 166 edges and 226 vertices. This means that the polygonal graph in this case almost has seven times the amount of edges as the mesh of the same warehouse. The most sensible choice in this case would thus be the use of a mesh for representing the warehouse.

2.4 Constraint Satisfaction Problem

A constraint satisfaction problem (CSP) is a way to describe the world that the problem resides in using a set of variables, a domain for each variable and a set of constraints[12]. Using these components, it is possible to describe the features of the world in a simpler way than using explicit states for each possible configuration of the world.

In this project, defining the world in this way helps describe the possible states that the world can assume, a state being a specific configuration that the warehouse can assume. It also describes the constraints that need to be satisfied to have a realistic representation of a warehouse.

Variables

The CSP should describe a warehouse of books, and therefore some logical conclusions should be drawn from the expected design of the warehouse, before the variables can be described.

The warehouse is expected to have shelves where books can be stored. Books on the shelves are to be stored as single books that can only can be picked up once each. Therefore, the quantity of each book must be represented in the system. Each shelf can have a limited amount of instances of items.

In this assumed world, a possible set of variables could be the following:

$$I_{ij} \qquad \forall i \ \forall j \tag{2.1}$$

where each I_{ij} is the number of instances of item j on shelf i.

Domain

The domain is the set of possible values that each variables can assume. Each variable describes the number of instances of a specific item on a specific shelf. This means that each domain $D_{I_{ij}}$ is a natural number.

$$D_{I_{ij}} = \mathbb{N} \qquad \forall i \ \forall j \tag{2.2}$$

Constraints

The constraints that the world resides under restricts the values that the variables can assume, to restrict the amount of possible states that the world can assume. In the model of the warehouse, each shelf can only contain a maximum of a constant o instances of items. Therefore, it is possible to construct the following constraint:

$$\sum_{\forall j} I_{ij} \leq o \qquad \forall i \tag{2.3}$$

With this CSP defined, it is possible to know which states are legal, and create algorithms that comply with the CSP rules.

Chapter 3

Choice of Platform

The choice of platform will impact the project in several ways: Programming language for implementation, ease of implementation, and ease of use. This section will focus on what the pros and cons of each platform are, and explain why a specific platform has been chosen.

3.1 Platform options

Before it is possible to make an informed decision about what platform to use in the project, the different options on platforms need to be explored.

3.1.1 Raspberry Pi

The Raspberry Pi is advertised as a small computer that can be used in a variety of projects. The Raspberry Pi 2 Model B is a complete computer with HDMI, USB, audio, SD, and Ethernet ports. In addition, it contains 40 General Purpose Input/Output (GPIO) pins, which means that the user can read and control sensors, LEDs, motors, etc.

Since the Raspberry PI has a HDMI port as well as a set of GPIO pins, it is possible to connect a display which fulfils the first criterion in section 1.1 about support for a display.

The 900MHz ARM processor with the 1GB of RAM that is available means that it can almost certainly run any task that is required. Since the Raspberry PI is basically a fully fledged computer that uses some distribution of Linux as its operating system, it supports a wide variety of programming languages.

The Raspberry Pi can be very portable. The entire system with screen and camera can be powered by a battery, with the right component.

A Raspberry Pi with a screen and camera can be a realistic product, with a high amount of portability, it is not unrealistic to expect warehouse employees to use the product in their everyday work. It would ideally be in a casing with only the screen and a few buttons showing.

Overall, the Raspberry Pi would be a solid choice. The system would be built almost from scratch with computer, camera, and screen interacting with custom software.

3.1.2 Android Smartphone

A smartphone running the Android operating system is a good choice based on the evaluation criteria: Most Android smartphones not only support but actually include a built-in camera and a display. Android smartphones come in many varieties with low and high amounts of processing power and RAM. We simply have to choose a specific one with the minimum specifications required.

The smartphone is programmable via the Android API that is used to control camera, screen, and other parts of the phone. However, older devices running Android versions 2.3 (code named Gingerbread) and earlier are hard to program, since backward compatibility does not reach that far back. This is something to bear in mind if we were to choose an older model.

Since 84% of Danish families owned one or more smartphones in 2017[5], it can be argued that smartphones generally are transportable. They can be carried in a pocket, and they generally include batteries that last several hours of heavy use.

Smartphones are realistically usable by warehouse employees. They are portable, include all required peripherals, and they are fairly easy to configure and program. A smartphone would therefore be a good choice for this project.

It can be argued that an Android smartphone is not an embedded system; while it likely does need to connect to a server, it does not physically need to connect to any other device. But it does, however, seem to work perfectly for our case, and so it is a nominee.

3.1.3 Lego Mindstorms NXT

Mindstorms NXT is a programmable robotics kit made by Lego. It is the second generation of the Lego Mindstorms line, which includes three generations: RCX, NXT, and EV3. Only NXT will be considered for this project due to the scarcity of EV3 at the university.

Mindstorms makes it possible to build a wide variety of robots by building and connecting interacting parts, and then programming actions for these in one of a variety of programming languages, including Mindstorm's own visual language, NXT-G.

Lego Mindstorms does support a light sensor that can be used to read printed codes such as barcodes or the more complicated QR codes. It also supports a display, though this is extremely basic and would require some creative thinking for it to be used for communicating a route through a warehouse.

Portability would likely not be an issue. The NXT is powered by an internal battery, and the main module is measured at about 11 cm in length, 7 cm in width. This is likely small enough to be carried around by employees.

There are a few examples online of NXT robots reading barcodes and displaying content on the screen. However, a shortcoming is that it is not able to scan regular barcodes. The light sensor cannot read small barcodes as seen on most wares in shops - it needs quite a large one to be able to distinguish between black and white. Furthermore, the sensor needs to be close to the barcode and scan it from one side to another in a constant motion. This will not work in a real-world scenario.

3.2 Choosing a Platform

The Lego Mindstorms NXT is not applicable for real-world use. It is not able to scan regular barcodes and QR codes, and it is slow and needs quite a bit of precision for scanning the ones that it supports. The Raspberry Pi would be a good choice. It fulfils the evaluation criteria. Android however also fills out these criteria and has the advantage that it is already used daily by most people. If we did not have a viable alternative, the Raspberry Pi would be the way to go, but the Android platform feels like a better choice of platform for the project.

To conclude, the Android platform is to be used in this specific project. It fulfils the evaluation criteria, and it includes all peripherals needed. It is portable, and most people in Denmark have one in their household. In addition, it is easily programmable (with some caveats regarding older devices and backward compatibility) via the Android API.

3.3 Android - Choosing a Device

This section will focus on the choice of devices to use in our project. The Android smartphone that is the target hardware will be the handheld device that the warehouse employee carries around. The Android device fulfils the demands of being able to take a picture of a printed code and decode it, send data to a database, receive data from a central server, and display this data correctly on the screen of the device.

The smartphone that will be used as test device is a Huawei Ascend Y530 which is approximately 3 years old. The smartphone was launched in February 2014.

One of the reasons we wanted to use an old Android smartphone is because they are not as powerful in terms of what hardware they have compared to what modern smartphones have nowadays. They are almost equally powerful to the Raspberry Pi 2B. The other reason is because we want our device to feel like a "real device" that employees would use in real warehouse situation instead of a "toy" like Lego Mindstorms NXT.

We will compare the Android hardware with Raspberry Pi 2B which is mentioned in subsection 3.1.1. The reason for doing this is so we can view how powerful the Android device is compared to such devices as Raspberry Pi.

Comparing List		
Hardware	Huawei Ascend Y530	Raspberry Pi 2B
CPU:	Dual-core 1.2 GHz Cortex-A7	900MHz quad-core ARM Cortex-A7
RAM:	512 MB	1 GB
Camera:	5 Megapixels	5 or 8 Megapixels*
WC	Wifi and Bluetooth	Wifi and Bluetooth*
Display:	4.5 inch Screen	HDMI output

Table 3.1: Comparing Huawei Ascend Y530 [6] with Raspberry Pi 2B [13] - WC is Wireless Communication - * Item is not on the standard Raspberry Pi but is a side component that can be purchased.

The CPU on the Android phone is better when it comes to applications that do not support multithreading. An application has support for multithreading, then it will probably run faster on Raspberry's quad core. Otherwise, as seen on table 3.1, the Raspberry Pi have more internal RAM than the Android. If willing to buy the extra components for the Raspberry Pi 2B then it will able to do what the Android can do in terms of display quality and camera quality. Since the Raspberry Pi have a HDMI port, it can connect to a monitor or screen which can receive input from a HDMI cable. The Raspberry Pi has several different choices when it comes to the camera component, which can be better, worse, or just as good as the Android camera.

One of the minor problems with this is that a handheld device should be easy for an employee to carry around a warehouse. If we have all these components and screen attached to the Raspberry Pi, it might prove difficult to use for the employee. In this case, it is easier to simply use an Android phone.

Chapter 4

Barcode

This chapter covers barcodes and how they work, how they are decoded, what kind of error correction they are using, as well as how it is calculated. The reason for this chapter is that we need some sort of barcodes in our application and it is an important part of our project.

Barcodes are often used to help organise and store information or prices about an object. A smartphone can function as a barcode scanner to decode the barcodes into numbers. Several different barcode types exist, each with their own unique patterns.

4.1 EAN-13

There are 95 columns in a barcode as seen in figure 4.1. These black and white columns represent 1 and 0, respectively.

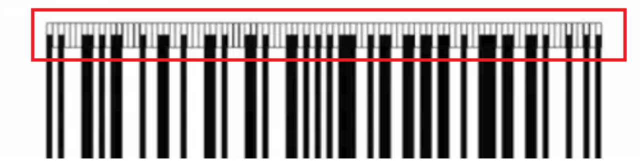


Figure 4.1: There is 95 columns in a barcode. - Picture edited from original [1]

Barcode type EAN-13 is nominated for use in the project due to its ubiquity on the backside of books due to each book having a 13-number ISBN. This section focuses on how to decode an EAN-13 barcode. Below is an example on an EAN-13 barcode with illustrated red markings around left- and right-handed bars.

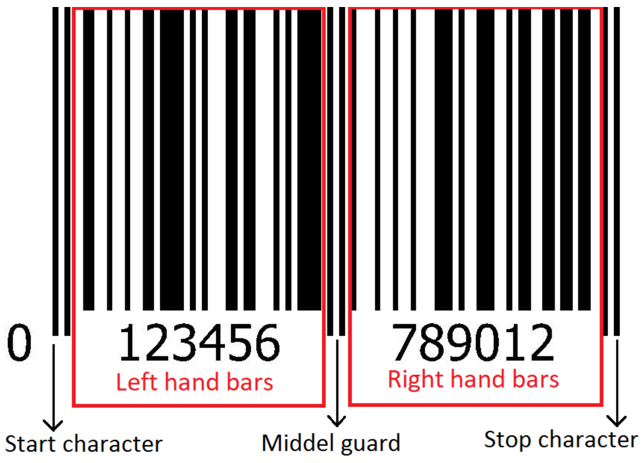


Figure 4.2: Example of an EAN 13 with illustrated left and right handed bars.

An EAN-13 barcode has the following structure[7]:

- The first digit (on the left side of the barcode) is the parity decider.
- Start character is encoded as 101.
- In the left-hand bars the values always start with a 0.
- Middle guard pattern, encoded as 01010.
- Stop character is encoded as 101.
- In the right-hand bars the values always start with a 1.

The columns of the barcode encodes digits in binary representations. Each digit of the number is represented by seven binary values. Six digits in each side of the barcode make up the resulting number encoded by the barcode. On the left hand bars, as illustrated on the figure 4.2, the representation always starts with the value 0 (white) and the right handed bars starts with 1 (black). The first digit of the barcode (the parity decider) dictates whether you should use ODD or EVEN parity in the left hand bars on the parity table 4.1. For decoding the binary values we need to use the table as seen on Table 4.2[7].

Parity Table						
FNSD	Parity to encode with					
	SNSD	Manufacturer Code Characters				
		1	2	3	4	5
0	ODD	ODD	ODD	ODD	ODD	ODD
1	ODD	ODD	EVEN	ODD	EVEN	EVEN
2	ODD	ODD	EVEN	EVEN	ODD	EVEN
3	ODD	ODD	EVEN	EVEN	EVEN	ODD
4	ODD	EVEN	ODD	ODD	EVEN	EVEN
5	ODD	EVEN	EVEN	ODD	ODD	EVEN
6	ODD	EVEN	EVEN	EVEN	ODD	ODD
7	ODD	EVEN	ODD	EVEN	ODD	EVEN
8	ODD	EVEN	ODD	EVEN	EVEN	ODD
9	ODD	EVEN	EVEN	ODD	EVEN	ODD

Table 4.1: This is the Parity Table for EAN-13 for the left-hand bars. - FNSD is First Number System Digit - SNSD is Second Number System Digit. [7]

Decode table			
Digit	Left-Hand Encoding		Right-Hand Encoding
	ODD PARITY	EVEN PARITY	
0	0001101	0100111	1110010
1	0011001	0110011	1100110
2	0010011	0011011	1101100
3	0111101	0100001	1000010
4	0100011	0011101	1011100
5	0110001	0111001	1001110
6	0101111	0000101	1010000
7	0111011	0010001	1000100
8	0110111	0001001	1001000
9	0001011	0010111	1110100

Table 4.2: This is the decoding table for EAN-13 - The first number of the barcode dictates that you should use ODD or EVEN parity in the left handed bars in the table 4.1. [7]

The last number in the barcode is a check digit which acts a form of error detection. The digit is used to verify that the barcode is read correctly. On the figure 4.3 below is a way to calculate the check digit of a barcode.

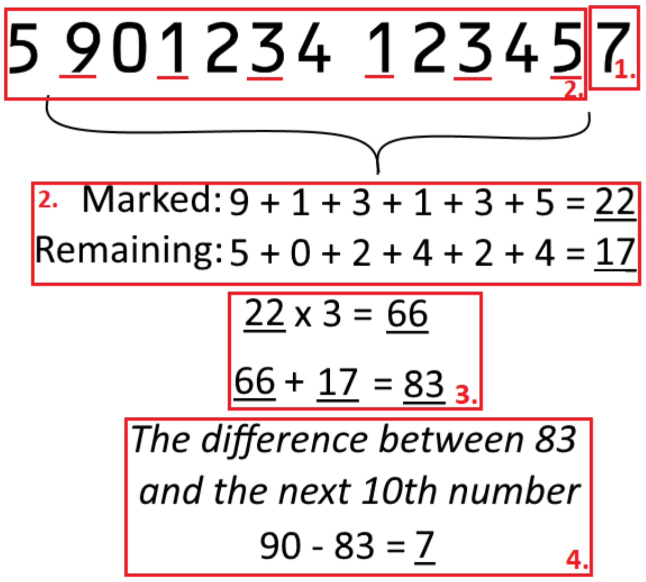

5 901234 123457
1.
2.
2. Marked: 9 + 1 + 3 + 1 + 3 + 5 = 22
Remaining: 5 + 0 + 2 + 4 + 2 + 4 = 17
22 x 3 = 66
66 + 17 = 83
3.
The difference between 83 and the next 10th number
90 - 83 = 7
4.

Figure 4.3: How to calculate the check digit on an EAN-13 barcode: **1.** Is the check digit of the barcode. - **2.** Every second digit is added together (marked) and the same for the remaining digits expect for the last digit. - **3.** The sum of marked digits are multiplied with three. That sum is added to the sum of the remaining digits. - **4.** The difference between 83 and the next 10^{th} number is 7 which is the check digit as seen in step **1.**

Below is an example on the figure 4.4 on how to decode a barcode EAN-13. Where we use the parity table to determine which parity to use by reading the first number on the left side of the barcode. Then we analyse the bars and spaces to get some binary numbers by counting how wide the bars and spaces are. Then these binary numbers are decoded to integer numbers by using the decode table.

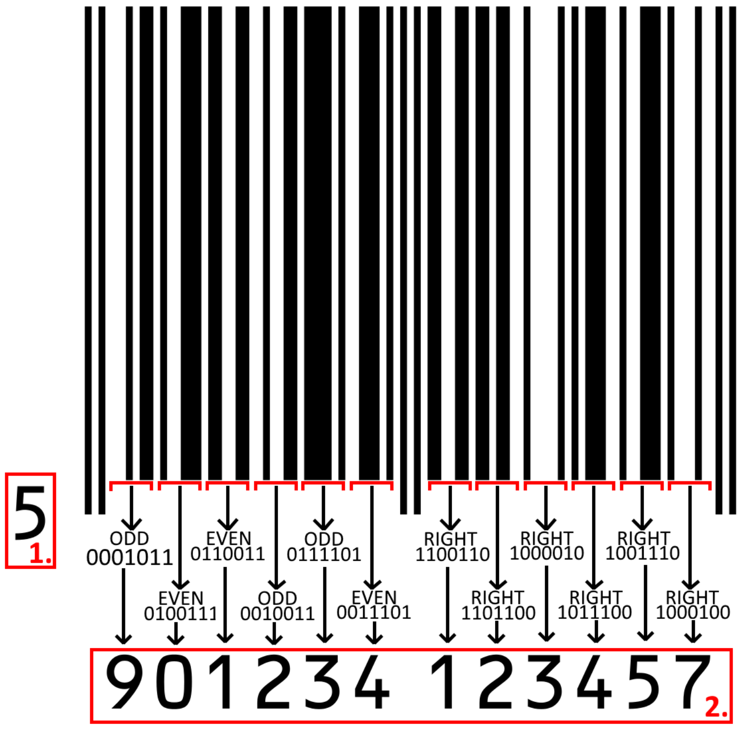


Figure 4.4: Example on how to decode an EAN-13 barcode - 1. The first number tells which parity to use from table 4.1. Then the decoding table, Table 4.2, is used to decode the bars and spaces. - 2. When decoding is done, the output should match with the numbers under the barcode.

4.2 QR Codes

This section focuses on the inner works of the type of matrix barcode that is the Quick Response (QR) code. In order for a reader to read a code, certain requirements of the composition of the code have to be met. There are a variety of QR types: "Normal" QR, Micro QR, Frame QR, and others. This section will describe the former version only.

4.2.1 Modules

A module is the basic building block of QR. It can be compared to a pixel on a computer screen - it contains a single colour, usually black or white. A naïve calculation of the amount of possible QR codes of size $177 * 177$ modules (version 40, described below) is $2^{177^2} = 9.3 * 10^{9430}$ possible combinations. This is not completely precise, however, since some patterns have to be present in the code in order to be read at all. Required patterns are described below.

4.2.2 Versions

The version number describes the dimensions of the QR code and as such the amount of data that a QR code can contain. A version is a number between 1 and 40. Version 1 can contain 17 to 41 numeric characters or 10 to 25 alphanumeric characters, depending on the amount of error correction, described below.

$$version = \frac{sideLength - 17}{4} \tag{4.1}$$

Essentially, for each version number increment, the height and width of the code are increased by four modules, starting from 21.

4.2.3 Parts of QR

Essentially, there are seven parts of a QR code. These parts can determine the actual data embedded inside, positional information, format of the information(error correction rate and masking), error correction, version of the QR code. The following sections describe the static elements

Finder Patterns

The finder patterns have a specific structure on all versions QR codes: A black $3 * 3$ square surrounded by a white frame surrounded by a black frame - a 1:1:3:1:1 ratio of black/white/black/white/black modules:

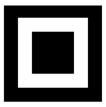


Figure 4.5: A finder pattern. Rotation does not matter.

Their purpose is code orientation. There are always three finder patterns in a QR code, and they are always placed in the bottom left, top left, and top right corners. The fact that the finder patterns cannot be rotated incorrectly means that readers look for these to actually know if it is a QR code that are being scanned, and then reorient the code accordingly if necessary.

Alignment Patterns

Alignment patterns are used when a QR code is distorted to help position detection. The more complex the code, the more alignment patterns are needed. This feature is used in version 2 and up.[3]

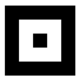


Figure 4.6: An alignment pattern. Helps the decoder to correct distortion.

Timing Pattern

The timing pattern is used to determine the physical size of the modules. The pattern consists of two lines of alternating black and white modules in the top and left of the code.

Quiet Zone

The quiet zone is a border of empty space around the QR code. This border is at least the width of four modules and is used to avoid interference from other graphics that might be printed nearby.

Lonely Black Module

There is a single module to the right of the white bottom left finder pattern border. This module is always black, but it is not used for anything.

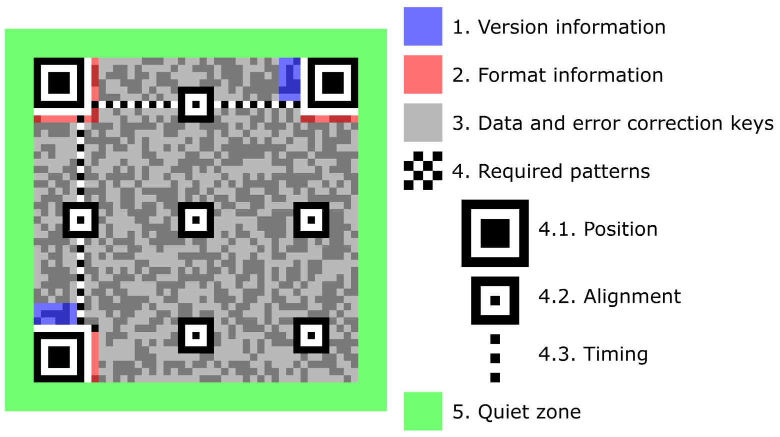


Figure 4.7: Structure of a QR code (version 7). Source: Wikipedia

Masking

A mask is applied to the data area of a QR code in order to have a nice balance of black and white modules. This avoids the possibility of having large areas with uniform modules, or even having what looks like timing patterns in the data area, possibly confusing the scanner.

The available patterns are as follows (rows and columns start at index 0):

000	(row + col) mod 2 = 0
001	row mod 2 = 0
010	col mod 3 = 0
011	(row + col) mod 3 = 0
100	((row/2) + (col/3)) mod 2 = 0
101	(row * col) mod 2 + (row * col) mod 3 = 0
110	((row * col) mod 2 + (row * col) mod 3) mod 2 = 0
111	((row * col) mod 3 + (row * col) mod 2) mod 2 = 0

Table 4.3: The different masks that are available for a QR code to use

In order to select a suitable masking pattern, each one is tested and given points based on four traits that has to be avoided as much as possible:

- Modules in the same row and column, respectively, are the same colour.
- Entire blocks are the same colour.
- Rows or columns have the 1:1:3:1:1 timing pattern, preceded or followed by four white modules.
- The entire symbol consists of more or less than 50% dark modules.

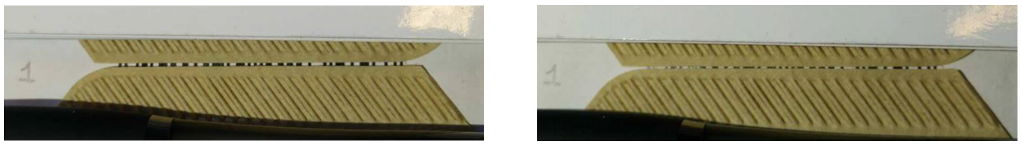
The specifics of how points are given will not be discussed here, but the essence is that the pattern with the least amount of points is selected for the masking of the QR code.

4.3 Barcode Performance Testing

This section covers performance tests on both EAN-13 and QR barcodes.

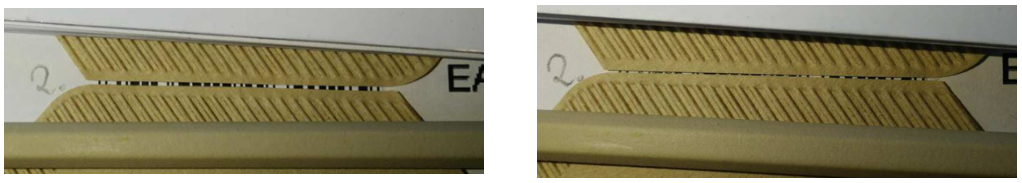
4.3.1 EAN-13 Tests

We have tested EAN-13 barcodes in order for us to know how much of the barcode is required to be recognised, to figure out which type of barcode is best for this project. The way we tested the different codes was by covering the codes with paper and then test if a barcode scanner application was able to scan the codes. We tested the barcode multiple times, because we had to make sure that the scanning test was following the scientific method.



(a) Barcode 1 that is scannable **(b)** Barcode 1 that is not scannable

Figure 4.8: Barcode 1 - Two test runs where **(a)** is scannable and **(b)** is not.



(a) Barcode 1 that is scannable **(b)** Barcode 1 that is not scannable

Figure 4.9: Barcode 2 - Two test runs where **(a)** is scannable and **(b)** is not.

As seen on figures 4.8 and 4.9 the barcode is almost covered but is still able to be scanned by the application. There is a chance that the scanning of a barcode gets a different output to what the barcode actually contains even if the barcode is not covered. That is a problem that our group is aware of and try to work around by crosschecking the barcode output with book codes that are stored in the database.

Test 1 - Full	✓	X	X	✓	✓	✓	✓	✓	✓	✓
Test 1 - Covered	✓	✓	✓	✓	✓	✓	✓	✓	X	✓
Test 2 - Full	✓	✓	X	✓	✓	✓	✓	✓	✓	✓
Test 2 - Covered	✓	✓	✓	✓	X	✓	✓	✓	✓	✓

Table 4.4: Test samples on two barcodes - There are two tests on each barcode where the barcode is fully visible and where the barcode is covered seen on Figure 4.8a and Figure 4.9a. - ✓is a correct reading, and X is a wrong reading.

As seen in table 4.4 the barcode samples that were used were tested ten times at each run to get a result that could indicate how many times the application read correctly. The figures 4.8a and 4.9a were the last scannable samples that the application was able to scan.

4.3.2 QR Tests

In order to test whether QR would be a suitable choice for our system, several tests with the Android smartphone were conducted. The QR code was scaled down in size, and then the distance from the QR code to the camera was measured. Version 2 of QR was used, since that is the version we are considering to use in the project. The camera distance has been measured three times per size to get a more accurate result on the camera. As seen on table 4.5, the camera can recognise a normal sized QR code approximately 39 cm from the QR code.

Version 2 QR code				
QR code size (cm)	Distance from smartphone to monitor (cm)			Average distance
	Test 1	Test 2	Test 3	
0.9	Cannot recognise QR code			
1	12,5	12	12	12,17
1,1	12,5	11,5	12,5	12,17
1,2	11,5	12	13	12,17
1,5	18	19	17,5	18,17
2	25	26	24	25
2,5	35,5	32,5	32	33,33
3	37,5	39	39,5	38,67

Table 4.5: QR code version 2 - Distance and size test with Huawei Ascend Y530 camera - Where 3 cm is the normal size of an QR code.

4.4 Choice of Barcode Type

Now that both QR and EAN-13 have been tested with results showing that both have the capabilities to work in the system, it should be possible to make a choice of which type should be used in the system. QR has the advantage that it can hold more information, but since the client only is supposed to send a small string of characters anyway this would be redundant. QR codes also have the ability to recover lost information, while a barcode does not. EAN-13 however has the advantage that they are used on a great deal of products including books[4]. EAN-13 may not use many different encodings, like QR but in the case of this system being limited to only numeric encoding is an advantage, since it prevents the clients from sending unsupported characters to the server.

From this information we can gather that EAN-13 barcodes would be better fit for our system. While it would be nice for the system to have access to QR-barcodes' ability to recover lost data, it is not necessarily needed in the system.

4.5 Performance Test

ZBar supports many features like cropping the size of the area it wants to scan and stride is the of the number of skipped pixels between each pass of the scanner.

4.5.1 Crop Size on Camera

Crop the size of the monitor offers more performance for the scanner but lower usability for the users. We as a group decided that we wanted to favor the usability for the users because the scanner is pretty fast to recognise the barcode with the full monitor available. The cropped size is proved to be faster by 28 milliseconds. When compared, the average of both normal and cropped the values are taken from table 4.6, so based on our tests we will keep the full monitor.

4.5.2 Stride

Stride is an option that dictates how often the scanner scans on the vertical and horizontal view for a scannable code.

You can choose how big a stride you want on the vertical view by setting it to a specific number and that number dictates how many pixels there are between each scan. This is done on both horizontal and vertical views and it can also be set to zero which means that that specific view is disabled for scans.

There is not a default setting when it comes to choose how big a stride should be on a view but a good target is between two and four stride to get a good balance between how many scans and performance [2]. In our case we choose to set the stride to 10 because we focus on performance when it comes to strides and we want the system to be fast. As you can see in table 4.6 that we tested different strides because we want to see how to optimise our project system. The last row on the table shows how the different normal or cropped samples reads/captures a image where there is something to scan and then how fast it is to scan the detected barcode.

Performance test - Barcode scanner					
Normal 720x1280 1 stride	Cropped 320x680 1 stride	Normal 720x1280 10 stride	Cropped 320x680 10 stride	Normal 720x1280 max stride	Cropped 320x680 max stride
171	92	92	69	69	56
182	101	69	65	67	54
175	90	97	71	55	87
168	97	91	90	66	64
165	87	88	96	59	89
163	103	97	69	65	60
164	94	65	60	96	66
162	102	98	61	68	75
156	100	70	61	63	62
182	91	93	58	70	84
164	88	93	65	64	58
180	95	64	70	58	62
170	97	70	70	69	82
182	102	96	58	57	68
156	110	70	67	73	75
169	**97**	**84**	**69**	**67**	**69**
Read: Fast Scan: Slow	Read: Fast Scan: Medium	Read: Fast Scan: Medium	Read: Fast Scan: Fast	Read: Slow Scan: Fast	Read: Slow Scan: Fast

Table 4.6: Performance test - Barcode scanner - 3 different samples for normal and cropped view - 15 tests on each sample and penultimate row represent the average for the samples. - The last row represent how fast the scanner captured and scanned the image that the barcode was in.

Chapter 5

Algorithms

To fulfil the requirements of the problem statement a number of algorithms are needed.

In this chapter, the theory behind the different algorithms, specifically the ones related to the requirement of obtaining an optimal route, will be explained in general, as well as the algorithms needed for storing books in the warehouse.

5.1 Efficiency Formula

One central formula is used to calculate a real number, which represents how well every book is placed in the warehouse.

$$\underset{s\,\in\,S}{min} \sum_{i\,\subset\,items} c(i) \cdot w(i,s) \tag{5.1}$$

The most efficient state of the warehouse is the one with the smallest evaluation value. Better placements are determined by short distances between items which are closely related to one another.

c in this formula specifies the importance coefficient of a relation between any amount of items and is mapped by:

$$c : \mathcal{P}(Items) \to \mathbb{R} \tag{5.2}$$

c can be defined as: $c(r) = \frac{a}{n}$, where a is the number of arcs in relation r and n is the number of nodes in the relations r.

w is a map assigning a weight to each set of items in a given state of the warehouse representing the cost:

$$w : \mathcal{P}(Items) \times S \to \mathbb{R} \tag{5.3}$$

The calculation of w is described in section 5.3, where several techniques are used to calculate $w(i,s)$, including graph searching.

To make an attempt at calculating efficiency, algorithms have to be created that calculates c and w for given relations. These algorithms will be further explained in the following sections.

5.2 Importance Coefficient

The importance coefficient is a measure of how important a set of books are to each other. It is calculated over a relation of books, which is another term for a set of books, in the context of how related the books are to each other. In addition, a relation from one book A to book B means book A has an edge to book B in the relation graph, and the two books are therefore more related. A set of books are said to have a high relation if many of the books in the set have relations to other books in the set. An example of a set of highly related books could be the relation over all three books in a trilogy, as customers often are interested in the other two books if they are already interested in the first one.

When calculating the importance c for a given relation we have to make sure that if the set of books i are highly related, the relation is given a high importance coefficient. This is done through the formula seen in Equation 5.4 where a is the number of arcs in a relation r and n is the number of nodes in the relation r.

$$c(r) = \frac{a}{n} \tag{5.4}$$

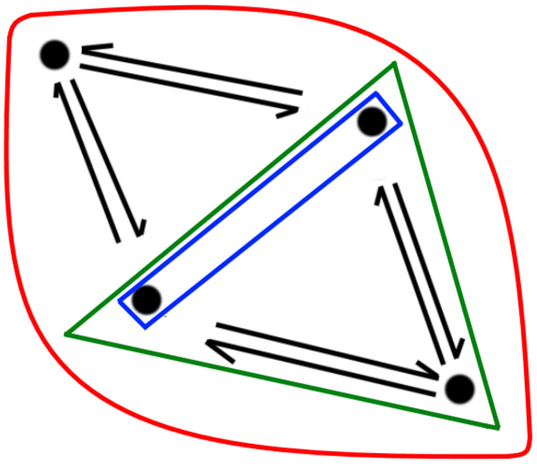


Figure 5.1: Illustration of the importance algorithm. The black nodes in the graph each represent a type of item. The red marking shows a relation of the entire set of items. The green marking shows a relation of a subset of all items. The blue marking shows another relation of a subset of all items. The black harpoons symbolise a relation from one item to another.

Figure 5.1 is an example of how three different relations could be over a set of four books. The importance for the entire set of items R in the figure would be calculated as $c(R) = \frac{8}{4}$ because the set contains 4 items and 8 arcs. The coefficient of the green subset G would be calculated as $c(G) = \frac{4}{3}$ since the set contains 3 items and 4 arcs. Lastly the blue subset B would be calculated as $c(B) = \frac{0}{2}$ since the set contains two items with no relations to each other.

The reason that we ended up using this calculation for the coefficient is that it translates reasonably well into the real world.

5.3 Calculating Weight

The weight $w(i, s)$ is the minimal distance required for an agent to travel to pick up all the books in a set of books i in the warehouse configuration s. The weight value can be calculated in several different ways, and each way has advantages and disadvantages.

The problem of picking up a set of different books in a warehouse is akin to a pathfinding problem, but not in the classical sense; each book in the set of required books needs to be picked up once, which means the pathfinding algorithm will have several goal nodes. The agent will need to move from the drop-off point, through each goal node, and return to the original node, taking the shortest possible route. In addition, several books can be picked up on each node, if the node contains several of the required books.

A possible solution to this problem is creating a graph G of all nodes that contain books as well as the drop-off point, where each node has a connection to another node, with the weight in the connection being the distance between the two nodes. This graph is represented using a matrix M where the value of each cell of the matrix is calculated using an all-pairs shortest path algorithm.

The graph G is then searched through in a lowest-cost-first search, where each frontier moves to a node that contains books in the set of required books, and each frontier crosses out the books that is contained in the visited node. When all books have been crossed out in this manner, and the first frontier has returned to the drop-off point, a heuristic for the shortest path has been found, only including the goal nodes, as all intermediate nodes have been skipped.

To optimise this approach, the weight of subsets of i is found and saved for use in future calculations. To achieve this, a dictionary C is created where each element is a tuple with a *key* element, and a *value* element.

- *key*: A set of item types.
- *value*: Another tuple with an element *marked* and an element *weight*.
  - *marked*: A binary value that denotes whether the value of this element should be updated.

- *weight*: The weight of the set of items, calculated in an earlier run of the *weight(i, G)* algorithm.

C is then searched for an already calculated solution to the *weight(i, G)* algorithm for the given set i, whenever a weight for i is requested.

In this way, $w(i, s)$ can be computed as:

$$weight(i, G) \tag{5.5}$$

where G is a graph constructed from the state s, where all nodes that are not shelves and not the drop-off point has been removed. The *weight(i, G)* algorithm is described in algorithm 1. In the following section, some elements that the algorithm makes use of are described.

The matrix M is a matrix of $n \times n$ elements, where each element in the matrix contains the weight of the shortest path between two nodes. Only the shelves and the drop-off points in s are necessary in M. Therefore, n is the total number of shelves in s plus one. All nodes have a distance 0 to itself, so for each $a \in \{1, \ldots, n\}$, $M_{aa} = 0$. Since all edges in the warehouse representation are undirected, the matrix M is symmetric.

The drop-off point is denoted d, and each shelf is denoted h_i where $i \in \{1, \ldots, n-1\}$.

$$M = \begin{pmatrix} 0 & dist(d, h_1) & dist(d, h_2) & \ldots & dist(d, h_{n-1}) \\ M_{12} & 0 & dist(h_1, h_2) & \ldots & dist(h_1, h_{n-1}) \\ M_{13} & M_{23} & 0 & \ldots & dist(h_2, h_{n-1}) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M_{1n} & M_{2n} & M_{3n} & \ldots & 0 \end{pmatrix} \tag{5.6}$$

Creating the M matrix before any weights are calculated is useful for optimising the solution. If the implementation required each weight calculation to make a pathfinding search to each node that contains a relevant book, the pathfinding algorithm would have to be executed very often. By using the M matrix, the weights between nodes can be reused, the overall efficiency have been increased when running evaluations on the state of the warehouse, and when calculating weight of a set of items.

The algorithm used to compute *dist(a,b)* is described in subsection 9.2.1.

The graph G is then created using M. Every node G_i has an edge to itself with weight 0, and an outgoing edge to every other node G_j with weight M_{ij}. Node G_0 is the drop-off point.

A dictionary C is also created to function as a cache. The dictionary has an element for each $i \in \mathcal{P}(Items)$. The elements are initialised to the values $\langle key := i, value := \langle marked := \textbf{true}, weight := 0 \rangle\rangle$ for each $i \in \mathcal{P}(Items)$, except for $C_\emptyset$, which is initialised to the value $\langle marked := \textbf{false}, weight := 0\rangle$, as collecting a set of no books takes no time, and the value does not need to be updated.

Every time a book b is added to or removed from a shelf, all elements in C where the key contains b should be updated to have $marked := \textbf{true}$.

Whenever a weight for a set of items are then needed, the dictionary is checked for whether the dictionary element for it is marked. If the element is not marked, then the *weight* value of the element is accurate, and the weight is easily read off the dictionary. If the element is marked, then the algorithm should run to check the weight, set the dictionary element to not be marked, and update the weight value.

This improves the performance when several weight calculations are done in bulk, such as during the evaluation function (section 5.1).

The *weight(i, G)* algorithm uses *distance(a, b)* to denote the value M_{ij} where i and j are respectively the indices corresponding to a and b.

Identifier		Denotes
C_i	:	A specific value in C with key i. Has attributes *marked* and *weight*.
f	:	The set of current frontiers.
f_r	:	A temporary variable that denotes the resulting frontier of the current evaluation.
f_i	:	A frontier, and the iteration variable when all frontiers in f is iterated through. Contains the attributes *route*, *books* and *weight*.
c	:	A cycle through the graph where, through the cycle, a given set of items are collected.
e	:	The set of already explored routes.
G_0	:	The drop-off point of books in the warehouse. A special node that does not contain any books, but like all other nodes, has the attribute *neighbours*.
G_l	:	A single node. The last node in the route of f_i.
G_i	:	A node that is a neighbour of G_l.

Table 5.1: The identifies and denotations of the variables in algorithm 1.

Algorithm 1 calculates the required distance to pick up a set of books in a specific state of the warehouse. The algorithm starts by initialising the variable f on line 1. f denotes the set of current frontiers. Each separate frontier has three elements:

- *route:* An ordered set of nodes in G.
- *books:* The remaining books required before the frontier should find its way back to the drop-off point.

Algorithm 1: weight(i, G)

```
1  $f := \{\langle route := (G_0), books := i, weight := 0\rangle\}$ /* Initialise the set of frontiers */
2  $e := \emptyset$
   /* Check if weight of 'i' in G has to be updated */
3  while $C_i.marked$ do
4      $f_r := \langle route := \emptyset, books := \emptyset, weight := \infty\rangle$
5      $c :=$ null
       /* Scan all frontiers */
6      for each $f_i$ in $f$ do
7          $G_l := f_i.route.last\_node$
8          for each $G_i$ in $G_l.neighbours$ do
               /* The node 'G_i' has some books that are still needed */
9              if $G_i.books \cap f_i.books$ is not $\emptyset$ then
                   /* If the extension of frontier 'f_i' with 'G_i' was not already seen, and its
                      weight is lower than current best, then... */
10                 if $e \cap (f_i.route \cup G_i)$ is $\emptyset$ and $f_i.weight + distance(G_l, G_i) < f_r.weight$ then
                       /* ... update current best route */
11                     $f_r := \langle route := f_i.route \cup G_i, books := f_i.books \setminus G_i.books, weight :=$
                        $f_i.weight + distance(G_l, G_i)\rangle$
12                     $c :=$ null
13                 end
14             end
15         end
           /* Check if the cache element of the set of books collected so far needs to be updated */
16         if $C_{i \setminus f_i.books}.marked$ then
               /* If the cycle is better than the current best frontier... */
17             if $f_i.weight + distance(G_l, G_0) < f_r.weight$ then
                   /* ... update 'c' with a new cycle */
18                 $c := \langle route := f_i.route \cup G_0, books := f_{i \setminus c.books}, weight :=$
                    $f_i.weight + distance(G_l, G_0)\rangle$
19             end
20         end
21     end
22     $f := f \cup f_r$ /* Update current set of frontiers */
23     $e := e \cup f_r.route$ /* Update current set of explored routes */
       /* If an efficient cycle was found in the last exploration ... */
24     if $c$ is not null then
25         $C_{c.books}.marked :=$ false /* Unmark the set of books found in the cycle ... */
26         $C_{c.books}.weight := c.weight$ /* ... and update its weight */
27     end
28 end
29 return $C_i.weight$
```

- *weight:* The weight of the path the frontier has taken.

f is initialised to contain exactly one frontier. The route of the initial frontier is initialised to only contain the drop-off point, the books are initialised to i, and the weight is initialised to 0.

On line 2, e is initialised, which denotes a set of already explored routes. Each element in e contains an ordered set of nodes. It is initialised to the empty set.

The loop on line 3-28 evaluates whether or not the element C_i is marked or not. If it is not marked, then *weight(i, G)* for the specific value of i has already been run since the state was changed, and the weight has been saved. In this case, the weight is read on line 27, and the value is returned. If the loop is entered, then the result should be calculated, which the contents of the loop handles.

The loop is a lowest-cost-first search of the graph G, searching for paths where the items in i is all collected.

On line 6-21, the next frontier to add to the set of frontiers is searched through. It finds a new frontier with the shortest route amongst all potential new frontiers that would collect any books in i that the current frontiers have not collected. In addition, if the weight cache element for the items collected in the frontier so far have to be updated, and the weight of the cycle that allows an agent to pick up these books are more efficient than the most efficient new frontier, the cycle is saved in c.

Then, the new frontier is added to the set of frontiers to be evaluated, to prepare for the next iteration.

On line 24-27, the books collected in a potential most efficient cycle has their corresponding weight cache element updated to not be marked, and their weight to be the weight of the cycle.

Finally, when the overall while-loop has been exited, the weight of the weight cache element for i is returned.

5.4 Warehouse Placement Algorithm

Warehouse placement is the problem of optimisation in placement of products in a warehouse[11]. This problem is specific from warehouse to warehouse. Normally, the placement is determined from a number of variables such as warehouse layout, product specifications, how the warehouse is operated, etc. In this project, the algorithm should work for all warehouses that contain books and are operated by people, though the solution can be integrated in an automated system operated by robots instead. The product specifications are also very static, since the project only concerns books, and the weight and size of different books are mostly the same.

Therefore, it should be possible to create an algorithm for optimal placement of books in the system.

The algorithm should be able to place a set of books at a time. When placing a set of books, a method is needed to find the book that should be placed next in the set of different books. This can help in optimisation of the state of the warehouse, since books that are closely related to many other books should be placed close to the drop-off point, and since the shelves around the drop-off point should not be saturated with books before the more important books get placed, the order of books placed should be shuffled too.

To achieve this, an algorithm is produced that prioritises books with a high amount of relations over books with a low amount of relations. The algorithm is shown in algorithm 2.

Algorithm 2: add_books(S_B)

```
1 $s$ := current state of the warehouse
2 $S_B$.sort()
3 for each Book $b$ in $S_B$ do
4     $s$.add_book($b$)
5     update_priorities($b$)
6     $S_B$.sort()
7 end
```

This algorithm requires each book to have a predetermined Priority B_P which is determined by the number of in-going relations of each book. The sort on S_B sorts by this priority B_P. The algorithm contains a for-loop that iterates on a set of books. For each iteration in the for-loop it places a book b, calls the subroutine seen in algorithm 3 with the same book as the argument, and lastly sorts the list of books by their priority.

Algorithm 3: update_priorities(Book b)

```
1 $b.Priority := b.Priority - k$
2 for each Relation $r$ in $b.Relations$ do
3     $r.Priority := r.Priority + 1$
4 end
```

The *update_priorities()* subroutine in algorithm 3 starts by reducing the priority of the Book b by the constant k. k is simply a constant and its value should be found through tests of the finished algorithm. After setting the Priority of b, the subroutine runs a for-loop where it increments priority value of each out-going relation.

The *add_book()* method places one specific book in the warehouse. This method uses a greedy descent approach to find the lowest local minimum, and places the book on the local minimum. This lowest minimum is found by moving a book around different shelves to find the lowest possible result of some evaluation function, to find a local optimum. This is done for each element in the calculation, to find a result that nears an optimal solution to the problem.

In the algorithm, I is the book that needs to be added to the system. Each accessible shelf space $H_1, \ldots, H_n$ is a node in the system, where n is the amount of shelves, and each shelf can have o instances of books, as described in section 2.4.

The greedy descent evaluation function is very alike the efficiency formula, described in section 5.1, as it has the same base functionality. It is as follows:

$$E(s) = \sum_{i \,\in\, \mathcal{P}(s_I)} c(i) \cdot w(i, s) \tag{5.7}$$

where s is the current state, or in other words, where the currently placed instances of books are located on the shelves. s_I is the types of books that currently have been added to the state.

Algorithm 4: add_book(I)

```
1  $s$ := current state of the warehouse
2  $H_i$ := clear shelf space closest to the drop-off point
3  continue := **true**
4  **while** *continue* **do**
5      continue := **false**
6      **for each** $h$ **in** $H_i$.*neighbours* **do**
7          **if** *$h$ has space for more books* **then**
8              **if** $E(s \cup \langle h, I\rangle) < E(s \cup \langle H_i, I\rangle)$ **then**
9                  $H_i := h$
10                 continue := **true**
11             **end**
12         **end**
13     **end**
14 **end**
15 $s := s \cup \langle H_i, I\rangle$
```

Algorithm 4 adds a book to the system on a shelf, using a greedy descent approach.

On line 2, it finds a clear space closest to the drop-off point, and uses this as the entry point for the greedy descent algorithm. Line 3 sets the *continue* variable that makes the algorithm loop on lines 4 to 14. On line 5, this variable is again set to false. It is expected this variable is set to true every time the algorithm decides to move the target shelf variable H_i to another shelf.

In the loop on lines 6 to 13, the algorithm enumerates through the neighbours of H_i, to find a better candidate for placement of the current book. This is done by first filtering the full shelves in the set of neighbours off, and then evaluating the $E(i, s)$ function with a state s that features the book on the neighbouring shelves, and comparing it to the same calculations if the book was placed on the H_i shelf.

If at least one better candidate shelf is found, H_i is set to be h, and *continue* is set to be true. The outermost loop then runs again, and checks for potentially better candidate shelves to place the book on.

Part II

Implementation

Chapter 6

Application Overview

To understand how the system has been implemented, it is necessary to have a general understanding of how the parts of the system work individually as well as how they work with each other.

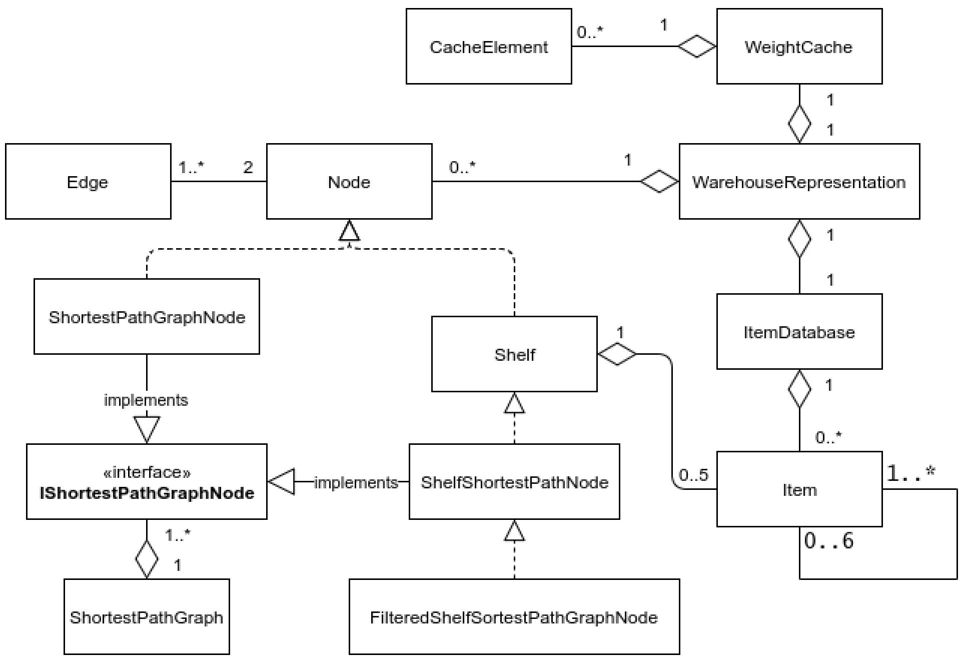


Figure 6.1: A diagram depicting the general overview over the structure of the server application

Figure 6.1 shows a general overview of the entire system, where the **Warehouse-Representation** holds information about the current warehouse. A set of **Node**s contained in the **WarehouseRepresentation** represent the walkable nodes in the system. Each **Node** can be a **Shelf** node, in which case it can contain a set of **Item**s. The shelves in the system, and the items that they contain is the warehouse configuration, and represents the state of the warehouse. **Edge**s from nodes to other nodes creates paths between each walkable node where agents can travel along.

Item has a relation to itself to represent item relations, where each item has between zero and six outgoing relations, and between 1 and any number of ingoing relations.

The **ItemDatabase** represent the collection of items that the **WarehouseRepresentation** can contain, and the items contained in the **ItemDatabase** is the only items that are allowed to be added to the system.

WeightCache is a cache of weights used when calculating weight in bulk. Each subset of items contained in the **WarehouseRepresentation** has an element in the **WeightCache**.

The **ShortestPathGraph** parts of the diagram, including all implementations of nodes that include the name, are all used when creating an all-pairs shortest path graph over the nodes of the system.

Chapter 7

Warehouse Representation

This chapter focuses on how the warehouse is represented in the server-side code. Some of the code that is important for the representation to work will be highlighted. Figure 7.1 shows an overview of the warehouse representation class. It tells what methods it has available, as well as what other classes and properties it uses to represent the warehouse.

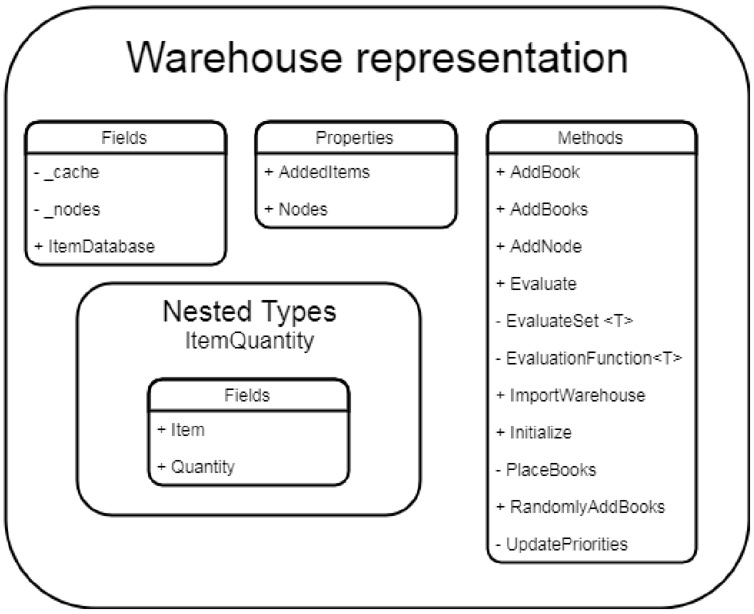


Figure 7.1: Classes used when representing the warehouse

A warehouse consists of nodes and shelves where the shelves themselves inherit from nodes and therefore are walkable. Shelves have a max capacity which is mirrored in the system, and we have chosen the max capacity to be 5. We have chosen to represent both nodes and shelves because we need to have nodes in the warehouse which represent the path and the distance the user should go in the warehouse. The shelves are used to contain the books and due to the placement policy you cannot be sure that a book of same type is in the same shelf as the others. Below in Listing 7.1 is the method that adds nodes to the warehouse. It shows how the warehouse graph is constructed.

```
public void AddNode(Node newNode, params int[] neighbourIds){
    if (_nodes != null && _nodes.Exists(n => n.Id == newNode.Id))
        throw new ArgumentException($"A node with the Id {newNode.Id}
            already exists.");
    if (_nodes == null)
    {
        _nodes = new List<Node>();
    }

    List<Node> neighbourNodes = new List<Node>();
    foreach (int id in neighbourIds)
    {
        neighbourNodes.Add(_nodes.Find(n => n.Id == id));
    }

    newNode.Edges = neighbourNodes.Select(n => new Edge<Node>() {
       @from = newNode, to = n, weight = newNode.EuclidDistance(n) })
        .ToArray();
    foreach (Node node in neighbourNodes)
    {
        node.Edges = node.Edges.Append(new Edge<Node>() { @from =
            node, to = newNode, weight = node.EuclidDistance(newNode)
            }).ToArray();
    }
    _nodes.Add(newNode);
}
```

Listing 7.1: This method adds the nodes to the representation of the warehouse

In Listing 7.1 when the method starts the first thing it does is to check if the list _nodes is not null and there exist a node in the list with a certain id. If this is true something went wrong and it has to throw an exception because a given node already exists. Next it checks if the list is null because then it should initialise the list. An empty list is constructed which is going to contain the neighbour nodes of the first node. The list is then filled with neighbour nodes if they match some id. The new node gets edges to its neighbours and calculated the distance between them. After this the neighbour nodes gets the same treatment. Finally the new node is added to the _nodes list.

When the warehouse graph is constructed it is possible to supply the shelves with books and calculate an optimal position of the books given the placement policy used in the system.

Chapter 8

Communication

We have chosen to implement the communication between the server and the client with sockets. The server is written in C# and the client is in Java, which both have implementations of sockets, via the Socket classes.

8.1 Client-Server Pattern

The criteria defined in section 1.1 specified that the system should be portable. To adhere to this criteria the system makes use of the client server pattern seen in Figure 8.1.

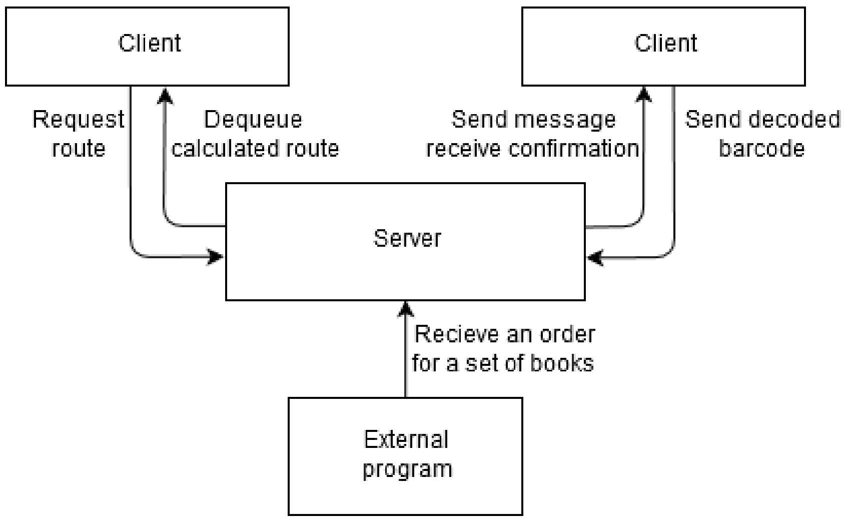


Figure 8.1: The client-server pattern as it is used in the system.

In this pattern the system is separated into a server where the pathfinding and sorting calculations as well as management of the warehouse is located, and several clients where each client can add instances of a book as well as obtain an order of books.

This is an implementation where the client can scan the barcode of a book and send the decoded information to the server through a socket(see section 8.2), and then the server will respond with a *Message received* confirmation. If the client is free to receive an order of books to collect from the physical warehouse, they can send a request to the server, who will then send a book order if any is available. In this implementation the server receives book orders from an external program. The instances of these books in the order are then found and removed from the system, and is placed in a queue ready for a client to receive.

The external program in Figure 8.1 is not actually implemented in the system since this system merely is for organising books in a warehouse, and not actually for handling actual orders for a library or bookstore. Currently orders can only be placed through the server's console user interface.

8.2 Socket

A socket is an endpoint for a connection between two nodes. The server may listen to the socket for an incoming connection, which is made by the client socket to a specific port. When the connection is established, the client communicates through its own socket to the server socket. In C#, the Socket class takes address family, socket type and protocol type as parameters. Address family tells the addressing scheme of the socket.

Since we need a connection to the internet the address family is set to 'InterNetwork' which tells the endpoint to expect an IPv4 address when a socket is connecting to it. Socket type tells what type the instance of the class should be. Socket type was set to 'Stream' which is a socket type that uses byte streams as a two-way connection. Protocol type specifies the protocols supported by the class [9]. Protocol type was set to 'TCP' because it is stable and it has a sorted order where the first package send is also the first package to be received[8].

8.3 TCP and UDP

Transmission Control Protocol (abbrv. *TCP*) is a protocol that treats its data as byte-streams. TCP has the benefit of a number of mechanisms that prevents duplication, loss of data and checks for errors and in case of errors it can resend the data and thereby guarantee that all data is correct. TCP checks if the transmitted data is received by the recipient within a certain timelimit. If not, it re-transmits the data. TCP needs to set up a connection before it is able to send data to the receiver. TCP is sequencing it packets so it will arrive in the same sequence as it was sent. This is not the case with UDP, which is an acronym for User Datagram Protocol. UDP is a very different internet protocol than TCP, because UDP is not sequenced and there is no guarantee that the packets are delivered, UDP does not need to be connected to the recipient to send data, and does not need acknowledgement from its recipient about reception of data. This means that UDP focuses mainly on speed. TCP has been chosen for this application because it is a reliable internet protocol which has to confirm before and after data is sent. In addition, it focuses on the integrity of the data rather than the arrival speed of the data.

8.4 Server

Using the .Net framework to create the server-side of the communication making the server able to receive new items from other devices which it can use to calculate an optimal placement.

```
public static void StartListening() {
    IPAddress ipAddress = GetIP();
    IPEndPoint localEndPoint = new IPEndPoint(ipAddress, 100);

    Socket socket = new Socket(ipAddress.AddressFamily,
        SocketType.Stream, ProtocolType.Tcp);

    try {
        socket.Bind(localEndPoint);
        socket.Listen(ConnectionQueueAmount);
        while (true) {
            AllDone.Reset();
            socket.BeginAccept(new AsyncCallback(AcceptCallback),
                socket);
            AllDone.WaitOne();
        }
    } catch (Exception e) {
        MessageLog.Add(e.ToString());
    }
    Console.Read();
}
```

Listing 8.1: The StartListning method.

Listing 8.1 is a method used for setting up the server. It starts out by getting all of the needed data: an IPaddress and a port. The IPaddress and port is first used to create a local endpoint which later is bound to a socket. When the socket is created the local endpoint is bound to the socket and the socket is set to listen for other devices trying to connect. `AllDone` is an event that blocks current thread and waits for signal from another thread to tell it to unblock it. While the thread is blocked the socket begins to connect to a found device. The thread is then unblocked when done.

```
private static void AcceptCallback(IAsyncResult ar) {
    AllDone.Set();

    Socket listener = (Socket) ar.AsyncState;
    Socket handler = listener.EndAccept(ar);

    StateObject state = new StateObject();
    state.workSocket = handler;
    handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0,
        new AsyncCallback(ReadCallback), state);
}
```

Listing 8.2: Accepts a connection from a client, prepares to receive data from the client and prepares for another client.

In Listing 8.2 `AllDone` allows all waiting threads gets to continue instead of blocking them. Then the socket connects to the client and begins to receive data.

The server makes use of asynchronous methods to handle the connection to other devices and thus avoiding halts in the program like the user interface not responding or the client not being able to connect to the server because the server was too busy to handle the requests at that time.

8.5 Client

The client is using the Java.net.Socket class making it able to send the data to the server. It is important that the client is coded in such a way that it does not have unnecessary power usage. This can be prevented by writing code that does not use busy waiting when listening to the server. This is done by essentially having the client attempt a connection to the server, sending some kind of message, waiting a short while for a response, and closing the connection. In an ideal situation, this will not take more than a fraction of a second, and so the client does need to maintain a connection all the time.

```
private void sendDataToServer(String data) {
    try {
        socket = new Socket(ip, port);
        PrintWriter writer = new PrintWriter(new
            OutputStreamWriter(socket.getOutputStream()));
        writer.print(data + "<EOF>");
        writer.flush();
    } catch (IOException e) {
        Log.e(TAG, e.getMessage());
    }
}
```

Listing 8.3: sendDataToServer is a method from the class ConnectionTask which handles connection to the server.

In Listing 8.3 the method creates a socket as well as an output stream, where all of the data that is sent from the client to the server is going through. The string has to be formatted before it is sent to the server so it is possible to know when the connection between the server and client should be closed. The string is formatted in such a way so it always ends with `<EOF>` because it compensates for the difference between Linux and Windows line endings.

```java
private String receiveDataFromServer() {
    StringBuilder response = new StringBuilder();
    try {
        BufferedReader reader = new BufferedReader(new
            InputStreamReader(socket.getInputStream()));

        for (String line = reader.readLine(); line != null; line =
            reader.readLine()) {
            response.append(line);
        }
    } catch (IOException e) {
        Log.e(TAG, e.getMessage());
    }

    return response.substring(0, response.length() - 5);
}
```

Listing 8.4: Method where the client listens to the server to check if it has sent a message

In Listing 8.4 the first thing the method does is to create an `InputStreamReader` that reads from the input stream of the socket. Then for all lines in the input stream append them, and remove the `<EOF>` symbol, since it is not relevant when the sever response is displayed.

8.6 Barcode Implementation

A large part of our solution is to scan barcodes. This is not an easy task, since it requires some kind of image recognition. We have chosen to use an existing library to scan and decode barcodes called ZBar. This section focuses on how ZBar works in our Android application. It will be a high-level overview, and no code will be shown.

Video Input

The first step is to produce a stream of images. This is handled by Android's camera API, beginning when the user presses a specific button when starting the application.

Optionally, the input is sent to an output window. This is in our case also handled by Android to display the input on the screen.

Image Scanner

The stream of images produced by the video input is sent to the ZBar image scanner. This stream is converted to an intensity sample stream - a list of values between 0 and 255 corresponding to black and white, respectively. This stream is not sorted into individual greyscale images, as patterns will be detected in a later step.

Linear Scanner

The linear scanner takes the intensity sample stream and uses signal processing to detect individual bars. These bars are then streamed to the decoder.

Decoder

The decoder takes as input the bar-width stream and searches for patterns in different barcodes. When it detects a familiar pattern, it decodes the stream to a string which is sent to the server for further processing. When a familiar barcode is detected, the steps taken to decode is shown in chapter 4 which shows how to decode an EAN-13 barcode.

Chapter 9

Algorithms

In this chapter, the implementation of the most important algorithms are detailed, to offer a better understanding of the underlying system.

9.1 Importance Coefficient

The implementation of the algorithm for calculating c as mentioned in section 5.2 can be seen in listing 9.1.

```
public static float ImportanceCoefficientAlgorithm(List<Item>
    setOfItems){
    float a = 0; // Number of Arcs
    int n = setOfItems.Count; // Number of nodes in the setOfItems
    foreach (Item item in setOfItems)
2    {
        a += item.Relations.Count(setOfItems.Contains);
    }
    return a / n;
}
```

Listing 9.1: Implementation of importance coefficient

This is the primary algorithm for calculating the importance coefficient. The algorithm takes two arguments; a list of items which is the set of items that the coefficient will be applied to, and a list of arcs which is all of the relations between the books in the entire system.

The algorithm contains a for loop that loops for each item in the set of items. Adds the number of relations from one item in the set of items to any other item in the same set to the variable a which contains the number of arcs in the set.

At the end of the algorithm the number of arcs in the set is divided with the number of items in the set and this value is returned.

9.2 Weight

The weight algorithm requires an all-pairs shortest path graph of the dropoff point, and all the shelves in the warehouse. In the implementation, this is known as the distance map, which will be described before the weight algorithm can be detailed.

9.2.1 Map of Distances

The map of distances class is directly comparable to the matrix M in section 5.3. It holds the information for distances between a set of nodes. The implementation of the method that calculates each distance to and from each relevant node are as shown in Listing 9.2. It implements a modified version of Dijkstra's algorithm to find all relevant distances to and from a node.

This implementation of Dijkstra's algorithm is different form the classical implementation in that the classical implementation finds a route from point A to point B, whereas Listing 9.2 finds the distance from a point A to a set of different points $B_1, \ldots, B_n$.

```
private void DistancesFromNode(Node node)
{
    Dictionary<int, float> nodeSubDictionary;
    if (_dictionary.TryGetValue(node.Id, out nodeSubDictionary))
    {
        Dictionary<Node, DijkstraData> dijkstraDict = new
           Dictionary<Node, DijkstraData>();
        foreach (Node n in _fullGraph)
        {
            dijkstraDict.Add(n, new DijkstraData()
            {
                Distance = float.MaxValue,
                Visited = false,
            });
        }
        dijkstraDict[node].Distance = 0;

        Node currentNode = node;

        while (nodeSubDictionary.ContainsValue(float.MaxValue))
        {
            foreach (Edge<Node> edge in currentNode.Edges)
            {
                float dist = edge.weight +
                   dijkstraDict[currentNode].Distance;
                if (dist < dijkstraDict[edge.to].Distance)
                {
                    dijkstraDict[edge.to].Distance = dist;
                }
            }

            float dummy;
            if (nodeSubDictionary.TryGetValue(currentNode.Id, out
                dummy))
            {
                nodeSubDictionary[currentNode.Id] =
                    dijkstraDict[currentNode].Distance;
            }
            dijkstraDict[currentNode].Visited = true;

            float minDistance = float.MaxValue;
            foreach (KeyValuePair<Node, DijkstraData> pair in
                dijkstraDict.Where(pair => pair.Value.Visited ==
                false))
            {
                if (pair.Value.Distance < minDistance)
                {
                    minDistance = pair.Value.Distance;
                    currentNode = pair.Key;
                }
            }
        }
    }
}
```

Listing 9.2: A map of distances between an array of nodes.

The method takes a node as parameter, and calculates the distance to each other node, and applies it to the dictionary `_dictionary`. `_dictionary` is the dictionary that holds the weights of each node pair. Each element in the dictionary has an integer value as key, and a subdictionary as value. The integer key describes the ID of the mapped from-node, and is used when looking up distances in the resulting distance map. The subdictionary of each element is in itself a directory, with the mapped to-node's integer ID as key, and a floating point number value as weight between the mapped from-node to the mapped to-node.

The dictionary are initialised to n different dictionaries, where n is the amount of nodes that the distance map are interested in. The key for each subdictionary in `_dictionary` are the ID's for each node that the distance map should find distances from and to, and the length of each subdictionary is the number of remaining nodes, initialised with the key as ID of the nodes as well.

The value of the subdictionaries are then initialised to be infinite.

	subdictionary				
`_dictionary`	id_1	id_2	id_3	$\dots$	id_n
id_1	∞	∞	∞	$\dots$	∞
id_2		∞	∞	$\dots$	∞
id_3			∞	$\dots$	∞
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
id_n				$\dots$	∞

Table 9.1: The initial value of `_dictionary`, where each row is a subdirectory, and n is the amount of nodes.

In the algorithm, a subdirectory is found based on the node, and the weights for the distances from the node to each necessary other node is found.

On line 6, a temporary dictionary is created to contain the necessary information to run a Dijkstra-style graph search. The key for each data point is a node, and the value is instances of the `DijkstraData` class. Each instance contains a boolean value `Visited`, that describes whether or not the node has been visited in the Dijkstra search, and a floating point value `Distance` that determines the distance from the initial node.

On line 7-15, the new dictionary is populated with data for each node in the full graph. The distance to the initial node is set to 0, since a node has 0 distance to itself.

On line 17, `currentNode` is set to be the initial node. This will be the node where calculations are based on.

The loop on line 19-46 contains the main calculations needed to find the shortest path from the initial node to each other node we are interested in. It loops until all values in `nodeSubDirectory` has been updated to be different from the max value of floats. As soon as each value in the subdirectory, each necessary node has been visited, and the calculations can stop.

In the loop on line 21-28, the edges from the current node is inspected, and if the distance from the current node, plus the distance to the initial node is lower than the value currently contained in `dijkstraDict` on the value addressed by the node where the edge points to. If it is, when the saved distance is set to be the newly calculated distance.

On lines 30-34, the current nodes are checked to be one of the requested nodes, as contained in `nodeSubDirectory`. If the subdirectory has a key equal to the ID of the current node, set the value addressed by the key to be the distance saved in `dijkstraDict`.

On line 35, the visitation of the current node is finished, so the current node's data element in `dijkstraDict` is set to have visited to be true.

On lines 37-45, a new current node is set to be the node with the lowest distance to the initial node, that has not already been visited.

This method is then run once for each requested node to find all relevant distances.

9.2.2 Weight

The main weight algorithm is very alike the pseudocode algorithm for weight in section 5.3, and a detailed description here would repeat the information found in the pseudocode section. Instead, the implementation for the weight algorithm is detailed in the appendix (C), if a detailed description of the code is required.

9.3 Placement Algorithm

In this section, the placement algorithm will be described. The placement algorithm makes use of a private class `ItemQuantity`. `ItemQuantity` holds two fields `item` and `Quantity` where `item` is of the `Item` type and `Quantity` is an integer value. This allows the `ItemQuantity` type to be a denominator for when more than one instance of a type of book is to be placed, and is used to pass sets of the same item to the placement algorithm to be placed in the warehouse.

```
private void PlaceBooks(ItemQuantity[] itemQuantities){
    ItemQuantity targetItem = null;
    int maxPriority = int.MinValue;
    foreach (ItemQuantity iq in itemQuantities.Where(iq =>
        iq.Quantity > 0))
    {
        if (iq.Item.Priority > maxPriority)
        {
            targetItem = iq;
            maxPriority = iq.Item.Priority;
        }
    }
    if (targetItem == null)
    {
        return;
    }
    AddBook(targetItem.Item);
    UpdatePriorities(targetItem.Item);
    targetItem.Quantity--;
    PlaceBooks(itemQuantities);
}
```

Listing 9.3: This is the placement algorithm that calls all of the needed auxiliary methods.

Listing 9.3 is an implementation of the pseudocode for the placement algorithm algorithm 2, but the implementation is significantly different from the pseudocode, since a direct implementation would be inefficient.

The approach in the code is to find the item type with the highest priority, and then add the book by type. When the book has been added, the quantity of books that needs to be added of that specific type has to be subtracted by one, since one of those books have now been added to the system. Then, the method is called recursively, until all `itemQuanities` have had their individual quantities set to 0. When this has happened, `targetItem` does not get set to a different value than null, and the method returns on line 14.

Whenever a book has been placed, the priorities are updated from the given type of book. The priority update function is very alike the pseudocode implementation, algorithm 3.

```
 1 public void AddBook(Item item){
 2     while (cont)
 3     {
 4         cont = false;
 5         FilteredShelfShortestPathGraphNode[] neighbours =
               ((Node)currentNode).Neighbours.Where(n => n is
               FilteredShelfShortestPathGraphNode)
               .Cast<FilteredShelfShortestPathGraphNode>().Where(n =>
               n.Capacity > 0).Take(5).ToArray();
 6
 7         foreach (FilteredShelfShortestPathGraphNode neighbour in
               neighbours)
 8         {
 9             if (markedNodes.Contains(neighbour))
10             {
11                 continue;
12             }
13
14             neighbour.AddFilteredItem = true;
15             float eval = EvaluationFunction(filterShortestPathGraph,
                   itemSets);
16             neighbour.AddFilteredItem = false;
17
18             if (eval < lowestEvaluation)
19             {
20                 currentNode = neighbour;
21                 lowestEvaluation = eval;
22                 cont = true;
23             }
24             markedNodes.Add(neighbour);
25         }
26     }
27     ((Shelf)currentNode.Parent).AddBook(item);
28     _cache.MarkItem(item);
29 }
```

Listing 9.4: This method finds the optimal position for a given book and adds the book to the shelf on that position.

Identifier		Denotes
`filterShortestPathGraph`	:	A temporary graph of `FilteredShelfShortestPathGraphNode`s and the dropoff point, with all nodes having an all pairs shortest path connections to each other. The evaluation function is evaluated on this graph.
`currentNode`	:	The currently most efficient node to place a book on. Initialised to the dropoff point.
`itemSets`	:	The sets of items that need to be evaluated on in the evaluation function.
`lowestEvaluation`	:	The lowest evaluation score measured. Initialised to the maximum value of a float.
`markedNodes`	:	The list of all nodes that already has been evaluated on. There is no reason to evaluate on the same node twice, as the score of evaluation would be the same result, so a list of evaluated nodes are kept to keep track of this.

Table 9.2: The variables declared before running Listing 9.4.

Listing 9.4 makes use of some variables that is initialised before the calculations in the algorithm is run. An overview of the variables are given in Table 9.2.

The `FilteredShelfShortestPathGraphNode` is a node in an all-pairs shortest path graph. In addition, it is an abstraction on a shelf node, that filters all instances of a book away from a parent shelf, unless the book is specifically placed on the shelf during the placement function. The filtered away book on each shelf are the book type that currently is in the process of getting added to the system when `AddBook()` is being run.

`AddBook()` contains a while loop, that runs until the most efficient local minimum is found.

The while loop is controlled by a boolean variable named `cont` which determines whether the loop continues or not. The loop should continue until no further more efficient shelves are found.

The `neighbours` variable are the neighbouring `FilteredShelfNodes` of `currentNode`, that has capacity for new books. Only the closest five shelves are included and evaluated on, since the neighbours are ordered in ascending order of weight.

Then for each `neighbour` in `neighbours`, add the item that was originally filtered away to the shelf for evaluation.

The graph is then evaluated, and the item is the removed from `neighbour`. The result is compared to the current `lowestEvaluation`. If it is lower than the `lowestEvaluation`, then `lowestEvaulation` will be set to the result, and the `currentNode` is set to be the `neighbour`, and `cont` is set to `true`, continuing the while loop, and evaluating the neighbours of the new `currentNode`.

The `neighbour` node is then marked to prevent evaluation of the node twice.

After the while loop is done, the `currentNode` has been set to be the local minimum, and the book is added to the shelf.

As the last operation of the method it marks the item in the cache, making sure the weight of the item gets updated in the next evaluation of the weight of the item.

Part III

Conclusion

Chapter 10

Placement Algorithm Efficiency Evaluation

In this chapter, the efficiency of the book placement algorithm is compared to the algorithm employed at Amazon's warehouses. In order to determine which of these approaches is the most optimal, certain tests are performed on each of them.

To test the efficiency of each algorithm, the following function will be used:

$$F(s) = \sum_{i \subset items} c(i) \cdot w(i, s) \tag{10.1}$$

Where s is the resulting state from running the different placement algorithms. The closer F is to zero, the less distance (weight) is between the items in the warehouse that are related, where the magnitude of these relations are taken into consideration. So this equation produces a number that represents how efficient the state of the warehouse is, and how well the books have been placed.

In the test environment, each shelf can contain five instances of items, and three different sizes of warehouses are evaluated. The small sized warehouse has three shelves to place items on, the medium sized one has seven shelves, and the large size warehouse has fourteen.

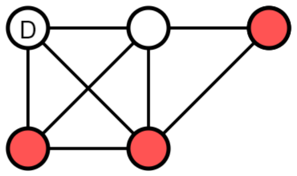


Figure 10.1: The small warehouse variant. The node marked with a **D** is the dropoff point, white nodes are walkable paths, and red nodes are shelves.

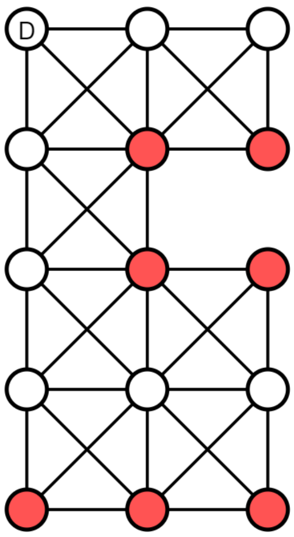


Figure 10.2: The medium sized warehouse variant.

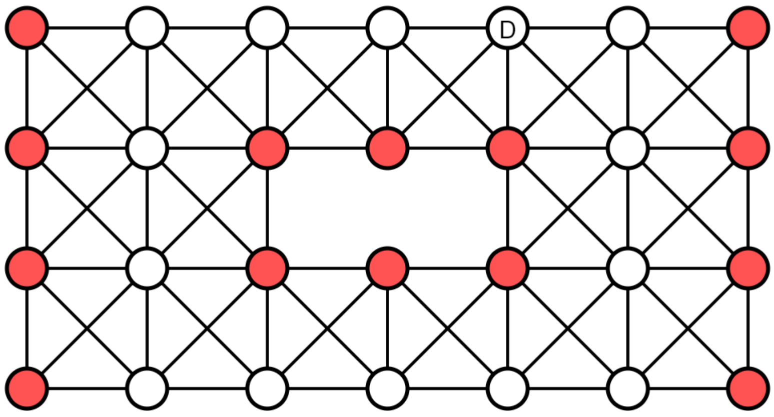


Figure 10.3: The large warehouse variant.

10.1 Amazon's Random Placement Policy

The algorithm that this project is competing against is Amazon's random placement technique, where employees place wares randomly in the warehouse. Since Amazon has coworkers placing the items randomly and not an actual algorithm, a pseudo-random placement algorithm was compared with instead.

This algorithm places each item on a random shelf. The shelves are always shelves that still has shelf space remaining, and one of the three shelves closest to the dropoff point, since it makes no sense to place items on shelves that are far away from the dropoff point, when the warehouse is mostly empty.

Since this algorithm is based on a pseudo-random element, 10 different readings are made with different seeds, and the average is compared with the book placement algorithm.

10.2 Book Placement Algorithm

The *book placement algorithm* uses a greedy approach to find where each book is best placed in the warehouse.

In the book placement algorithm, a constant k need to be defined (see section 5.4). The value is specifically defined to be 6, since this is lowest amount that ensures that the books on each shelf is varied, which creates a well-optimised state. This value of k was also tested against other specific values of k to yield a good state of the warehouse.

10.3 Comparing the Test Results

The $F(s)$ values are determined on the state that the two different algorithms yield, and the results of comparisons on different size warehouses are presented in Figure 10.4.

As is evident in the figure, the state of the warehouse is better optimised using the book placement algorithm rather than the random placement algorithm. As the warehouse scales up, the efficiency of the random placement algorithm degrades, while the efficiency of the state that is yielded by the book placement algorithm remains rather efficient.

This shows that the book placement algorithm is effective at placing a sufficiently varied set of books close to the dropoff point, as the time it takes for the agent to pick up the books remains low, even as the size of the warehouse scales up.

The random placement algorithm ends up not placing the books as efficiently on the shelves when the warehouse scales up, which is an indication that books end up being placed far away from books that has a strong relation.

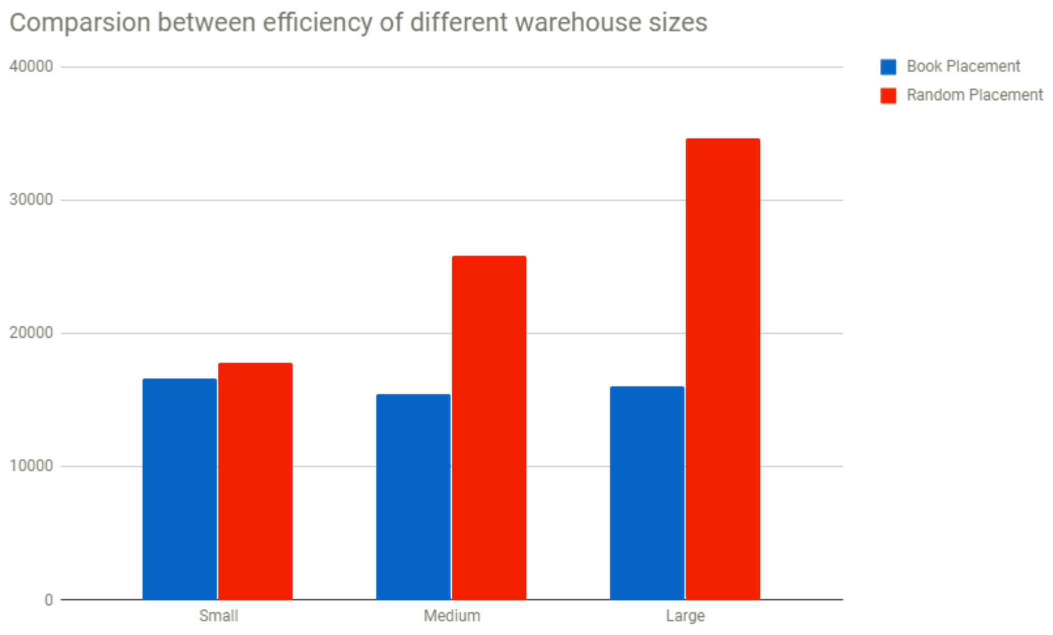


Figure 10.4: The efficiency of the book placement algorithm in comparison to a random placement algorithm. The comparison is made on three different sized warehouses. Lower is better.

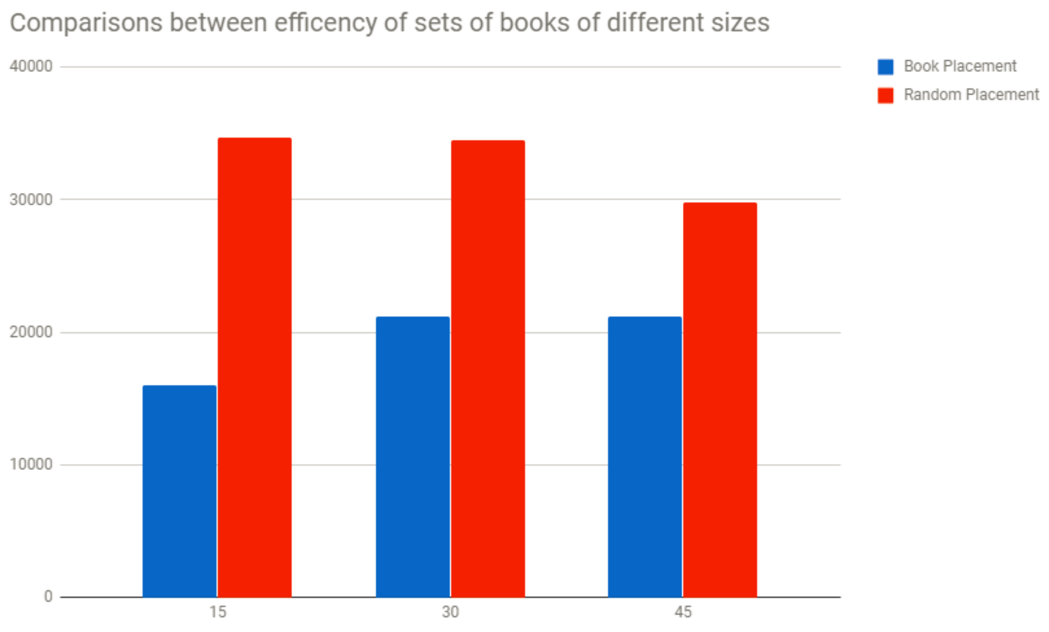


Figure 10.5: The efficiency of the book placement algorithm in comparison to a random placement algorithm. The comparison is made on three different sets of books, with different sizes. This comparison was made on the large warehouse variant. Lower is better.

Comparisons are also made on different sets of books, and the efficiency of the resulting state of the warehouse when run with both the random placement algorithm, and the book placement algorithm, and the results are shown in Figure 10.5.

According to the figure, the book placement algorithm has its performance reduced when run with more than the initial 15 books. The efficiency goes from being 16000 to being 21000, and then reaches a plateau, where the performance is not being reduced further. The initial bump in efficiency is a sign of over saturation of the concentration of books, pushing relevant books further form the drop-off point. After reaching the plateau however, no relevant books are added close enough to make a difference in the evaluation function.

You could argue that other factors could be affecting the efficiency of the algorithm such as the order in which the books are placed. The order in which the books are added to a warehouse could affect the efficiency of the warehouse, since a book could be placed on one shelf if another book was placed in the warehouse, but on another if that book had not been placed there yet. We did, however, not perform any tests on this and as such have no actual data to compare with.

With the random placement algorithm, the performance is improved when being run with a higher number of books. This is evidence towards the inefficiency of the run with the low amount of books being made up for with a higher amount of books to sample from. The random placement algorithm might approach the efficiency of the book placement algorithm even further, since the book placement algorithm is dependant on the relations of the books and these relations can change, while the random placement algorithm does not have this dependency. The random placement algorithm will not likely become more efficient than the book placement algorithm, since the random placement algorithm will have the same problem with pushing relevant books further out. It will not be possible for both algorithms to have all relevant books concentrated close to the dropoff point, because of a lack of space.

In addition, it is possible to conclude that the book placement algorithm yields a warehouse state that is on average 66% more efficient than its counterpart, on the tests run.

Therefore, it is all in all possible to conclude that the book placement algorithm is more efficient than an algorithm that places the books arbitrarily, especially on large warehouses.

10.4 Evaluated Efficiency versus Running Time of Placement Algorithms

The book placement algorithm yields an efficient warehouse, but in a real world scenario, where the placement algorithms would be implemented in a warehouse administration system, the running time of the algorithm need to be considered as well.

The random placement algorithm is significantly faster than the book placement algorithm. In the testing environment with a large warehouse and 45 books to place running on an Intel Core i5-3570 processor, the random placement algorithm was finished practically instantly, whereas the book placement algorithm took about 20 hours to complete.

In a real world system, this running time is unacceptable, especially since the warehouses will in most cases be bigger than the large test warehouse, and a higher amount of wares to place, which makes the system unusable unless the code is optimised significantly.

Chapter 11

Unit Testing

Unit testing is a strategy to make sure that the implementation of the different algorithms and models contain as few bugs as possible in the tested parts of the code. Testing is focused on important parts of the code and parts that are most prone to error, to make sure the core functionality of the code base works as intended.

In this project the tests were written after writing the code and it could thus be said that the tests might be fitted to the code rather than fitting to the expected results. We have, however, tried to remain unbiased when writing these tests and the tests have also led to changes in the actual code.

An example of such a test is the test on the AddBook method in Warehouse-Representation, where the method is given a book that does not exist in the current database. The test expected the method to throw an exception since it would not recognise the book, however, the method accepted the book and added it to the representation, causing the test to fail. This test thus let to the addition of an additional check in the AddBook method, that prevents unintended behaviour.

In general the unit tests were set up to test whether a valid input gave the correct output as well as if incorrect input gave the expected errors. Thus the tests had a focus on both error handling as well as correct output.

Overall, the unit test coverage is 57%, a rather high amount, considering the unit tests only focus on the most important parts of the code, and no unit tests has been created for the code that manages the UI. When the UI is excluded, the percentage lies on 75%.

The weight algorithm is one of the most important parts of the code, as it is used both when placing books and when evaluating the state of the warehouse. Another important algorithm is the importance algorithm, which is also used for both placement of books and evaluation of state in the warehouse. The last important algorithm is the AddBook algorithm that is used whenever the server receives a decoded barcode from a client. The coverage of all three of these important algorithms have a coverage of 100%.

Aside from unit tests where we just tested individual units in the system, we also made system tests where we ran the server and executed the different commands on the server. When running these tests we uncovered exceptions that were not caught and thus the server crashed.

This led to a lot of exception handling in the Controller since the server should rather send a message to the user when an illegal action occurred rather than crash.

Chapter 12

Discussion

This section focuses on some of the choices that have been made in the project, and reflect upon whether or not the choices made were correct.

12.1 Barcode

In the beginning stages of the project we wanted to use QR code for our warehouse system, but because the books in librariesalready uses EAN-13 to encode each book's ISBN, we went with EAN-13 instead of converting the ISBN to QR codes. Quite some time was spent trying to understand how QR codes work and how to read them using a scanner.

is this true

The choice was between EAN-13 and QR, described in sections 4 and 4.2. We needed a simple printed code that contained some information about a book, and EAN-13 fit the requirements for the project.

Tests were conducted on EAN-13 to determine how much of the code had to be visible for the scanner to able to read it correctly. As is described in section 4.3, the scanner was able to scan the codes even if they were almost completely obscured - as long as the scanner was able to see all the bars and spaces of the barcode, the code could be scanned and decoded.

As mentioned in 4, we are using barcode type EAN-13 which is the barcodes that are most commonly printed on books. Another possibility was to use the MSI barcode type which is a barcode commonly used in warehouses to organise inventory on shelves [14], but since the focus of this project is on books, we instead chose to focus on EAN-13.

Since part of the solution should fit on an embedded system, it was not feasible to chase the most flexible and most secure implementation of printed codes and implement QR codes, and instead implement one of the more lightweight barcodes. The EAN-13 is simple and easy to understand, and it meets the requirements that our project group established for this project.

12.2 Algorithms

12.2.1 Different Importance Coefficient Algorithms

Before we ended up with the current Importance Coefficient we had some alternative ideas. One of the alternate methods of calculation was:

$$c(r) = \frac{a}{m-a} \tag{12.1}$$

where m = maximum number of arcs and a = actual number of arcs, but we chose the current definition over this one because in the case where the set was a single item the calculation would give $c = \frac{0}{0}$ and that would produce errors.

Another way we considered calculating c was by calculating the smallest sub-tree between nodes in a set and counting the nodes, however it did not distinguish between connected points in sets and unconnected points in sets. Figure 12.1 shows two sets: {1,4} and {1,3,4}. In this example both sets would have the rating of 3 even though set {1,4} would make use of a node that was not part of the subset.

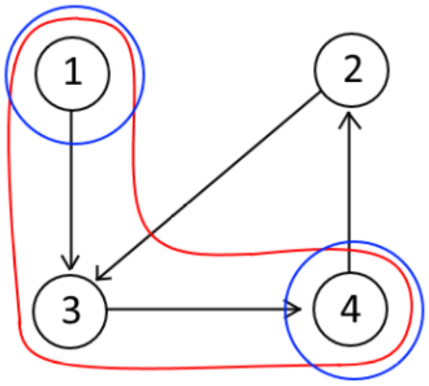


Figure 12.1: Illustration of an alternative to the importance formula. The graph has two sets: blue:{1,4} and red:{1,3,4}. It calculates the smallest sub-tree, but it does not not distinguish between connected points in sets and unconnected points in sets.

Since a customer would not be more likely to buy book 4 if he or she only buys book 1, according to the graph, it would not make sense to rate the two subsets the same. The other formula was chosen over this, since distinguishing between unconnected points and connected points is useful to make connected points rate higher.

12.2.2 Map of Distances

As explained in subsection 9.2.1, we need a so called "map of distances": a matrix that holds information about the distance between a set of nodes. For this, we employed a modified version of Dijkstra's algorithm. Another algorithm we considered at an earlier stage was Floyd Warshall's algorithm.

Floyd Warshall's algorithm has a time complexity of $\mathcal{O}(n^3)$. We have managed to create the modified version of Dijkstra's algorithm with a time complexity of $\mathcal{O}(n^2)$, as it can be seen in Appendix B. However, our modified version of Dijkstra's algorithm only computes the distance from one start node, to all other given nodes.

Another alternative to Dijkstra's and Floyd Warshall's algorithm is the A^* (A Star) pathfinding algorithm. The time complexity of A^* depends largely on the heuristic function. Which purpose is to make an estimate of a distance from any given node to another. The time complexity of A^* is known to be $\mathcal{O}(b^d)$, where b is the **branching factor**, which is the average number of successors per state. This algorithm is guaranteed to find the optimal path. However, this also means that the A^* will have to examine all paths that have a chance to be the optimal path. As the A^* algorithm takes a start and a goal node, in our case the time complexity will have to be increased even further, we would need to run A^* from every single node to every other node, which in practice, means that A^* would have to be run n^2 times. So the resulting time complexity of A^* in our case, is $\mathcal{O}(b^d * n^2)$. In most cases, this would be a better result, consisting of only optimal paths. But the increased cost of time is simply not worth it.

12.2.3 Book Placement Algorithm

The book placement algorithm is currently not fast enough to be used in a real warehouse, with some of the tests running as long as 20 hours each (section 10.4), so a more efficient book placement algorithm is needed for the system to be usable in a real world scenario.

The evaluation function is one of the parts of the algorithm that can be optimised. Currently, the newly added item is evaluated with the overall efficiency of all relations that include the item, where importance is more than 0. An alternative evaluation function would only evaluate with the sets that include the newly added item, and one other item. This way, instead of evaluating the weight function for each subset that contain the item in the powerset of all items, only the subsets that contain two items including the newly added item is evaluated, significantly reducing the amount of times the weight algorithm need to be run, and still placing the new item on a good position, since all highly related items are still considered when evaluating the state.

12.2.4 AddBook Optimization

This method is partially responsible for book placement. In order to decrease the extreme running time of our Book Placement Algorithm, we discovered a better alternative to the *markedNodes* list, that we use in the *AddBook()* method. This list is responsible for keeping track of which nodes we have already investigated, so we avoid re-visiting nodes again. The *markedNodes* variable is of type *List*, and since the only methods being accessed on the *List* class are *Contains()* and *Add()*, the required time to complete would in most cases be improved by using a *HashSet* instead. A *HashSet* implementation would improve the total run time to be more efficient than a *List* implementation, as the size of the simulated warehouse increases.

This is simply because the time complexity of of *List.Contains()* is $\mathcal{O}(n)$, where the *HashSet.Contains()* is only $\mathcal{O}(1)$, which is a vast improvement, especially because this method is executed in nested loops many times, and the *markedNodes.Contains()* call, itself is also in the innermost loop of the *AddBook()* method.

12.3 Evaluation Technique

The basis of the project was to create a placement algorithm that is an improvement to what is known of Amazon's placement algorithm, and as such the evaluations are done in comparison to an algorithm that is assumed to be like Amazon's, using random placement of books.

To achieve more accurate evaluations of the placement algorithm, it could have been compared with an algorithm exactly like what Amazon use in their warehouses, but since Amazon do not use an actual algorithm but rather a placement policy or technique, an algorithm that was exactly like Amazon's was not possible to produce. Therefore, comparing with Amazon's exact algorithms was impossible, and thus achieving accurate results was challenging.

12.4 Future Works

In the future there are a few things we would like to improve. Most importantly, the book placement algorithm is in vital need of optimisation. The current case, where the run time climbs to 20 hours before reaching completion, is making our solution unusable in many cases.

Currently the shelves in the system all have a max capacity of five books. It is currently not possible to change this in the system. In the future it would make sense to change this since it is very uncommon for a bookshelf to only hold a maximum of five books, thus this extended functionality would be sensible to add in the future.

Furthermore, we would like to implement the capability of scanning all bar-codes for all books. It is currently only possible to scan the bar codes of a few books we have chosen to use for the demonstration of our solution. This is an easy but time consuming implementation since the system already supports ISBN and thus the only fix is expanding the database with additional ISBN.

Another useful feature we would like to implement, is the ability to look up all information about a book purely from its ISBN. This could be accomplished by connecting to Amazon's website whenever a ISBN is scanned. This could, however, be rather difficult to implement since Amazon tries to block any crawlers from obtaining data from the site.

Another thing that could be changed in the future is the static class *Algorithms* which currently act as a wrapper of certain parts of the system such as the *Importance* and *Weight* algorithms that instead could be moved to other parts of the systems. It could be argued that this static class is not an object oriented way of programming and thus the elements contained in *Algorithms* should be moved to better fitting places in the system.

Chapter 13

Overall Conclusion

Based on our problem formulation, we investigated whether we have met the requirements of the project. The problem formulation is shown below.

- *How can an assortment of items be placed in a warehouse in a way that an employee will be able to collect a set of items fast and efficiently in the most probable cases, so that the collection process will be, on average, as quick as possible?*

To fulfil this problem statement, the items need to be placed on the correct positions, then an optimal route should be calculated, and then the route should be communicated to the warehouse employee.

The system uses a greedy descent approach to place books on positions in the warehouse that approach optimal positions. Employees in the warehouse can scan the barcode of books, to let them know where the book should be placed. To find the optimal route when collecting books, a weight algorithm on the server searches through the graph in a lowest-cost-first search, and finds the shelves where an agent should pick the specific books up that they are looking for. The route is communicated to the employee using sockets, and represented on the hand held Android client that they are equipped with.

In general, the problem formulation has been met as we have developed a embedded system on an Android smartphone that communicates with a server, and can calculate an optimal route in a warehouse for specific items. But our placement algorithm is rather slow. One of the evaluations had a calculation speed of 20 hours, and as such, if it should be used in a warehouse, it should be optimised first. Some options of optimisation are described in subsection 12.2.3 and subsection 12.2.4 which, if implemented, would reduce the time spent calculating the placement of books.

We can conclude that our placement scores higher than the random placement algorithm on the scale of most optimised placement as you can see in section 10.3, but as stated above it is much slower than the random algorithm. This raises the question whether you want a fast algorithm with a lower optimisation score, or if you want a slower algorithm which has a better placement.

With some optimisations, it might be possible to implement the system in a small warehouse, but due to the nature of small warehouses the optimisation of placement of items in the warehouse is less important, since routes would not have the need to be optimised as much. In addition, the unreadability of positions of items might be too large a price to pay in a real warehouse.

It can be concluded that this project meets the requirements we set for the project, in regards to the embedded system. The Android smartphone acts as a client that is an embedded system that can communicate with a server, and fulfils all requirements.

Bibliography

[1] Raj Arjit. *How do barcodes work?* https://www.quora.com/How-do-barcodes-work-1. [Online, accessed 19-11-2017].

[2] Jeff Brown. *Types of Barcodes: Choosing the Right Barcode.* http://zbar.sourceforge.net/iphone/s [Online, accessed 12-12-2017].

[3] KEYENCE CORPORATION. *What is a QR code?* `http://www.keyence.com/ss/products/auto_id/barcode_lecture/basic_2d/qr/`. [Online, accessed 29-09-2017].

[4] *EAN/UPC barcodes.* https://www.gs1.org/barcodes/ean-upc. [Online, accessed 27-11-2017].

[5] *Familiernes besiddelse af elektronik i hjemmet efter forbrugsart og tid.* `http://www.dst.dk/da/Statistik/emner/priser-og-forbrug/forbrug/elektronik-i-hjemmet`. [Online, accessed 22-10-2017].

[6] Gsmarena. *Huawei Ascend Y530.* https://www.gsmarena.com/huawei_ascend_y530-6103.php. [Online, accessed 13-10-2017].

[7] Barcode Island. *EAN-13 BACKGROUND INFORMATION.* http://www.barcodeisland.com/ear [Online, accessed 19-11-2017].

[8] John Kristoff. *The Transmission Control Protocol.* https://condor.depaul.edu/jkristof/technotes/ [Online, accessed 06-11-2017].

[9] microsoft. *Socket Class.* https://msdn.microsoft.com/en-us/library/system.net.sockets.socket(v= save-lang=1&cs-lang=csharp#Remarks. [Online, accessed 31-10-2017].

[10] Amit Patel. *Map Representation.* `http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html`. [Online, accessed 29-09-2017]. 2017.

[11] Dave Piasecki. *Warehouse Slotting.* `www.inventoryops.com/articles/warehouse_slotting.htm`. [Online, accessed 09-10-2017]. 2017.

[12] David L. Poole and Alan K. Mackworth. *Artificial Intelligence: Foundations of Computational Agents.* 2nd ed. Cambridge University Press, 2017.

[13] Rashberrypi. *RASPBERRY PI 2 MODEL B*. https://www.raspberrypi.org/products/raspberry-pi-2-model-b/. [Online, accessed 13-10-2017].

[14] Scandit. *Types of Barcodes: Choosing the Right Barcode*. https://www.scandit.com/types-barcodes-choosing-right-barcode/. [Online, accessed 06-12-2017].

Appendix A

Data Collection

For the relation graph to be accurate, it needs to have the correct relations between books. This graph should be sufficiently large, and for the size that the project requires, a manual approach would be unrealistic. Instead, an automated solution was found.

The solution was an automated crawler that automatically scrapes relations between books off Amazon's store pages. For each product on Amazon, the user is recommended a number of other products that buyers of the original product often also bought. The crawled books are used as nodes in the relation graph, and the recommended books are represented as directed arcs between nodes. These arcs are used to calculate the importance each book have to each other. The higher importance that a number of books have to each other, the closer they should be located in the final warehouse representation, to get the best possible layout.

The crawler uses algorithm 5, algorithm 6 and algorithm 7 to collect data and put the data in database files. It uses the *scrapy* framework to crawl Amazon's store pages, which uses the method *scrapy.request()* to create a request for a page, and returns a response once the HTML request responds.

algorithm 5 is the main algorithm that parses web pages. On line 1, the name of the book is parsed from the page, by referring to the page item with ID `productTitle`, and getting the text in the item. In same fashion, a divider with ID `tmmSwatches` define the formats that the product has. The formats are parsed to check if the product comes as a book.

The *carousel_books* variable contains the recommended books for the current product. It is a list of 6 items, including a link to the item, and the name of the book. On line 5-10, actions are performed on each book. The link of each book is added to the queue of pages *link_queue* that the crawler will crawl in the future, if the link has not been already crawled. On line 9, the *add_relation()* method is called with the name of the current book and the name of each the recommended books as parameters.

Finally, on line 11, *crawl()* gets called recursively, and uses the *scrapy.request()* method to create a new HTML request.

Algorithm 5: crawl(response)

```
1  current_book_name := response.path('*[@id="productTitle"]').get_text()
2  format := response.path('div[@id="tmmSwatches"]')
3  if format is book then
4      carousel_books := response.css('li.a-carousel-card > div:first-child >
        a:first-child').extract(6)

5      for each book in carousel_books do
6          if book.link not in visited_book_links then
7              visited_book_links.append(book.link)
8              link_queue.append(book.link)
9          add_relation(current_book_name, book.name)
10     end
11 crawl(scrapy.request(link_queue.pop()))
```

The *add_relation()* algorithm (algorithm 6) adds a relation between two books to the database. The first 4 lines checks if the new relation either is a relation from a new book that has not been entered in the database, or to a new book, or both. If they are new books, the method *add_book()* method adds it to the database.

On line 5, a line is written to the end of the *relations* file. The line is composed from the index of *book_a* and *book_b* in *book_list*, the list of all known books. The two indexes are separated by a comma.

Algorithm 6: add_relation(book_a, book_b)

```
1  if book_a not in book_list then
2      add_book(book_a)
3  if book_b not in book_list then
4      add_book(book_b)
5  relations.write(book_list.index_of(book_a) + ', ' +
    book_list.index_of(book_b))
```

Finally, the *add_book()* method (algorithm 7) adds a new book to the database and to *book_list*. This is done by first writing the length of *book_list* to act as an ID for the data element, followed by a comma, and then the name of the book, to the end of the *items* file. After adding the book to the file, the name of the book is appended to *book_list*.

Algorithm 7: add_book(book)

```
1  items.write(book_list.length + ', ' + book)
2  book_list.append(book)
```

Once these algorithms are done, the resulting *items* and *relations* files are comma-separated files that contain the data required to work the database.

The *items* file contains the IDs of books in the system and their names, while the *relations* file contains pairs of from/to relations using the ID values from *items*.

Appendix B

Algorithm Efficiency Analysis

As a part of determining how well we have solved our problem statement, an essential part is efficiency, and the best way to describe the efficiency of an algorithm is to decide it is time complexity.

Weight Calculation Complexity

The time complexity, of the **Weight** algorithm, which is specified in algorithm 1 is mainly affected by the three loops in it. Firstly, the main *while-loop* will keep looping as long as the current element should be updated. Secondly the two inner *foreach-loops* is mainly responsible for the remaining time. The time complexity, is calculated and reduced to $\mathcal{O}(n * m * k)$

Where n is the number of cache elements that should be updated. If the state of the warehouse does not change, this is directly proportional to the amount of times this algorithm is executed, meaning that the more times the algorithm gets executed, the lower n's value will be. Since the results from previous calculations are saved.

m is the number of frontiers.

And k is the number of neighbours, contained by the last node of the current frontier's route.

By the call to **NotExplored** on line 8-12 in the figure, note that the call is not written explicitly in the pseudo code, the complexity should be multiplied with $m * j$ where j is average length of each frontier path.

This concludes the time complexity to be $\mathcal{O}(n * m * k * m * j)$ Which can be reduced to $\mathcal{O}(n * j * k * m^2)$

Book Addition Complexity

The algorithm *add_ book* algorithm, which can be seen in algorithm 4, has a time complexity of $\mathcal{O}(n)$. The only factor worth mentioning that contributes to the time complexity is the *foreach-loop* that counts the number of that book already on the shelf.

Book Placement Complexity

The algorithm *BookPlacement* algorithm, which can be seen in algorithm 2. Firstly the algorithm has the main *foreach-loop* which contributes with n iterations, furthermore the algorithm also has the **UpdatePriorities** call within the foreach-loop, which together makes up for the first part of the time complexity, namely the $\mathcal{O}(n*m)$. Afterwards, still within the main *foreach-loop*, we sort the set of items. This method is the iEnumerable.OrderBy, which has the time complexity of quick-sort, namely the average case of $\mathcal{O}(n * log(n))$ time, and the worst case of $\mathcal{O}(n^2)$ time.

Added up together this algorithm has a average case time complexity of $\mathcal{O}(n * m + n * log(n))$ And a worst case complexity of $\mathcal{O}(n * m + n^2)$, which however can be reduced to $\mathcal{O}(n^2)$.

Where n is the number of books that are to be placed, and m is the number of outgoing relations from each book to be placed.

Importance Complexity

One of the smaller algortihms we have made, is the importance coefficient calculation. The only factor that affects the time complexity is the *foreach* loop that iterates over the input item set. So this algorithm has a time complexity of . Where n is the number of items that has to be calculated upon.

DistanceFromNode Complexity

The algorithm is previously described in Listing 9.2 and can be described in terms of tme complexity as follows:

The first factor of the algorithm that plays a role, in the consideration of time complexity is the foreach-loop (line 38 in the figure), this iterates over all nodes in the graph, and can therefor be described with $\mathcal{O}(n)$, where n is the number of nodes in the graph.

By line 19 of the figure, which is a while loop, the time complexity is increased to $\mathcal{O}(2n)$.

At line 96 of the figure, we have a foreach-loop, which iterates over the edges of each node. This further increases the time to $\mathcal{O}(n + n * m)$, where m is the average amount of edges, per node in the graph.

Lastly, we have another loop which iterates over the amount of nodes once again. This concludes the time complexity of **DistanceFromNode** to be $\mathcal{O}(n+n*m+n^2)$.

This can however be reduced to $\mathcal{O}(n * m + n^2)$. In every case where $n >= m$ this can be expressed as $\mathcal{O}(n^2)$

LoadAllDistances Algorithm

This is a relative short algorithm, and its time complexity is determined to be $\mathcal{O}(n^2)$, where n is the number of nodes in the input list. The time complexity is only affected by the two foreach-loops, where one of them is nested, and both iterates over all nodes.

CalculateDistance Complexity

This algorithm calculates the distance between two arbitrary nodes. Its time complexity is calcualted as follows:

Due to the while-loop and the inner foreach-loop we have a time complexity of $\mathcal{O}(n * m)$, where n is the number of nodes, and m is the number of edges per node. However, within the inner foreach-loop, we have a $List < Node > .Contains$ call, which has a time complexity of $\mathcal{O}(m)$, so combined with the given context, we will reach a final time complexity of $\mathcal{O}(n * m^2)$

Appendix C

Weight

In this section, the code for the main weight algorithm is explained in detail.

The main weight algorithm takes a set of items, and a graph of nodes, and then returns the weight of the set of items on the graph, or in other words, the distace required to travel to pick up the set of requested items. The implementation of the weight algorithm is shown in Listing C.1.

```
public static float Weight(Node[] graph, Item[] itemSet, out Node[] path, WeightCache cache = _cache, DistanceMap distanceMap = _map)
{
    List<Frontier> frontiers = new List<Frontier>();
    Node dropoff = graph[0];
    frontiers.Add(new Frontier(new[] { dropoff }, itemSet, 0));

    while (cache[itemSet].Marked)
    {
        Frontier resultingFrontier = new Frontier(null, null, float.MaxValue);
        foreach (Frontier frontier in frontiers)
        {
            Node lastNode = frontier.route.Last();

            foreach (Shelf neighbour in lastNode.Neighbours.Where(n => n is Shelf).Cast<Shelf>().Where(s => s.Contains(frontier.books)))
            {
                float distance = Distance(distanceMap, lastNode, neighbour);
                if (NotExplored(frontiers.ToArray(), frontier.route.Append(neighbour).ToArray()) && frontier.weight + distance < resultingFrontier.weight)
                {
                     resultingFrontier = new Frontier(frontier.route.Append(neighbour).ToArray(),
                           frontier.books.Where(i =>
                           !neighbour.Contains(i)).ToArray(),
                           frontier.weight + distance);
20                  }
21              }
22
23              CacheElement c;
24              if
                    (cache.TryGet(itemSet.Except(frontier.books).ToArray(),
                    out c) && c.Marked)
25              {
26                  float distance = Distance(distanceMap, lastNode,
                        dropoff);
27                  if (frontier.weight + distance <=
                        resultingFrontier.weight)
28                  {
29                      resultingFrontier = new
                            Frontier(frontier.route.Append(dropoff).ToArray(),
                            frontier.books, frontier.weight + distance);
30                  }
31              }
32          }
33          frontiers.Add(resultingFrontier);
34          if (resultingFrontier.route.Last() == dropoff)
35          {
36              CacheElement c =
                    cache[itemSet.Except(resultingFrontier.books).ToArray()];
37              c.Marked = false;
38              c.Weight = resultingFrontier.weight;
39              c.Path = resultingFrontier.route;
40          }
41      }
42
43      path = cache[itemSet].Path;
44      return cache[itemSet].Weight;
45  }
```

Listing C.1: Calculates and returns the weight of a set, updating a cache of elements underway.

The parameters of the weight method is firstly a set of nodes. These nodes are expected to be a all-pairs shortest path graph of shelves and the dropoff point only. Furthermore, the set of items `itemSet` is the set of items that the evaluation of weight need to be calculated on. An output parameter `path` gives the path of the lowest weight calculation as output if needed, and a cache of weights are expected to already be initialized, as well as a distacemap that describes the distance between each pair of nodes. If these are not initialized where the weight method is called, standard versions saved in fields are used instead.

Firstly, on line 3, a list of frontiers are initialized. Each frontier has as fields `route`, an array of nodes that describe the route that the frontier has taken, `books`, the remaining books that the frontier need to have picked up before it can return to the dropoff point, and `weight`, the weight of the route saved in the frontier.

On line 4, the dropoff point is saved as the first node in `graph`. Then the first frontier is added, a new frontier where the route is initialized as the dropoff point only, the full requested itemset as the remining books, and a weight of 0.

The while-loop on line 7-41 runs while the requested set of books are not marked in the cache. If the set of books are marked, it means the weight saved on the cache location is outdated and needs to be updated before reading. If it is not marked, then the weight is accurate, and can be read fromt he cache and returned directly. Otherwise, the while loop runs until the set is marked.

On line 9, inside the while-loop, `resultingFrontier` is initialized. `resultingFrontier` is the most efficient frontier in this iteration of the while loop, and is the new frontier that needs to be added to the list of all frontiers before the iteration of the while loop is finished. It is initialized to have `route` be `null`, `books` be `null` and `weight` to be the max value of float.

Then, on line 10-32 a foreach-loop iterates through each frontier in the list of all frontiers. In the loop, `lastNode` is set to be the last node in the current frontier's route, and on line 14-21, the neighbours of the last node is iterated through, but only the neighbour nodes that are shelves and contain at least one book in the books of the frontier.

In the inner foreach-loop, the distance from `lastNode` to the neighbouring node is found based on the distance map, and the potential new route that would be added if a new frontier is added is checked for weather or not it has already been explored before. If it has not, and the weight of the potential new frontier is lower than the `resultingFrontier` that is currently saved, the resulting frontier is set to this new frontier. The new frontier is initialized on line 19, where the route is the current frontier's route with the node appended, the books are the current frontier's books minus the books contained on the neighbour, and the weight is the current frontier's weight pmus the distance from the last node to the neighbour.

On lines 23-31, it is checked wether the current frontier containes a set of books whose cache element need to be updated. If this is the case, the new frontier might be a frontier that returns from collecting books, and is traveling back to the dropoff point. If the most efficient new frontier is the frontier that returns from collecting books, then the resulting frontier is set to be a frontier that has the dropoof point appended to the current frontier's route as route, the cuurent frontier's books as books, and the current frontier's weight plus the distance form the last node to the dropoff point as weight.

On line 33, the resulting frontier is added to the set of frontiers. Afterwards, the last node of the resulting frontier is checked to be the dropoff point. If this is the case, the resulting frontier represents a cycle that collects a set of books, and the cache entry of the set of books need to be updated. In the if statement on lines 34-40, the cache element is updated to not be barked, and the weight is updated to be the weight of the resultingfrontier. Furthermore, path is set to be the resultingfrontier's path.

On line 43 and 44, the cache element data entry for `itemSet` is accurate, so `path` is set to the cache element's path field, and the weight of the cache element can safely be returned.