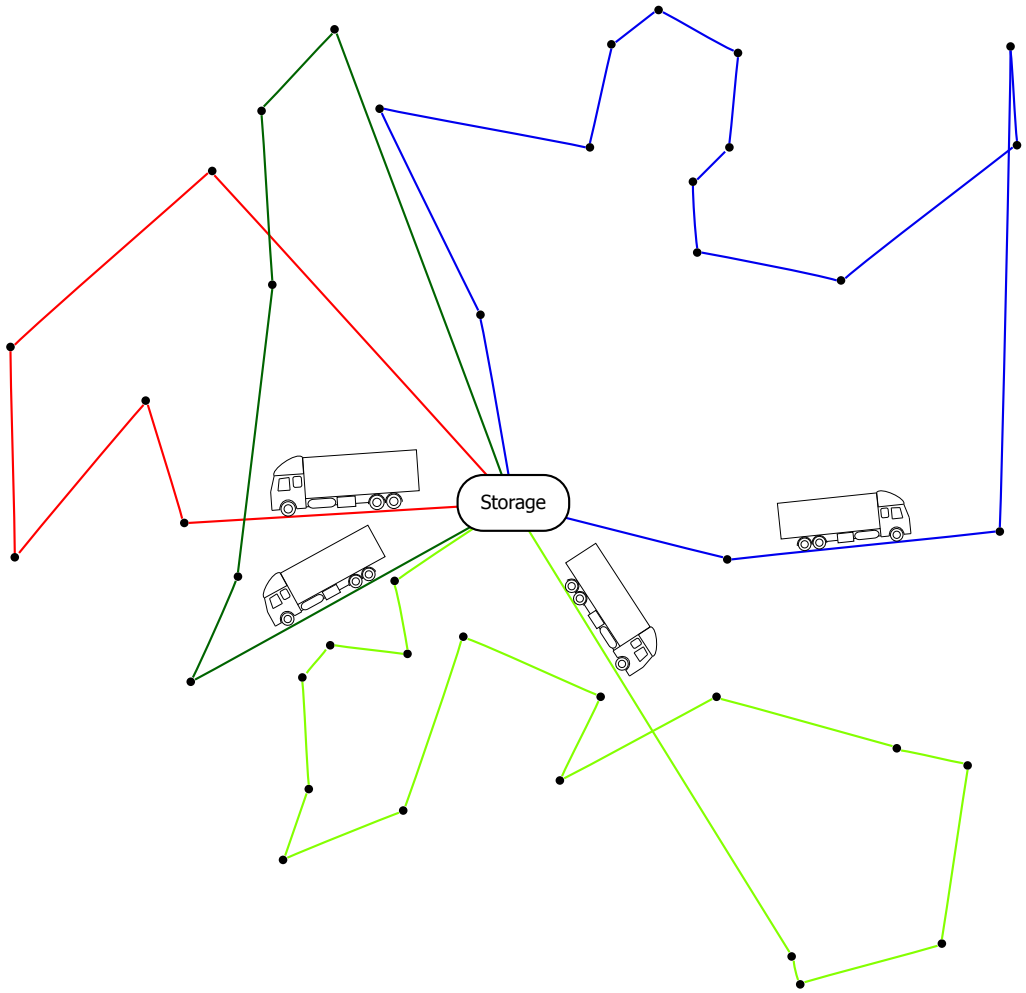
Multiple Routes Planning

P2 project, Aalborg University
Computer Science
TEK-NAT Basis



Group C218
Rune Leth Wejdling
Nicholas Tinggaard
Torben Løkke Leth
Kristian Jensen
Niels Husted
Mikael Møller
Jakob Knudsen

AALBORG UNIVERSITET

Det Teknisk-Naturvidenskabelige Basisår
Datalogi
Strandvejen 12-14
Telefon 96 35 97 31
Fax 98 13 63 93
http://tnb.aau.dk

Titel:

Multiple Routes Planning

Tema:

"Modellernes virkelighed", undertema "Netværk og algorithmer"

Projektperiode:
P2, forårssemestret 2006

Projektgruppe:
C218

Deltagere:
Jakob Knudsen
Torben Løkke Leth
Rune Leth Wejdling
Niels Husted
Kristian Riishøj Jensen
Mikael Harkjær Møller
Nicholas Tinggaard

Vejledere:
Morten Kühnrich
Claus Monrad Spliid

Oplagstal: 14 stk

Sidetal: 85

Appendiks: 3 kapitler

Bilags antal og -art: 1 stk CD-rom

Afsluttet den: 29/05 2006

Synopsis:

Based on an interview with L.C.H. Import A/S, we have identified a route planning problem, which we regard as a Travelling Salesman Problem. The problem is represented using graph theory. In an attempt to help solve this problem, we have developed a program design which uses Christofides algorithm while creating routes in a complete graph.
The program design consists among other things of a demand specification based on the identified problems, with demands which needs to be kept for the program design to be usable. The program design has been implemented in a prototype in ANSI C, which has been evaluated in regard to the demand specification. Our test results show that the prototype produces routes, less than 1.5 times longer than the optimal route, within an acceptable timespan. We can conclude that a software based solution can ease a route planning process.

Preface

We are a group of Computer Science students at Aalborg University, and wrote this report as part of a 2^{nd} semester project. We worked on this report during the spring semester of 2006 in the period 2^{nd} of February to the 29^{th} of May. During this project, courses in "Samarbejde, Læring og Projektmanagement", "Diskret Matematik", "Natur, Menneske og Samfund", "Fagets metoder og formidling" and "Programmering i C" were followed and one of our main goals with the project, were to use what we learned in these courses.

We have chosen to work with a project draft named "Find rundt i CW-Obels komplekset". This draft introduced problems in finding a route, between two locations without violating predefined rules. The draft also encouraged us to extend the location of the way finding, therefor we transformed the primary problem into a problem concerning route planning and route optimization. We made this transformation because we found the route planning problem more interesting and complex. We have in our research of the problems, cooperated with a company called L.C.H Import, who had some logistic problems like the ones we planned to work with.

We want to give thanks to Per Bergmann from L.C.H. Import A/S for his co-operation throughout the project and for giving access to internal documentation and his time when needed. We would also like to thank Jens Jørgensen from Daka Bio-industries, for giving us insight in the implementation process of a software based solution in a logistics operation.

We have chosen to write this report in english opposed to danish, to make us more capable of handling demands that can arise later in our studies, as we might get non-danish supervisors or we choose to study abroad. We believe we have more room to experiment now during the earlier projects, than we will have later on in the education.

Rune Leth Wejdling
runelw@tnb.aau.dk

Simon Nicholas Moesby Tinggaard
nicholas@tnb.aau.dk

Torben Løkke Leth
sshadows@sshadows.dk

Kristian Riishøj Jensen
krj1985@tnb.aau.dk

Niels Husted
nielsh@tnb.aau.dk

Mikael Harkjær Møller
mikaelhm@tnb.aau.dk

Jakob Knudsen
jakobsk@tnb.aau.dk

Reading instructions

Throughout the entire report each chapter will start with at short introduction. Each introduction is constructed such that it gives the reader a quick overview of a chapter, with a short description of the chapter contents. It should then be possible to read the introduction and chapter conclusion, to get a good idea of what the purpose and results of the chapter is. We have used different types of references in this report; the "[0]" represents source references to the bibliography. References to figures and sections are written in the text as normal numbers. For instance, a reference to the problem analysis could look like this: see section 2.1 on page 2.

CONTENTS

CHAPTER

1

Introduction

Solving logistical problems can often be a complicated and expensive task for a company. Managing many trucks and routes for a lot of customers, while ensuring the truck drivers still comply with regulations in respect of driving hours and rest hours, can often be a nearly insurmountable problem for one or even several employees.

In the duration of this project we have made contact with a company that has logistical organizing problems, which currently are solved manually. The problems include, among other things, the time it takes to plan routes for a set of trucks and finding a good way of utilizing these trucks. We believe it is possible to find a solution, so these problems could be solved and optimized, or at least a solution that can reduce the workload of the logistics employees.

As a way of solving the problems in our problem analysis, we aim to work with C programming and implement a working prototype of a program in ANSI C, which can solve the problems. The criteria of the solution is, here at project start-up, that the solution will produce a satisfactory output, to a correct given input. To determine if the output is satisfying, it will be necessary to setup a set of criteria we can test against, based on information found during the problem analysis and theory sections of the report. To ease the programming phase, we want to find or create a development method that fits our project group and work model. Another goal with this project is to work with and understand graph theory, including Minimum Spanning Trees (MST) and the Travelling Salesman Problem (TSP). Another aspect we need to consider, is if we can solve our problem with a software solution. When solving a TSP, we might not be able to find an optimal solution to this problem within a reasonable amount of time.

1.1 Travelling Salesman Problem

The Travelling Salesman Problem, or TSP for short, is a concept named after what it is originally all about. Before the concept was formally known as a TSP, the problem was formulated by a travelling salesman, who wanted the shortest way through the cities he was supposed to visit and then back home, while only visiting each place once. Later in section 4.3 on page 22 we will go into details about TSP, but at present time it is enough to know that a TSP essentially deals with finding a path trough a series of points and returning to the point of origin, while travelling the shortest distance.

CHAPTER

2

Project phase overview

The aim of this project is solving a specific problem using known techniques from computer science. This section describes the various processes and phases of the project needed to come to the conclusions in chapter 7.

The course of this problem based project is split into a number of phases, as seen on the chart in figure 2.1 on the facing page. Each phase has a distinct purpose and planned outcome, which is described in the following sections. Each box in the chart represents a specific task or minor phase and the enclosing boxes represents major phases of the project. The arrows represent dependencies. An arrow from one minor phase to another specifies that significant parts of the second phase were based on the first one.

2.1 The problem analysis

The problem analysis is the process of explaining how travelling salesman problems are identified in a company, identifying the relevant problems in a case, discussing how these problems occur and what impact they have on the company. We have the hypothesis that logistic operations in a company can be perceived as a travelling salesman problem. On that background we find it necessary to investigate the field of logistics and identify what processes can be perceived as travelling salesman problems. After having illustrated the role of travelling salesman problems in a company, we will perform an interview with a company and base a case on it, which serves as an example of the travelling salesman problem in a company. The interview will not only serve as documention of the existence of an actual travelling salesman problem in a company, but will also serve as means to identifying specific problems that can be addressed by a software based solution. The interview is directed towards the company's logistics employee and should identify relevant problems and enable us to plan associated data types for use in a later development phase. The information we gain from the interview with the company, should provide us with multiple problems to solve and provide a base consisting of problems, that actually exist in a company. We expect that the amount of problems we find will be somewhat large, meaning we might have to delimit these to a manageable size suiting the project. The problems we choose to work with should enable us to sketch a rough outline, of the software we wish to develop, with an overview of which information we need to acquire to solve those problems. This rough outline will introduce a number of practical questions, concerning how we actually build it. These questions need answers before the development can proceed. Though we intend to solve the problems posed in the problem analysis with a software based solution, we find it relevant to first document that it is a plausible solution to the problem before actually trying to solve it. The background for making a software based solution, is drawn from a third party interview, with a company that has already implemented a solution to problems similar to those we found earlier in the problem analysis. This interview will also serve as a way of examining which effects an actual implementation of a software based system could have in a company, which then later can be used to form more qualliyfied predictions or guesses of what effects our solution could have in a company.

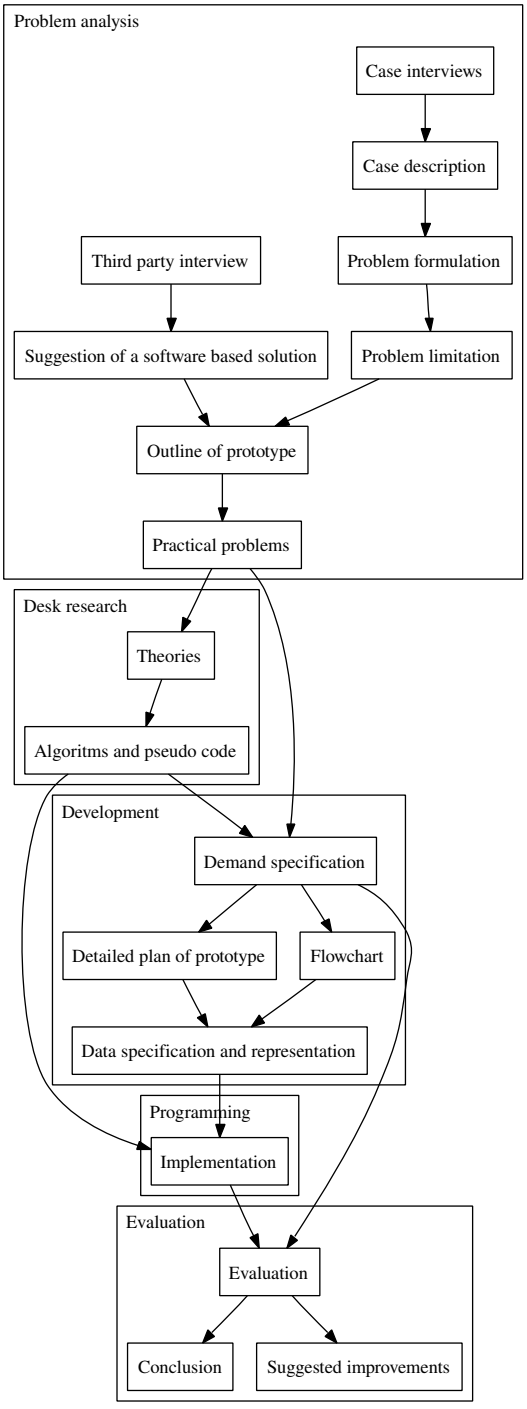


Figure 2.1: Outline of the different phases of this project.

2.2 The desk research phase

The desk research phase, is the phase where we gather and gain generic knowledge about travelling salesman problems and how to effectively solve them. Before we investigate the travelling salesman problem area, we need to have some mathematics and graph theory correctly defined, since it will be necessary to understand the algorithms, which can be used to solve a travelling salesman problem. Through desk research, we seek knowledge and understanding of the algorithms and theories needed to provide answers to the practical questions. The information we gather must be presented in a precise form, to make it as usable. At the same time definitions are made which provide the building blocks for our prototype, so we have a definite toolbox to use when designing the solution.

2.3 The development and planning phase

The development or planning phase is the process of applying the gained knowledge, about theories and methods found during the desk research phase, on the plan for our program. Here we design and construct our software specification from the algorithms, theories and demands found throughout the previous research. The detailed plan of the prototype is where the functions and data flow of the system is described. The flowchart is not a detailed specification of the prototype, but rather an overview of the structure of the software based solution. The flowchart and the detailed plan of the prototype is the basis for our definition of the data specification and representation. Before we actually start programming the procedures described, we will examine different programming methods and from these define the method we will use, when programming our prototype.

2.4 The programming phase

Through a programming phase the prototype is programmed in ANSI C, with the purpose of creating a testable prototype. The reason we want an actual prototype is to be able to practically test it and see if it produces the planned output, from the defined input and to get an output we can evaluate according to the demands identified previously. The purpose of this phase is also to gain programming experience in ANSI C, as it is one of the technical goals of the project.

2.5 The evaluation phase

During the evaluation phase the programmed prototype is tested according to a series of demands, based on the problems found in the problem formulation. According to these demands the effectiveness of the prototype is evaluated and discussed. As we plan to code a simple but working prototype, there will be room for improvements, which will lead to the consideration of steps which could be made in order to increase the efficiency of the software solution. The considerations will be made both from results gained from the actual test but also from what we perceive as possible improvements in a future version of the program.

CHAPTER

3

Problem analysis

The following chapter deals with the problem analysis, in which we identify the problems we want to solve. First general logistics is described in section 3.1. Here several problems concerning logistical operations, which can be identified as being travelling salesman problems, are described. Later we identify problems in an specific company, which at the final steps of the problem analysis results in a problem formulation. In the problem formulation, in section 3.3 on page 12, we define the problems to be solved. These problems require, that a number of practical problems are solved.

3.1 Logistics

Logistics is a widely known concept which, depending on whom you ask, has different definitions. Most people have an idea of what logistics is about, however the subject is usually more complex than first perceived. Logistics is not just the process of delivering goods to the customer, but a wide field of different processes that helps organize and optimize an entire production process, from the planning and design of the product, to the actual delivery and sale of the finished product. As a relatively young field of business science, research is done in the field to improve the current logistic methods and finding new methods. As such there is no definite definition on logistics; however there are several accepted definitions which describe the subject. The following section is written from the definitions and information given in *"Indkøbs- og Materiale Styring (logistik)"* [2] and *"En mosaik af dansk logistikforskning"* [11].

"The definition of logistics is the process of planning, implementing, and controlling the efficient, cost-effective flow and storage of raw materials, in-process inventory, finished goods, and related information from point of origin to consumption for the purpose of conforming to customer requirements." - En mosaik af dansk logistikforskning [11] (Translated)

The above quote is a traditional definition of logistics. Although logistics is a young field of business science, the concept of logistics has been known for a long time. It was originally used within military organizations to supply troops with the supplies they needed and the right amount when they needed it. During the past 50 years logistics has shifted focus from being mostly used by military organizations to being used extensively by business organizations and it has become an important factor for the success of any company. The increasing globalization and demands for swift and flexible delivery of goods, has made it essential for a company to be able to process orders and deliver goods as efficiently as possible.

In a business, logistics has the function where you from a higher level systematically control the flow of materials and production processes in manufacturing and distributing goods. The overall goal of this plan is: delivering the goods on time, raising the quality of the goods, removing errors in delivery and product quality and removing unnecessary storage facilities, securing a good information flow about the goods and a good cooperation on all levels of the production and distribution processes.

Apparently there are several interesting subjects of logistics, which can be seen as Travelling Salesman Problems. One problem to look at, is the production process and finding a way of optimizing the production on a factory line or the production of a specific product. Another problem is optimizing the efficiency of how the finished goods are delivered to the customer; the ordering process involved and the information flow linked to this process. The latter is the subject we wish to work with during this project, which is depicted as the the boxes on a gray background in figure 3.1. We find this part of the process has the most interesting problems to work with, when considering the various optimizable processes of a production process. The reason we chose to work with the delivery process, is that it is the process that is most often the subject to change in a company. One can easily imagine that a company has to deliver goods to different customers on various days and that these routes can be optimized. It is harder to imagine a company who often has TSPs involved in the design process, as the design for a product rarely is remade once a satisfactory solution is found. The same applies to the assembly process of an assembly line, for instance - once a satisfactory assembly process has been found, it is rarely re-done as it takes time to reposition machines, upgrade equipment and find a new position for these items within a factory hall. Based on these observations it seems that solving problems based on the TSP principle in the delivery process is most suitable for our project.

The part of this process we will work with during this project, is the process of handling the finished product, and distributing it from the inventory of the company to the doorstep of its customers. This problem is what can be defined as delivery service, but can again be divided into a series of subproblems, which each have an impact on the efficiency of the process: delivery time, inventory service, delivery flexibility and delivery information. Each of these phases are described in depth in our sources about logistics ([2] and [11]).

Delivery time can be described as the time it takes from when the company receives an order in the sales department to when the ordered goods are delivered at the customer. The time it takes from the order is made till the delivery is done, depends on two time dependent elements: the administrative time elements and the physical time elements. The administrative elements include but are not limited to when the technical order processing and the order processing time. The technical order processing is a process which is run when a customer needs to have a specific product designed. Order processing time is the time it takes to actually treat the order through the company from contract agreement to the purchase of the goods required to fill the order. The physical time elements include but are not limited to; the delivery of raw materials to be processed, the time it takes to process these into a finished product, the time it takes to package the products for delivery to the customers and the actual delivery to the customer by transportation.

When a company delivers goods to a series of customers, there are a typical chain of events until the goods are in the hands of the customer. Depending on the type of business, the customers will usually call in an order which is then processed through the accountancy.

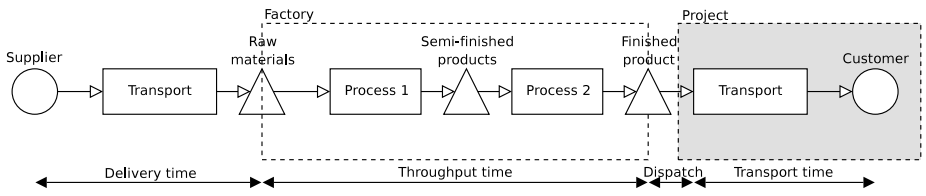


Figure 3.1: The general phases of a production process, organized as a logistics operation.

Once this process has come to its end, the order is then sent to the relevant department. The order is then sent to the inventory, where the order is processed and the goods are packaged and made ready for distribution to the customers. The distribution itself can be done in many different ways and by many different means. However a common pattern is that once an order is ready, it has to reach the customer within a specific timespan, requested by the customer. It is this process we wish to work with in this project.

3.1.1 Optimizing the delivery process

When the objective is to optimize the delivery process, from an order is submitted to the goods are delivered, there are several things you have to take into consideration. Firstly it is important to know which information an order holds; typically an order will include the type and amount of goods that needs to be delivered, and in which span of time it has to be delivered. Secondly it is important to know which tools are available to deliver the order on time. A company will usually possess the means of delivering goods; using either their own vehicles or contracting an outside company. When seeking to optimize the processes it is important to consider the following subjects and their relationship to each other: order processing, order packaging and the time it takes to make the delivery.

Common for each of the before mentioned aspects is they are time dependent. If you wish to optimize the delivery process, you need to optimize the time it takes the goods to get from the customer to the company. Typically a company will try to optimize the use of its delivery vehicles, by allowing it to carry goods for more customers at a time and delivering the goods in order within each customers requested delivery timespan. The plans for delivering the goods are typically pre-laid by a logistics employee, who uses information from the submitted orders to create a number of delivery routes.

3.1.2 Logistics conclusion

Through the logistics section we have found that logistics is a large subject which can be split into several different areas. We have chosen to focus on the delivery process where the overall aim is utilizing the available resources for transportation of goods as efficiently as possible. This optimization process deals with using the trucks available to distribute as much goods as possible, within as short a span of time as possible. This must be done without breaking the regulations for driving hours. This means that the problem, to be solved, deals with making a series of efficient routes, while ensuring that the customers get the goods they have ordered within a set deadline. The characteristics of the logistical problems, defined in this section, resemble the TSP defined in section 1.1. This and the nature of TSP described by Gerhard Reinelt [4], leads us to believe that the logistical problems involved in the delivery process, can be identified as a TSP.

3.2 Case interview

To identify a number of problems to treat during this project, we established contact to a local company. This forms the basis of the case scenario. In the following section this case is described. It is based upon an interview we have made with a local company called L.C.H. Import A/S (referred to as L.C.H. from here on). One of our group members is an employee at L.C.H., and through him we took initial contact to the company. L.C.H. is an importer and distributor of fireworks and has roughly 1000 customers spread throughout Denmark.

3.2.1 Interview method

As a method of identifying a number of problems of the nature described in section 3.1.2, we have interviewed a local company to identify problems to be solved. After having evaluated the various information gathering methods found in the article by Togeby, Mehlbye and Rieper [6], we decided to gather the information by performing a one-man interview constructed as a conversation. In this case we find it the best way to gather the relevant information we need to identify the problems. One-man interviews are characterized by giving very precise information about the problems, but it requires that the interviewer is good at paying attention, and is able to gain the trust and confidence of the interviewed person. It is essential for the project that we get information we can use in a later analysis of the problems. The advantage of a one-man interview is that, you have the opportunity to get very precise information about even the most complex problems, from the right people. Using this method it is also possible to get the interviewed persons view of the problem. However this can also be a disadvantage since the information can be biased by the views of the person in question.

For the interview itself, we created a questionnaire guide with questions regarding the information we initially wanted to gather from the interviewed person. The questionnaire was based on the inside knowledge we had about the problem, though the questions were still loosely formulated, as we wanted to avoid guiding the conversation with our own perception of the nature of the problems. The loosely formed questions allows for a more in-depth conversation during the interview which could open up for new information we might not previous have been aware of.

We interviewed the logistics supervisor Per Bergmann of L.C.H. Import A/S. The questionnaire, transcription and recording of the full interview can be found in the interview directory on the CD supplied with the report. Notice that the recording of the interview is missing the first 3 minutes, this is due to a minor technical problem at the time of recording.

3.2.2 Case description

On the 3rd of November 2004, one of the biggest firework disasters in danish history took place. A large fireworks factory exploded and destroyed a large area in the small village of Seest, near Kolding. One fire-fighter did not survive the rescue operation as described in this police report [3]. After the disaster in Seest, the law [5] about how to handle, store and sell fireworks has been tightened. The new law has made it more difficult for L.C.H. to sell and distribute their firework products. It limits the period in which they are allowed to sell and distribute fireworks to their customers. The customers are also not allowed to have as large quantities of fireworks in their storages, as they were before the disaster. Therefore it is important for L.C.H., that their logistics system is well functioning. Otherwise they can not live up to the delivery demands from their customers.

Previous years L.C.H. have been relying on a freight company, to transport the fireworks, but the freight company has canceled their contract with L.C.H. because it was not profitable to transport the fireworks after the new laws were enforced. The law states that they are no longer permitted to transport other cargo in a truck carrying fireworks, which makes it a difficult task for a freight company. This forces L.C.H. to hire all the trucks they need by themselves. It is very expensive for them to hire a truck with driver; it costs about 6000 kroner per day, and they can only use a truck for roughly 10 hours a day. At the peak of their season they use around 120 trucks, which means it is important that they use the trucks as efficiently as possible. This is a new situation for them, since last year the freight company

took care of almost all the logistics involved in transporting the fireworks. The only thing L.C.H. was responsible for, was packing the fireworks, and determining which orders were most critical and had to be delivered first. Now L.C.H. is responsible for all steps involved in the process of transporting fireworks. This is a very demanding task for the employees, who have not handled logistics in such a large scale before.

"We have some deadlines which are critical and it is very important the fireworks leave from here in time. It means that if you want to make the deadline, you need to calculate backwards." ... "The fireworks needs to be packed, typically in the storage throughout the day, evening and night and then they are loaded on to x number of trucks, which drive around the country or in a non-definable area."- Per Bergmann, L.C.H. Import, from the interview (Translated)

L.C.H. have three very large customers; Jem & Fix, Harald Nyborg and Aldi. During the season, these large customers need a constant delivery of fireworks to all their shops around the country. Sometimes they even need deliveries up to three times a day. Trucks from L.C.H. drives out to the stores and ask what they need, so they do not need to order, just wait for a truck to come by. Every time a truck runs low on one kind of fireworks, it needs to go back and refill. It is hard to predict where on the route the truck needs a refill, but it will save a lot of time and money, if it were possible to have a forecast and plan routes by this. It is very expensive to drive all the way home for a refill. L.C.H. uses a lot of resources on planning the routes for these three customers, and it is possible that the routes can be even better.

All their smaller customers order fireworks by phone. Last year the customers called their orders in by phone a day or two before they needed the fireworks delivered. The amount of deliveries could sometimes reach 500 orders a day. When L.C.H. received an order, the order was sent to the accountancy, where it was enqueued in a pipeline after processing. The pipeline contained all orders which had been received and were currently being processed inside the company. When an order physically left the company storage, it was also removed from the pipeline. The pipeline was accessible from the storage, where the inventory clerks were responsible for packing the orders. After the fireworks had been packed the status of the order in the pipeline was updated. While the order was being packed a logistics employee determined which orders, were the most critical and needed to be dispatched first. Last year L.C.H. had to have a deadline for next day's orders at 11.00, otherwise they could not keep the deadlines for delivery.

In the coming season, L.C.H. will be forced to handle all the logistic problems by themselves, after the cancellation of the contract with the freight company. Therefore they have planned a new procedure for the order flow. Customers will still order fireworks by telephone. The orders passes trough the accounts department, and into the pipeline as they did last year. The main difference, this year, is how the orders in the pipeline are supposed to be handled. A new task for L.C.H. is to create routes for the trucks, in a cost-efficient way. They have planned to have an order deadline at 16:00. Orders ordered before the deadline, will be delivered the following day, except for a few customers located far from the storage. The orders from these customers, are not guaranteed to be delivered the following day, but not later than two days after the order is received.

Since L.C.H. is responsible for ordering all the trucks, they order trucks every day. The number of trucks needed is different every day. The trucks must be ordered before 15.00 which means that the logistics employee needs to make an estimate of how many trucks they need for the next day, before he knows how many orders the customers will place. If he orders more trucks than he needs to use the next day, L.C.H. must pay for all of them even

if they do not use them. Likewise, it is not possible for L.C.H. to get more trucks if the logistics employee has not ordered enough trucks for the day, unless they are very lucky and the freight company has one available. This means it is very important to optimize the usage of the trucks at their disposal.

After the deadline, the route planning takes place. One or more employees try to get an overview of the orders in relation to their location and how much capacity the cargo for these customers uses in a truck. From this information they will try to build routes for the deliveries of the next day. Early next morning, the trucks are packed, and ready for delivering to the stores at opening time.

"It is in itself a fight to phone in an order and then the order needs to be handled by our customer service and added to our economic system. An order slip is sent to the storage, where the fireworks are packed and made ready for transport. Now it is possible to start taking action" - Per Bergmann, L.C.H. Import (Translated)

Currently L.C.H. would like to move the deadline for incoming orders. They want to make it possible for the customers to place orders in their entire opening hours. Since many of their customers are open until 20:00, a deadline at 21:00 would be a good deadline for L.C.H.. Currently they do not have enough time to make the routes, for all of the trucks if the deadline is later than 16:00.

3.2.3 Case problems

The problems we want to solve in this project are based on the case. From our point of view, L.C.H.'s business can be split in two subparts. These subparts have a common primary problem, which then in turn consists of a distinct set of secondary problems.

This primary problem is the planning a number of routes for a number of trucks hired by the company. It takes a lot of time to manually plan routes for these trucks because of the complexity of the problem. There are many variables one needs to consider to ensure the routes are efficient regarding the quality of the delivery and the cost of the overall route. The problems found in this case can, as defined in the logistics section 3.1, be characterized as delivery optimization problems. The problems and subproblems we have found in the case are summarized below.

3.2.4 Part one (Jem & Fix, Harald Nyborg and Aldi)

This subpart of L.C.H.'s business, deals with the delivery of fireworks for their largest customers. Some of these customers are; Jem & Fix, Harald Nyborg and Aldi, which all have a lot of stores throughout the country. Each store needs one or more deliveries a day and needs a limited, yet unspecified, amount of fireworks on each delivery. The routes for these customers are generated before the season starts. Each truck in this subpart has its own route to a specific set of customers, which it repeats the entire day.In this part we have found the following subproblems:

Problem 1: Planning Time.
Initially planning the static routes manually is a complex and time consuming process, as the logistics employee has to maintain an overview of a potentially large amount of trucks and orders at the same time and still produce a satisfactory result. This process is also very expensive for L.C.H., since they currently do not possess any alternative to doing it manually and this requires a lot of man-hours in order to make satisfactory routes. As a further complication, it is difficult to impose minor changes to already laid routes in case additional orders are received after the deadlines, since it this often makes it necessary to generate new routes altogether.

Problem 2: Optimizing truck usage.
Hiring a truck is expensive for the company, meaning it is important to use as few trucks as possible to deliver the cargo and still comply with regulations in respect of driving hours and resting hours. The optimization problem of this subproblem is minimizing the amount of trucks used to deliver the fireworks, while minimizing their time on the road.

Problem 3: Distance to storage.
The distance to the storage for a truck, can be a problem if a truck runs out of a certain type of fireworks while it is on a route. The truck has to be refilled, and the only way of doing this is at the storage. The distance to the storage is a parameter one needs to consider when planning routes for the trucks. The problem is then to determine where on a route a truck is going to need a refill and plan the routes by these forecasts.

3.2.5 Part two (Phone ordering.)

This subpart of L.C.H.'s business contains the minor customers, who phone in their orders by telephone before a set deadline. L.C.H. then orders trucks for the following day to serve these customers and plans routes for them overnight, which is sent to the truck drivers. The planning of these routes is very time consuming and the routes laid out manually are often inefficient, which gives L.C.H. additional expenses in the form of extended delivery periods. In this part we have found the following subproblems:

Problem 4: Planning Time.
At present time the minor customers have to phone in orders before 16.00, but, due to a request from the customers, L.C.H. would like to postpone this deadline until at least 21.00. Because of the complexity of planning, sometimes as much as 500 routes manually, it is a time consuming process and it is, as such, not possible to postpone the deadline any further.

Problem 5: Ordering Trucks.
Every day at 15.00, L.C.H. has to order the number of trucks they need for the following day. If they move the deadline for phoning in orders on fireworks until 21.00, another problem occurs, which is to estimate how many trucks they need for the following day. In effect, if the deadline for phoning in orders for fireworks is postponed to 21.00, L.C.H. will have to estimate how many orders they potentially receive in the period between 15.00 and 21.00, and order trucks accordingly.

Problem 6: Optimizing truck usage.
It is necessary to optimize the usage of the trucks. Since the number of trucks available is estimated, L.C.H. might get in the situation where they have not ordered enough trucks for the following day and will then need to optimize the use of these, to service their customers. This means we have to make sure that a trucks entire capacity is used on a route.

Problem 7: 48-hours deadlines
When planning routes "48-hour deadline" orders are considered. These do not have to be delivered before the day after the 24 hour deadlines. If such a "48-hour deadline" order is not delivered, it will be a "24-hour deadline" order the next day. It would be a benefit to deliver as many of the "48-hour deadline" orders as possible, while they still are "48-hour deadline" orders. It might be possible to use a truck less the next day.

3.2.6 Case conclusion

In the logistics section 3.1, we found that delivery optimization problems could be defined as TSPs. We could likely solve any of the two parts as TSPs, using different algorithms and theories from the field of graph theory. However, before we examine theories and solution suggestions, we have to decide which of the above problems we wish to solve. The two scenarios seem very much alike and as such it would seem obvious to solve them both. However we will only focus on a single scenario to better concentrate on the process and problem solution, instead of spreading our work out far, with the risk of not solving the problems to a satisfactory degree.

3.3 Problem formulation

We choose to find a solution to the problems associated with part two of L.C.H.'s company described in section 3.2.5. Both parts of the case deal with optimizing the delivery process. Despite the fact that the subproblems are very alike, the parts are still different. The route planning, in part one, deals with planning a limited amount of routes to a fixed number of customers, using a fixed amount of trucks. Once the route is planned, the trucks drive their routes a certain period and drop off goods to customers. The routes are then only rarely changed.

We find the second part the most interesting. There are more variables to be considered, when designing a solution. In part two however, it is necessary to generate new routes on a daily basis, as the amount of customers from day to day varies depending on how much sale L.C.H.'s customers have. They might not need a new shipment of fireworks every day and, as such, the routes change from day to day according to the amount of orders phoned in. This number ranges from 100 to 500 orders a day. This means L.C.H. might need different amounts of trucks each day. However we choose not to work with the estimation of the number of trucks needed, because it is based on the experiences of the logistics employee. The trucks are ordered at 15.00, where the last customer orders have not yet been received. This means the amount of trucks ordered in a real scenario is based on forecasts of how many customers usually phone in orders after 15.00, and as such is a variable we cannot account for.

We believe that once a solution is found to the second scenario, it is possible to modify this to also be able to solve the problems posed in scenario one, as the nature of the problems essentially are the same, namely optimizing the delivery process.

3.4 Program draft

In the following section, we will analyze some of the more technical aspects of our problem, and what we need to solve them. Our thesis is that the problem we chose to work with in the problem formulation can be solved using a software based solution, but we can not at present

time know if there is an actual basis of solving them, using that type of solution. Thus we have taken contact with a third party company, where they have had a similar problem which was solved using a "semi-automatic" software solution. The information gathered through this interview is then used to support the decision to make a software based solution for our problems and to gather information concerning the effects the implementation of a software based solution has had in the company.

Once we have established the background for making a software based solution, we will sum up what our solution must be capable of. This rough summary is then used to define in detail which information and knowledge we need in order to design a proper solution to the problem.

3.4.1 Third party interview

We have made contact with Daka Bio-industries, from here on referred to as "Daka", since they currently use a software solution to plan routes for a large amount of trucks. Daka is a company which collects fallen livestock from farmers and has 40 trucks at their disposal to make about 12,000 collections during a weeks time. It is a demanding problem to calculate efficient routes for these trucks, and as a solution to this problem Daka successfully implemented a software system in 1998. Based on their experience we set out to make a software based solution to the problems found through our case.

3.4.2 Daka interview method

On Daka's website [1] we have seen that they use a software solution to solve their logistical problem, which resembles the problems in our case. The interview will be directed at the logistically responsible person, since we need to interview a person, who knows how the logistics functioned before and after the software solution was implemented. We use the phone interview method described in "Håndbog i Evaluering"[6]. Daka company headquarters, where the system is physically implemented, is located in Ringsted and it would require quite some travel time if we had chosen to perform a personal interview with an employee. While a personal interview would be the optimal solution, the phone interview is still a lot better than the alternatives, for indstace an email with a questionnaire form. The phone interview gives us the opportunity to make a more in-depth analysis of their experiences with a software solution, compared to contacting the company by email. We will also have the opportunity to establish trust between the interviewer and the person we interview. The interview will consist of a few yes/no questions, followed by varied questions to clarify the contextual experiences with the software implementation. We interviewed the department chief Jens Jørgensen, from the Randers department. A recorded version of our interview can be found on our appendix cd-rom, in the interview directory.

3.4.3 Daka interview

Daka is using a "semi automatic" software system to handle their route planning and receipt of customer orders. They have an automated phone ordering system, where their customers phone in orders on an automated registration system, and afterwards, a logistics employee transfers all the orders to the planning software, which makes the first edition of a route. It is now the logistics employees job to check the routes for errors and to optimize them.

Before 1992 when Daka implemented the automated ordering system, they carried out all the planning and order receiving by hand. They had offices placed in different regions of Denmark, where the customers should call the office in their region to order a collection

of fallen livestock. After the deadline for ordering was reached, the employees would begin planning next days routes for collecting fallen livestock in their region. Daka used this procedure until 1998, where they decided to centralize the planning and ordering. This workforce reduction was made possible because they bought a software planning system called Roadshow, which only needs a single employee to supervise it, meaning they could close down the small offices.

The route planning procedure is essentially the same as before they got the software, with the difference that their ordering deadline has been moved to midnight, where a planning supervisor arrives at work. The supervisor transfers the orders to the planning software and starts the planning process. During the planning a few errors in the routes might occur, making them less efficient. These optimization problems occur because the system has the country divided into regions. If a collection point is placed close to a border, it could be a better solution to give that collection point to the truck driving in a neighboring region. Daka has had a great deal of success with the system, however the employer would like to fully automate the processes.

3.4.4 Program outline

Based on experiences Daka Industries has had with a software based solution, it seems that a possible solution to the problems presented in section 3.2.5, could be a software solution. The main purpose of such a software solution, would be to generate a set of optimized routes which are human readable. The initial data, used by the solution, would be a; set of customers and their locations, a set of orders, and set of available trucks to deliver the orders. Each order would contain information about the quantity of cargo to be delivered and of course which customer to deliver it to.

Based on this input data the program should generate routes for the trucks, that are to deliver the cargo. The output routes should be readable and ready for use. A valid route must not contain an amount of cargo, which exceeds the capacity of a truck. A truck can only drive one route at a time and a route must not force the driver to violate the regulations for driving and resting hours.

The above is an idea of one way to solve the problems in part two of our case 3.2.5 on page 11, however we need to know how to construct the program needed. We already have a set of input data consisting of a set of orders containing the needed information. Now we need to determine how we can represent these in a software solution and how we can process them to get a usable output.

We need to figure out how to represent the following data; customers and their locations, orders, trucks and routes. The representation of this data depends on the algorithms needed for the before mentioned program, and as such we need to describe these algorithms before we can determine how to represent the data. This is done in chapter 4.

CHAPTER

4

Theory

The purpose of the theory chapter is to describe the theory needed to solve the travelling salesman related problems described in section 3.3.

An algorithm called "Christofides' algorithm", as described in *"The Traveling Salesman: Computational Solutions for TSP Applications"*[4], can be used to solve these. We are planning to implement this algorithm in our product and as such we need to understand it and make pseudo code describing the steps of it. This pseudo code will not only help present the algorithms in an uniform manner, it will also help us later implementing the algorithms in a test prototype.

Christofides algorithm requires some knowledge of graph theory, which leads us to describe this field before moving on to the actual algorithms. Before defining the elements of graph theory using logical expressions, we need to make some mathematical definitions which are used later in the description of the elements of graph theory. Once this is done we define the graph theory which we need.

4.1 General math

Before we can describe the graph theory we use, we need to define a few mathematical expressions for later use in the chapter. Functions like minimum and maximum of a set of elements are going to play an important role when we want to find the shortest path and as such need to be defined.

4.1.1 Maximum and minimum in set

The maximum element in a finite set of numbers E is called $\max(E)$ (see definition 1 for reference).

Definition 1: $\max(E)$ of a finite set $E \subset \mathbb{R}$

$$\max(E) = e \ \leftrightarrow \ e \in E \ \wedge \ \forall n \in E : n \leq e$$

The maximum element, identified by the function $f : E \to \mathbb{R}$, e is the element in a finite set E which results in the greatest value when passed to the function f (see definition 2 for reference).

Definition 2: $\max(E, f)$ of a finite set E and the function $f : E \to R$

$$\max(E, f) = e \ \leftrightarrow \ e \in E \ \wedge \ \forall n \in E : f(n) \leq f(e)$$

The minimum element in a finite set of numbers E is called $\min(E)$ (see definition 3 for reference).

Definition 3: $\min(E)$ of a finite set $E \subset \mathbb{R}$

$$\min(E) = e \ \leftrightarrow \ e \in E \ \wedge \ \forall n \in E : n \geq e$$

The minimum element in a finite set E, identified by the function $f : E \rightarrow R$, is the element in E which returns the smallest value when passed to the function f (see definition 4 for reference).

Definition 4: $\min(E, f)$ of a finite set E and the function $f : E \rightarrow R$

$$\min(E, f) = e \ \leftrightarrow \ e \in E \ \wedge \ \forall n \in E : f(n) \geq f(e)$$

4.1.2 Concatenation of tuples

The concatenation of a series of touples, is a touple consisting of the ordered elements of initial touple followed by the ordered elements of the touple that is to be concatenated (see definition 5 for reference).

Definition 5: The concatenation $A + B$ of two tuples A and B

$$(a, b, c) + (d, e, f) = (a, b, c, d, e, f)$$

4.1.3 Iterative sum of function

If E is a finite set and f is a function $f : E \rightarrow \mathbb{R}$, the sum $\sum_{e \in E} f(e)$ is the sum obtained when passing each element in E to the function f (see definition 6 for reference).

Definition 6: $\sum_{e \in E} f(e)$, where E is a finite set and f is a function $f : E \rightarrow \mathbb{R}$

$$\sum_{e \in E} f(e) = \sum_{i=1}^{n} f(e_i) \ , \ E = \{e_1, e_2, ..., e_{n-1}, e_n\}$$

4.1.4 Multisets

A multiset is a tuple containing a finite set of elements E and a function $q : E \rightarrow \mathbb{N}_0$. $q(e)$ is said to be the quantity of the element e in the multiset. The set of all multisets is called $\mathbb{M}$ (see definition 7 for reference).

Definition 7: The multiset (E, q), where E is a finite set and q is a function $q : E \rightarrow \mathbb{N}_0$

$$\mathbb{M} = \{(E, q) \mid E = \{e_1, e_2, ..., e_{n-1}, e_n\} \ \wedge \ q : E \rightarrow \mathbb{N}_0\}$$

Two multisets are said to be equal when the set of elements in each multiset are the same and the quantity of each element in one multiset is the same as in the other (see definition 8

for reference).

Definition 8: Equal multisets

$$(A,p)\doteq(B,q) \;\leftrightarrow\; A=B \;\wedge\; \forall a\in A : p(a)=q(a)$$

A submultiset of the multisets (E,q) is a multiset whose elements is a subset of E and whose quantity function for each element $e \in E$ is less than or equal to $q(e)$ (see definition 9 for reference).

Definition 9: The submultiset (A,p) of the multiset (B,q)

$$(A,p)\dot{\subset}(B,q) \;\leftrightarrow\; \begin{pmatrix} A\subseteq B \\ \wedge\forall a\in A : p(a)\leq q(a) \\ \wedge A=B \rightarrow \exists a\in A : p(a)<q(a) \end{pmatrix}$$

$$(A,p)\dot{\subseteq}(B,q) \;\leftrightarrow\; \begin{pmatrix} A\subseteq B \\ \wedge\forall a\in A : p(a)\leq q(a) \end{pmatrix}$$

The cardinality of a multiset (E,q) is the sum of the quantities of each element in E (see definition 10 for reference).

Definition 10: The cardinality $|(E,q)|$ of a multiset (E,q) is defined by:

$$|(E,q)| = \sum_{e\in E} q(e)$$

4.2 Graph theory

Finding a solution to a travelling salesman problem can be done using various algorithms, which computes the shortest circuit in a graph. This means we need understanding of graph theory and as such we have to define which parts of it we will use. We need to precisely define the various definitions of graph theory concepts such as graph types, as we will need it later, when writing the pseudo code for the algorithms we need in our program design.

4.2.1 Edges

A directed edge is a tuple consisting of two distinct[1] ordered elements called vertices. A single element in a set of vertices is called a vertex. The order of the vertices in the tuple decides the direction of the edge which is from the first vertex, called the initial vertex, to the second one, called the terminal vertex. The set of directed edges is called $\mathbb{E}_d$ (see definition 11 for reference).

[1] Allowing the initial and terminal vertex to be the same would introduce "looping edges", but they are not needed in the context in which we are going to apply graph theory.

Definition 11: The set of directed edges $\mathbb{E}_d$

$$\mathbb{E}_d = \{(v_i, v_t) \mid v_i \neq v_t\}$$

An undirected edge is a set of exactly two vertices. The set of undirected edges is called $\mathbb{E}_u$ (see definition 12 for reference).

Definition 12: The set of undirected edges $\mathbb{E}_u$

$$\mathbb{E}_u = \{\{v_i, v_t\}\}$$

The degree of a vertex v in a set of edges E is the number of edges in E which contains v. It is denoted $\mathrm{Deg}(v, E)$ (see definition 13 for reference).

Definition 13: The degree $\mathrm{Deg}(v, E)$ of a vertex v in a set of edges E

$$\mathrm{Deg}(v, E) = |\{e | e \in E \land v \in e\}|$$

Two vertices are said to be adjacent in the set of edges E when an edge exists in E which connects the two (see definition 14 for reference).

Definition 14: Adjacency $\mathrm{Adj}(v_1, v_2, E)$ of two vertices v_1 and v_2 in a set of edges E

$$\mathrm{Adj}(v_1, v_2, E) = \mathrm{True} \leftrightarrow \begin{pmatrix} E \in \mathbb{E}_u \rightarrow \big(\{v_1, v_2\} \in E\big) \\ E \in \mathbb{E}_d \rightarrow \begin{pmatrix} (v_1, v_2) \in E \\ \vee (v_2, v_1) \in E \end{pmatrix} \end{pmatrix}$$

4.2.2 Graphs

A graph is a tuple consisting of two sets: A nonempty set of elements V, called vertices, and a set of edges E, where each edge represents a connection between two vertices in V. If the edges of the graph are directed, the graph itself is said to be directed. The set of directed graphs is called $\mathbb{G}_d$ (see definition 15 for reference).

Definition 15: The set of directed graphs $\mathbb{G}_d$

$$\mathbb{G}_d = \left\{ (V, E) \;\middle|\; \begin{array}{l} V \neq \emptyset \\ \land E \subseteq \mathbb{E}_d \\ \land \forall (v_1, v_2) \in E : \{v_1, v_2\} \subset V \end{array} \right\}$$

Likewise, if the edges of the graph are undirected, the graph itself is said to be undirected. The set of all undirected graphs is called $\mathbb{G}_u$ (see definition 16 for reference).

Definition 16: The set of undirected graphs $\mathbb{G}_u$

$$\mathbb{G}_u = \left\{ (V, E) \;\middle|\; \begin{array}{r} V \neq \emptyset \\ \wedge E \subseteq \mathbb{E}_u \\ \wedge \forall e \in E : e \subset V \end{array} \right\}$$

A weighted graph is a 3-tuple containing the same information as the basic graph: A set of vertices V and a nonempty set edges E. The weighted graph also contains a function $w : E \to \mathbb{R}_+$. $w(e)$ is called the weight of the edge e. The set of weighted edges is called $\mathbb{G}_w$ (see definition 17 for reference).

Definition 17: The set of weighted graphs $\mathbb{G}_w$

$$\mathbb{G}_w = \left\{ (V, E, w : E \to \mathbb{R}_+) \mid (V, E) \in \mathbb{G}_d \cup \mathbb{G}_u \right\}$$

The weight of a weighted graph is the sum of weights of all the edges in the graph (see definition 18 for reference).

Definition 18: The weight $w((V, E, f))$ of a weighted graph (V, E, f)

$$w((V, E, f)) = \sum_{(e \in E)} f(e)$$

A multigraph is a graph in which the same edge can appear twice. An undirected multigraph is a nonempty set of vertices and multiset of edges. The edges can either be directed or undirected (see definition 19 for reference).

Definition 19: The set of unweighted multigraphs $\mathbb{G}_m$

$$\mathbb{G}_m = \left\{ (V, (E, q)) \;\middle|\; \begin{array}{r} (V, E) \in \mathbb{G}_u \cup \mathbb{G}_d \\ \wedge (E, q) \in \mathbb{M} \end{array} \right\}$$

A weighted multigraph is simply a multigraph with a weight function, similar to what is defined in definition 17 (see definition 20 for reference).

Definition 20: The set of weighted multigraphs $\mathbb{G}_{mw}$ is defined by:

$$\mathbb{G}_{wm} = \left\{ (V, (E, q), w) \;\middle|\; \begin{array}{r} (V, E, w) \in \mathbb{G}_w \\ \wedge (E, q) \in \mathbb{M} \end{array} \right\}$$

A subgraph of a graph G is a graph whose set of vertices is a subset of the set of vertices in G and whose set of edges is a subset of the set of edges in G. If G is a weighted graph then the weight function of the subgraph is the same as the one in G. The statement that a graph S is a subgraph of a graph G is denoted $S \subset G$ (see definition 21 for reference) .

Definition 21: Subgraph (A, B) of the graph (C, D)

$$(A, B) \subseteq (C, D) \leftrightarrow \begin{pmatrix} A \subseteq C \\ \wedge B \subseteq D \end{pmatrix}$$

$$(A, B, v) \subseteq (C, D, w) \leftrightarrow \begin{pmatrix} (A, B) \subseteq (C, D) \\ \wedge \forall e \in B : v(e) = w(e) \end{pmatrix}$$

An Euler graph is a graph in which each vertex has an even degree (see definition 22 for reference).

Definition 22: The set of Euler graphs $\mathbb{G}_e$

$$\mathbb{G}_e = \left\{ (V, E) \;\middle|\; \begin{array}{r} (V, E) \in \mathbb{G}_u \cup \mathbb{G}_d \\ \wedge\ \forall v \in V : \deg(v, E) \bmod 2 = 0 \} \end{array} \right\}$$

A matching $M((V, E))$ in a graph (V, E) is a subset of E such that every vertex in V is present in at most one edge in M (see definition 23 for reference). If a vertex of V is present in an edge of M it is said to be matched.

Definition 23: The set of matchings $M(G)$ of a graph G

$$M((V, E)) = \{M | M \subset E \ \wedge\ \forall v \in V : \deg(v, M) \leq 1\}$$

A perfect matching $PM((V, E))$ in a graph (V, E) is a matching in (V, E) such that at most one vertex of V is not matched. (see definition 24 for reference)

Definition 24: The set of perfect matchings $PM(G)$ of a graph G

$$PM((V, E)) = \left\{ PM \;\middle|\; \begin{array}{c} PM \in M((V, E)) \\ \wedge |\{v | v \in V \ \wedge\ \deg(v, M) = 0\}| \leq 1 \end{array} \right\}$$

4.2.3 Paths and circuits

A path in a graph G is a n-tuple of vertices from G in which each vertex is connected to the next in line by an edge in G. Like with edges, the first and last vertex in the n-tuple is called the initial and terminal vertices. The set of all paths in a graph G is called $P(G)$ (see definition 25 for reference).

Definition 25: The set of paths $P((V, E))$ in a graph $(V, E$

$$P((V, E)) = \left\{ (v_1, v_2, ..., v_n) \;\middle|\; \begin{array}{l} \{v_1, v_2, ..., v_n\} \subseteq V \\ \wedge \forall v_i \in \{v_1, v_2, ..., v_{n-1}\} : \begin{pmatrix} E \in \mathbb{E}_u \rightarrow \{v_i, v_{i+1}\} \in E \\ \wedge E \in \mathbb{E}_d \rightarrow (v_i, v_{i+1}) \in E \end{pmatrix} \end{array} \right\}$$

A path in which the initial and terminal vertex are the same is called a circuit. The set of all circuits in a graph G is called $C(G)$ (see definition 26 for reference).

Definition 26: The set of circuits $C((V,E))$ in a graph (V,E)

$$C((V,E)) = \left\{ (v_1, v_2, ..., v_{n-1}, v_n) \;\middle|\; \begin{array}{l} (v_1, v_2, ..., v_n) \in P((V,E)) \\ \wedge v_1 = v_n \end{array} \right\}$$

A Hamilton circuit in a graph G is a circuit in G which contains all the vertices of G exactly once. Consequently the degree of each vertex in respect to the edges in the Hamilton circuit is exactly 2. (see definition 27 for reference)

Definition 27: The set of hamilton circuits $HC((V,E))$ in a graph (V,E)

$$HC((V,E)) = \left\{ (v_1, v_2, ..., v_n) \;\middle|\; \begin{array}{l} (v_1, v_2, ..., v_n) \in C(G) \\ \wedge |(v_1, v_2, ..., v_n)| = |\{v_1, v_2, ..., v_n\}| \\ \wedge \{v_1, v_2, ..., v_n\} = V \end{array} \right\}$$

An Euler circuit in a graph G is a circuit in G which contains all the edges of G exactly once (see definition 28 for reference).

Definition 28: The set of Euler circuits $EC((V,E))$ in a graph (V,E)

$$EC((V,E)) = \left\{ (v_1, v_2, ..., v_n) \;\middle|\; \begin{array}{l} (v_1, v_2, ..., v_n) \in C(G) \\ \wedge \{\{v_1, v_2\}, \{v_2, v_3\}, ..., \{v_{n-3}, v_{n-2}\}, \{v_{n-1}, v_n\}\} = E \end{array} \right\}$$

4.2.4 Trees

A tree is a graph in which no circuit exists (see definition 29 for reference).

Definition 29: The set of all trees $\mathbb{T}$

$$\mathbb{T} = \left\{ G \;\middle|\; \begin{array}{l} G \in \mathbb{G}_u \cup \mathbb{G}_d \cup \mathbb{G}_w \\ \wedge C(G) = \emptyset \end{array} \right\}$$

The set of spanning trees $ST(G)$ in an unweighted graph G is the set of all trees which connects all vertices of G using the edges from the set of edges of G (see definition 31 for reference).

Definition 30: The set of spanning trees $ST((V,E))$ in an unweighted graph (V,E)

$$ST((V,E)) = \left\{ (V,B) \;\middle|\; \begin{array}{l} (V,B) \in \mathbb{T} \\ \wedge B \subseteq E \\ \wedge \forall v \in V : \exists E \in B : v \in E \end{array} \right\}$$

The set of weighted spanning trees $ST(G)$ in a weighted graph G is the set of all trees which connects all vertices of G using the edges and weights from the set of edges of G (see definition 31 for reference).

Definition 31: The set of weighted spanning trees $ST((V, E, w))$ in the weighted graph (V, E, w)

$$ST((V, E, w)) = \{(V, B, w) \mid (V, B, w) \in ST((V, E))\}$$

The set of minimum spanning trees $MST(G)$ in a weighted graph G is the set of weighted spanning trees of G which has the minimum weight of all the weighted spanning trees of G (see definition 32 for reference).

Definition 32: The set of minimum spanning trees $MST(G)$ in a weighted graph G

$$MST(G) = \left\{ T \;\middle|\; \begin{array}{l} T \in ST(G) \\ \wedge \forall S \in ST(G) : w(S) \leq w(T) \end{array} \right\}$$

4.3 The Travelling Salesman Problem

Though the layman description, in section 1.1, basically explains the concept of TSP, a more thorough describtion can be made. This is done in this section, based on the descriptions in Kenneth H. Rosens "Discrete Mathmatics and Its Applications"[7] and Gerhard Reinelts "The Travelling Salesman"[4]. TSP can be described as set of combinatorial optimization problems. The solution is finding the shortest way through a number of locations, by going through each location only once and returning to the point of origin. TSP related problems are numerous. Practical problems, like, minimizing the distance travelled between cities or planning network connections and wiring in electrical components, can be identyfied as a TSP. Abstract problems like, sequencing jobs on a machine, taking up the least possible amount of time, can also be solved in this manner. Using graph theory, TSPs can be described as finding the shortest possible Hamilton circuit (see definition 27) in a weighted, undirected graph $G(V, E, w)$. The vertices in V each represents a location and the weight $w(E)$ of any edge $e \in E$ represents the distance between the vertices of V. There are several approaches to TSP, but the most straightforward method would be to find all possible Hamilton circuits and then picking the one with the least possible weight as the solution. While this is a very simple approach, it would become increasingly time consuming as the amount of vertices in the graph increases. Based on the information in source [7], the total amount of Hamilton circuits in a graph with n vertices is $(n-1)!$. This means that finding the complete set of circuits in a graph with a high amount of vertices is largely beyond what one can expect to do within a reasonable timespan. However methods exist for finding a Hamilton circuit with a weight close to the minimal possible weighted Hamilton circuit in a graph. One of these is an algorithm known as "Christofides' algorithm".

4.3.1 Christofides' algorithm

Christofides' algorithm describes a series of steps used to find an approximated, short Hamilton circuit in a complete weighted graph (see figure 4.1 on page 24). A Hamilton circuit found by correctly following the steps of Cristofides' algorithm is at most 1,5 times as longer than the optimal Hamilton circuit [4]. This means, by successfully implementing Christofides' algorithm to create a Hamilton circuit to solve our logistics problem, we will get what we deem is a satisfactory result.

In this report we will not explicitly look at the runtime of the algorithms used in Christofides algorithm. Results provided by source [4] says the runtime of a successful implementation can be done in $n * log(n)$ time, where n denotes the number of vertices in a graph. As the number of vertices increases, the runtime increases logarithmically, meaning runtime of the algorithm grows massively as the amount of vertices increases. We have used the version of Christofides' algorithm provided in source [4] as the model for the one we want to use in the product. The only distinction between our implementation of Christofides' algorithm and the one in the source, is the creation of the minimum perfect matching algorithm used within Christofides' algorithm. We reckon that it is possible to use the already provided runtime approximation, when evaluating the runtime of the algorithm. However if our implementation of the algorithm proves to run longer than the time estimated in source [4], we do not think it as an actual problem - as long as the runtime of the entire product, does not exceed what is defined by our demands. The steps of the algorithm are described below.

1. Compute a minimum spanning tree $MST(G)$, from a complete, simple, weighted graph $G(V, E, w)$.
2. Compute a minimum weight perfect matching M for all the odd-degree-vertices in V of $MST(G)$. Add the subset of edges M to E in $MST(G)$ to create an Euler graph $EG(V, E)$.
3. Find an Euler circuit $EC(G(V, E))$ in the Euler graph $EG(V, E)$.
4. Use the Euler circuit $EC(G(V, E))$ to get a Hamilton circuit $G(V, E, w) = HC(EC(G))$, which has no greater length than the previously found Euler graph.

The various steps of Christofides' algorithm is done using a set of other algorithms, that perform the required steps for finding a satisfactory Hamilton circuit in a graph. Writing pseudo code for Christofides' algorithm and the algorithms it uses to compute a Hamilton circuit will make it easier for us, when designing a test of the solution for the problem presented in the case. The following text describes these algorithms, their concepts and the actual pseudo code. A pseudo code of Christofides' algorithm is seen in algorithm 1. The algorithms used within are presented in the following sections. Figures 4.2, 4.3, 4.4 and 4.5 on the following pages, shows the output from the various steps of the algorithm.

Algorithm 1 Christofides $(G(V, E, w))$

$G(V, E, w)$: A complete weigthed simpelgraph

1: $v \leftarrow u, u \in V$ ▷ *v, the starting vertex for prims*
2: $MST(V, E, w) \leftarrow Prims\ (G(V, E, w), v)$ ▷ *MST, the minimum spanning tree*
3: $EG(V, E) \leftarrow CreateEulerGraph\ (MST(V, E, w))$ ▷ *EG, the Euler graph created*
4: $EC(v_1, v_2, ..., v_n) \leftarrow FindEulerCircuit\ (EG(V, E))$ ▷ *EC, the Euler circuit found*
5: $HC(v_1, v_2, ..., v_m) \leftarrow GetHamilton\ (EC(v_1, v_2, ..., v_n))$ ▷ *HC, the Hamilton circuit*

Christofides' algorithm returns a Hamilton circuit $HC(v_1, v_2, ..., v_m)$ found in $G(V, E, w)$

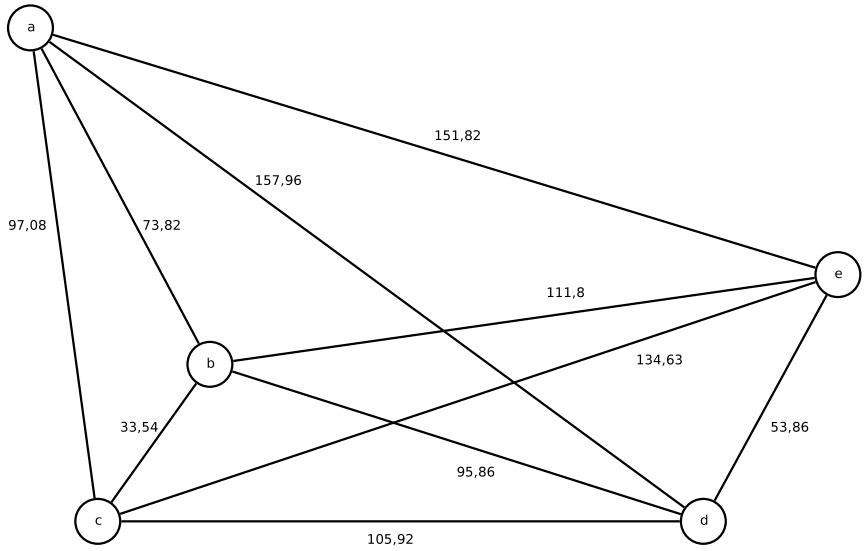


Figure 4.1: This figure shows an example of an input graph on which Christofides' algorithm can be used to find the shortest path. The vertices a, b, c, d and e are connected to each other so the graph is a complete, weighted graph.

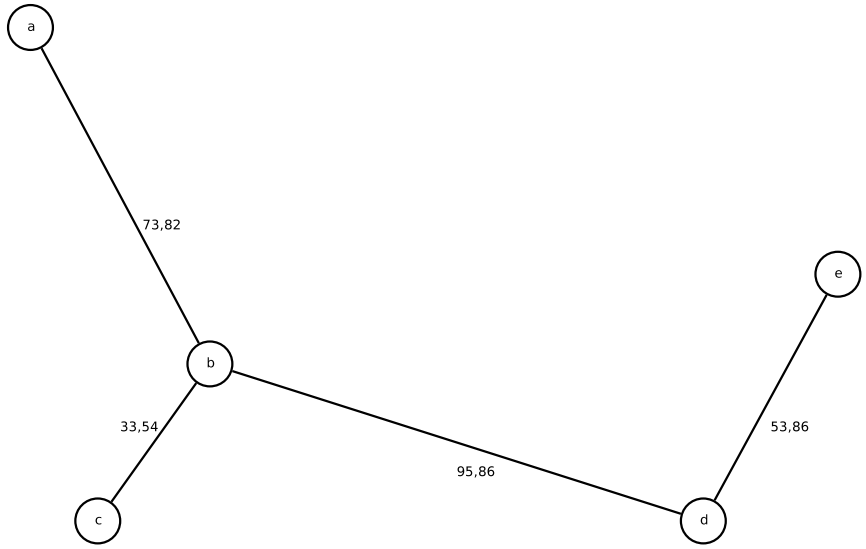


Figure 4.2: This figure shows a minimum spanning tree found in the input graph in figure 4.1 using Prims algorithm introduced in section 4.3.2 on the next page. A minimum spanning tree is characterized by the fact that no circuit exists in it, and the sum of all the edge weight is minimum

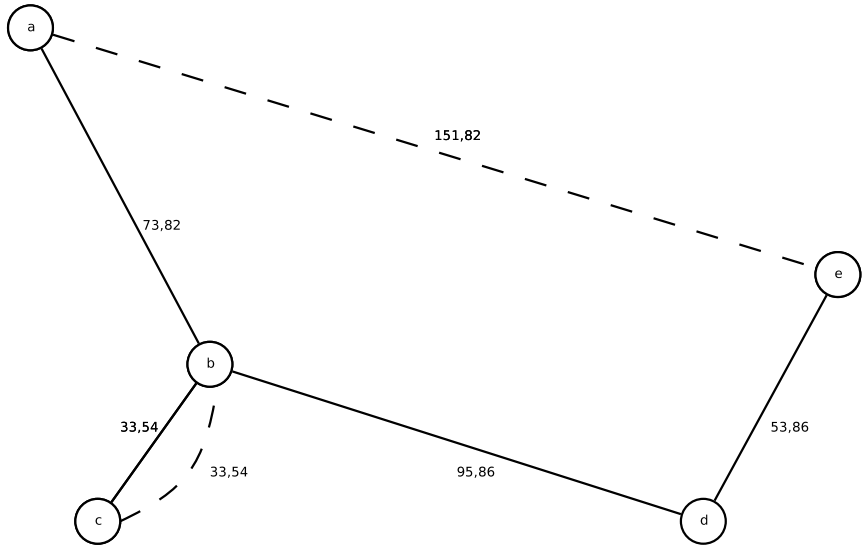


Figure 4.3: This figure shows an Euler graph found by finding the minimum weighted perfect match and adding this subset of edges to the minimum spanning tree in figure 4.2. The edges added to the minimum spanning tree are shown by dotted lines.

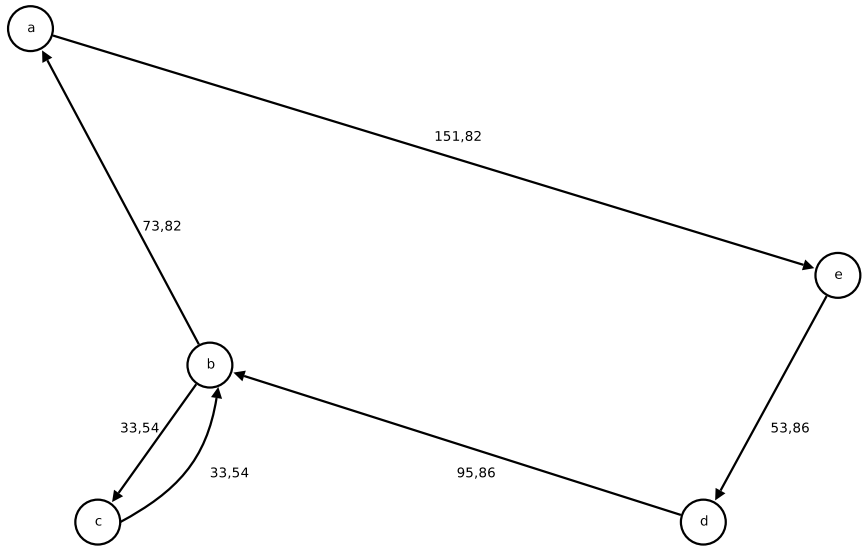


Figure 4.4: This figure shows an Euler circuit found in the Euler graph from figure 4.3. The circuit is represented by a sequence of vertices found by traversing the edges of the Euler graph. In this graph the Euler circuit would be the following sequence of vertices: a, e, d, b, c, b, a.

4.3.2 Prims algorithm

One method to construct a minimum spanning tree in a weighted, undirected graph is to use a known algorithm called Prim's algorithm[7]. We define the function $Prims(G(V, E, w), v)$ as

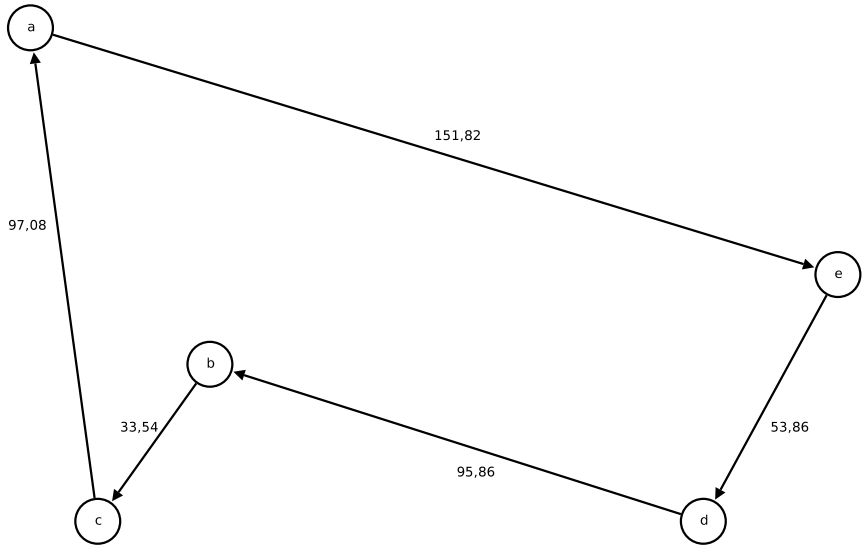


Figure 4.5: This figure shows the output of the final step of the algorithm, where a Hamilton circuit has been found. The Hamilton circuit is formed by travelling through the sequence vertices of the Euler circuit seen in figure 4.4, exactly once and returning to the starting vertice. In this graph the Hamilton circuit is traversed by going from vertex to vertex in the following order: a, e, d, b, c, a.

a function which returns a minimum spanning tree from the entire set of existing minimum spanning trees, that can be formed with the vertices existing in the undirected, weighted graph $G(V, E, w)$ (definition 17). The pseudo code for the algorithm can be seen in algorithm 2 on the facing page. Prims algorithm works by successively adding the edges with the least weight of the input graph to a new graph, until all the vertices are included in the tuple U at which point it stops. The method of doing this is adding edges from E to T, consisting of a vertex already in U and a vertex of the old graph not yet in the new graph. This ensures that the end result is a connected tree without circuits, as the algorithm cannot add an edge to T, where both vertices are already contained in U. At the same time as it generates a tree, it will ensure it is a minimum weighted tree, as it picks the edges with the minimal weight, of all the edges in the input graph. Thus the overall weight of the tree is ensured to be minimal (see figure 4.2 for reference).

4.3.3 Creating an Euler graph

The next step of Christofides' algorithm is to construct an Euler graph, as defined in definition 22, using the minimum spanning tree found during the last step. This is done by adding edges to the minimum spanning tree until every vertex in it has an even degree. We do this by picking all vertices in the input tree with odd degree, and constructing a new complete subgraph with these vertices. We now construct the minimum weighted perfect match, as defined in definition 24, by choosing the edge in the subgraph with the minimal weight and adding it to U, removing the two vertices in the edge from the subgraph and then continue doing this until there are no vertices left in the graph. We now have a number of edges which we add to the edges in the minimum spanning tree received as input. Once these edges are

Algorithm 2 Prims$(G(V, E, w), v)$

$G(V, E, w)$: The Graph in which we want to create the minimum spanning tree
v: The starting vertex of the tree.

1: $A \leftarrow \{b | adj(v, b, E)\}$ ▷ *the set A is created consisting of vertices adjacent to v*
2: $u \leftarrow b, \{v, b\} = \min(E, w) \wedge b \in A$ ▷ *the vertex u, is the second vertex in the tree*
3: $T \leftarrow \{\{v, u\}\}$ ▷ *T is the set of edges in the tree*
4: $U \leftarrow \{u, v\}$ ▷ *U is the set of vertices in the tree*
5: **while** $U \neq V$ **do**
6: $e \leftarrow \min(\{u, v | u \in U, v \in V \backslash U\}, w)$
7: $T \leftarrow T \cup \{e\}$
8: $U \leftarrow U \cup e$

The 3-tuple (U, T, w) now represents a minimum spanning tree in the graph $G(V, E, w)$

added to the previously found minimum spanning tree, the graph is a correct Euler graph, since the vertex degrees of the newly formed graph are now all equal, as shown in figure 4.3. The pseudo code for the algorithm can be seen in algorithm 3 on the next page.

4.3.4 Finding an Euler circuit

Once an euler graph has been found, the next step of Christofides' algorithm is to compute an euler circuit in the newly found Euler graph. Algorithm 4 on page 29 can be used to compute an Euler circuit in an Euler graph $G(V, E, w)$. The following description is based on the graphs seen on figure 4.6 on page 30. Initially the simple circuit $P\{a, b, c, a\}$ is created. As c is a vertex in the circuit with a degree higher than 0, the circuit is bisected, meaning it is split into the two paths $O\{a, b\}$ and $Q\{a\}$. A new simple circuit consisting of the tuple $P\{c, d, e, c\}$ is then created. The three tuples O, P, Q are then concatenated into a single tuple $P\{a, b, c, d, e, c, a\}$. The procedure is then repeated, where c again is the vertex in P with a degree higher than 0. This bisects the tuple P into the two paths $O\{a, b\}$ and $Q\{d, e, c, a\}$. A new simple circuit consisting of the tuple $P\{c, f, g, c\}$ is then created. The three tuples O, P, Q are then concatenated into a single tuple $P\{a, b, c, f, g, c, d, e, c, a\}$. The tuple P is now an Euler circuit, as there are no vertices in P with a degree higher than 0

4.3.5 Finding a Hamilton circuit

Once an euler circuit has been found in a graph, it can be used to construct a Hamilton circuit. Using definition 27, a short path in a graph is found, satisfying the criteria of solving a travelling salesman related problem. The algorithm for making a Hamilton circuit is described in algorithm 5 on page 31. Algorithm 5 starts with initially picking a vertex in the Euler circuit, found during the previous step of Christofides' algorithm and adds it to the tuple Q. It then subsequently adds the next vertex of the Euler circuit to Q and the edges between the the new element and the previous element of Q to the set Y, until all vertices are in Q and the edges of the circuit between these vertices is in Y. Y will now contain a Hamilton circuit, which has a weight no larger than the weight of the previously found Euler circuit, shown in figure 4.5 on the preceding page.

Algorithm 3 CreateEulerGraph($G(V, E, w)$)

$G(V, E, w)$: A weighted minimum spanning tree

1: $i \leftarrow 0$
2: $U \leftarrow \emptyset$ ▷ *U is a set of vertices with odd degree*
3: **while** $i \neq |V|$ **do** ▷ *Check all vertices, if they have odd degree, add them to U*
4: **if** $(deg(v_i, E)\ mod\ 2) \neq 0$ **then**
5: $U \leftarrow U \cup v_i$
6: $i \leftarrow i + 1$
7: $PM \leftarrow \emptyset$ ▷ *PM is a perfect matching, a set of edges*
8: **while** $U \neq \emptyset$ **do** ▷ *As long as there are still vertices with odd degree, generate the Perfect Matching*
9: $l \leftarrow 0$
10: $k \leftarrow 1$
11: $C \leftarrow \emptyset$ ▷ *C, edges of a complete graph with vertices in U*
12: **while** $l < (|U| - 1)$ **do** ▷ *Construct the complete graph C with the vertices in U*
13: **while** $k < |U|$ **do**
14: $C \leftarrow C \cup \{u_l, u_{l+k}\}$
15: $k \leftarrow k + 1$
16: $l \leftarrow l + 1$
17: $k \leftarrow l + 1$
18: $m \leftarrow min(C, w)$
19: $PM \leftarrow PM \cup \{m\}$
20: $U \leftarrow U/m$
21: $E \cup PM$

The 3-tuple (V, E, w) now represents the euler graph

Algorithm 4 *FindEulerCircuit*$(G(V, E))$

$G(V, E)$: Euler graph.

1: $s \leftarrow v, v \in V$ ▷ *s is the starting vertex, a random vertex v of V*
2: $P \leftarrow (s)$ ▷ *P is a tuple of vertices, which contains the route*
3: $t \leftarrow v, v \in \{v|adj(s, v, E)\}$ ▷ *t is the next vertex on the route(random neighbor pick)*
4: $P \leftarrow P + (t)$
5: $E \leftarrow \{E \backslash \{s, t\}\}$
6: **while** $t \neq s$ **do** ▷ *Build a circut in the Euler graph*
7: $u \leftarrow t$ ▷ *u, remembers the previous chosen vertex*
8: $t \leftarrow v, v \in \{v|adj(u, v, E)\}$ ▷ *t, the next vertex on the route, which is adjacent to u*
9: $P \leftarrow P + (t)$
10: $E \leftarrow \{E \backslash \{u, t\}\}$
11: **while** $E \neq \emptyset$ **do** ▷ *As long as there are edges left in the eulercircuit, construct subcircuits*
12: $t \leftarrow v, v \in \{b|b \in P \wedge deg(b, E) > 0\}$ ▷ *t, a vertex which still is in a edge, if it is not, the dog will return 0*
13: $O \leftarrow ()$ ▷ *O contains the route P, from start until the vertex t*
14: $i \leftarrow 1$
15: **while** $(P_i \neq t)$ **do**
16: $O \leftarrow O + (P_i)$
17: $i \leftarrow i + 1$
18: $i \leftarrow i + 1$
19: $Q \leftarrow ()$ ▷ *Q contains the rest of P after the vertex t.*
20: **while** $(i \leq |P|)$ **do**
21: $Q \leftarrow Q + (P_i)$
22: $i \leftarrow i + 1$
23: $s \leftarrow t$ ▷ *s is now the new start vertex*
24: $t \leftarrow v, v \in \{b|adj(s, b, E)\}$
25: $P \leftarrow (s, t)$
26: $E \leftarrow E \backslash \{s, t\}$
27: **while** $t \neq s$ **do**
28: $u \leftarrow t$ ▷ *u, remembers the previous vertex*
29: $t \leftarrow v, v \in \{b|adj(u, b, E)\}$
30: $P \leftarrow P + (t)$
31: $E \leftarrow E \backslash \{u, t\}$
32: $P \leftarrow O + P + Q$

The tuple P now represents an euler circuit, a tuple of vertices.

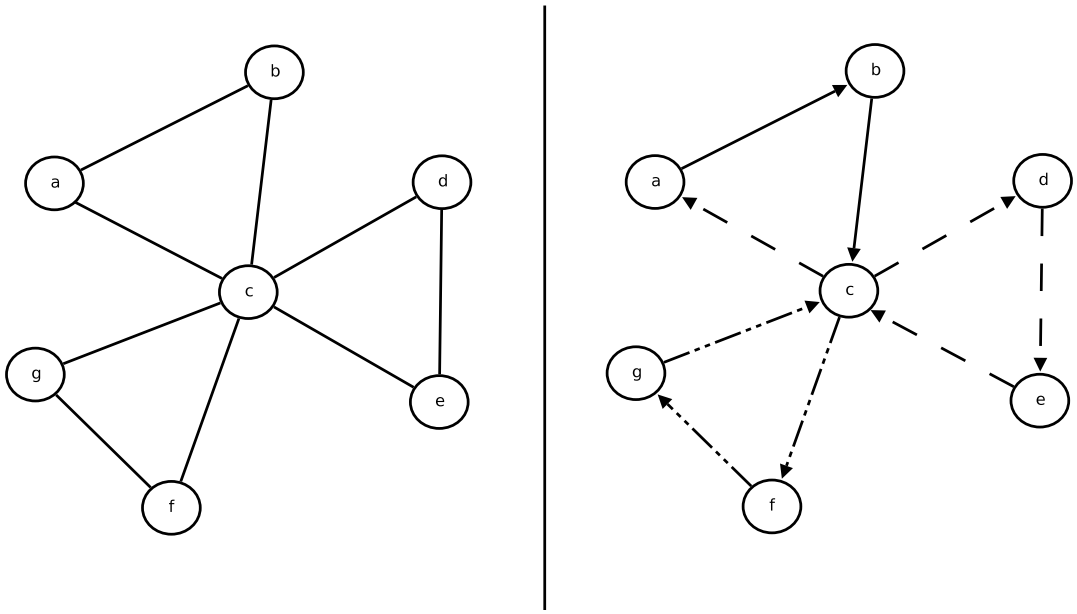


Figure 4.6: This figure shows two graphs. The left graph is an Euler graph in which we are going to find an Euler circuit. The right graph shows what happens on lines 11 to 32 in algorithm 4 on the preceding page. See section 4.3.4 on page 27 for a description of the procedure.

4.4 Theory summary

We now have the tools necessary to work with graphs in a sensible way, and the algorithms needed to compute a short Hamilton circuit in a graph. A route produced by Christofides' algorithm is a short circuit, which lives up to our demand: finding a short route. The input data required is a graph, with respective graph data such as vertices and edges, meaning if we are to solve the problem posed in the case, we have to translate some kind of order data into data represented by a graph. We have not calculated specific runtime estimations of the pseudo code. This means we could risk ending up with a system, that will not produce a valid output, within a reasonable amount of time.

Algorithm 5 GetHamilton($P(v_1, v_2, ..., v_n)$)

$P(v_1, v_2, ..., v_n)$: An Euler circuit from a complete graph G.

1: $Q \leftarrow (v_1)$ ▷ *Q, the tuple with the Hamilton circuit*
2: $n \leftarrow |\{v_1, v_2, ..., v_n\}|$ ▷ *n the number of different vertices in P*
3: $j \leftarrow 2$
4: **while** $|Q| < n$ **do**
5: **if** $v_i \notin Q$ **then**
6: $Q \leftarrow Q \cup (v_i)$
7: $i \leftarrow i + 1$
8: $Q \leftarrow Q \cup (v_1)$ ▷ *Add the startpoint at the end of the circuit*

The tuple Q now contains a Hamilton circuit from the complete graph $G(V, E, w)$

CHAPTER

5

Development

Before we can start a development process, it is necessary to establish some rules and standards concerning the overall work process. In the next section we will outline two known development methods along with the specific method we will use. First off we will describe a simple and straightforward method. After this we describe the Waterfall method, which is deeply rooted in the commercial sector. Next we describe the eXtreme Programming method, which is a relatively new approach to development. After reviewing these three methods, we have created our own method, which is a combination of Waterfall and eXtreme Programming.

The demand specification will then be defined, based on a combination of the problem formulation, theory, program outline and the practical problems. This will lead to the description of the program design and a sketch of how to implement the theory described.

5.1 Development method

The simplest development method, consists of only two steps. It resembles the method developers often would use, when working on small projects with low complexity. The first step is an analytical step where the developer analyzes the problem to figure out how it can be solved. The second step includes everything from coding to testing and release. This method is very effective on small projects with one developer, but in larger projects, with greater complexity and more developers, it can be very difficult to control.

We have chosen to describe the Waterfall method and eXtreme Programming (XP), because these methods have a very different approach to a development process and we believe we can use some elements from both methods. They are both well known in the commercial sector as well as in the scientific and research sector. The Waterfall method is an old, and well tested method, for planning the aspects of a software development process. XP is a modern method adapted to newer development projects and larger scale software productions. Our description of XP is build on the considerations done in the human aspects of programming from "Human Aspects of Software Engineering" [10] and the more practical approach in "Extreme Software Engineering: a hands on approach" [9].

5.1.1 Waterfall method

The Waterfall method is a programming method that resembles a waterfall, by steadily going through the process step-by-step. The method was first described by Dr. Winston W. Royce in 1970, where he described the method as "... risky and invites failure [8]". Nevertheless the method has become widespread today. Royce's waterfall method consists of 7 phases, as seen on figure 5.1 on the facing page. We have chosen to base our description of the Waterfall method on Royce's original paper [8]. Each phase must be fully completed before the next phase can be started. If there is something in a earlier phase, that needs to be changed, the development must be restarted from that phase. If the change has any effect on the following phases, all these phases must be remade.

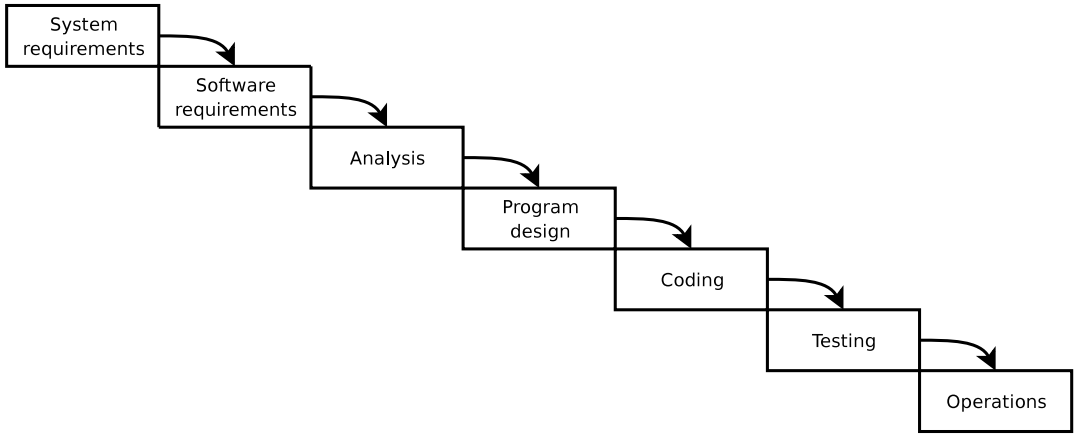


Figure 5.1: The 7 phases of Royce's Waterfall method.

In the first and second phase, the hardware and software requirements of the project are laid out by the costumer and all the requirements are examined. When and only when these phases are completed, is it possible to continue to the next phase. This phase is the analysis phase, where all the problems are examined and ways to solve them are found. When the analysis phase is finished, the design phase is started. In the design phase a very detailed "blueprint" of the software is made. The "blueprint" can contain flowcharts or other forms of outlines of the solution. When the "blueprint" is finished, the coding phase can begin. If the developers have to deviate from the "blueprint", they can be forced to start all over with the design phase. When the coding phase is finished it is time to show the costumer the product. This initiates the test phase. If the costumer is satisfied with the solution, it is possible to continue to the operations phase. In the operations phase the software is implemented and the software is maintained. If the costumer is not satisfied, it can become a problem. If the "blueprint" is flawed, the developers needs to go back to the design phase and redesign the "blueprint". If that does not solve the problems, the team needs to start all over with the project.

5.1.2 eXtreme Programming

XP is an agile method of software development. Agile programming methods are more tied to the human aspects of software development[1]. When a development team works with the XP method, it brings the customer in as a part of the development team. In this way the project builds on a closer contact between the customer and the developers. This means many problems and misunderstandings between the customer and the developers, can be solved and eliminated before they have any large effect on the product. An XP development team works in a non-hierarchal manner, meaning every developer on the team has the same responsibility and authority with respect to the produced software. Any team member could in fact have done any task. In general this leads to three guidelines; whoever notices a problem fixes it, whoever is working on a particular piece of the system designs and writes it and whoever discovers a better way of doing something implements it.

One of the cornerstones in XP is pair programming. This means that no team member

[1]See the "Manifesto for Agile Software Development" at http://agilemanifesto.org/

ever writes any code on his own. The developers always work in pairs of two with only one workstation, one keyboard and one mouse. A pair consists of two roles whilst working. The "driver" works the keyboard and mouse and writes the code, while the "navigator" looks for defects in the code. This leads to many defects being discovered in the first development phase and to better knowledge sharing. If a team member leaves the team, no private knowledge is lost and development can go on without further hinderances.

Everything in XP is done in small steps to make it easier to plan, perform and evaluate, with the added benefit of small steps being easier to undo. The small step philosophy builds on flexibility, simplicity, lowered risk and feedback. Every small step of the process has to be complete before you start the next. The idea is to always have a product, that can compile and run, which gives the developers and the customer a lift in morale by always seeing the product perform the wanted processes.

The 13 steps of XP

XP consists of 13 practices that are to be followed by the development team. They describe the whole process from planning the development phase, to the development of the final product and the interaction with the customer. The 13 practices are shown in the core cycles of XP in figure 5.2 and described below.

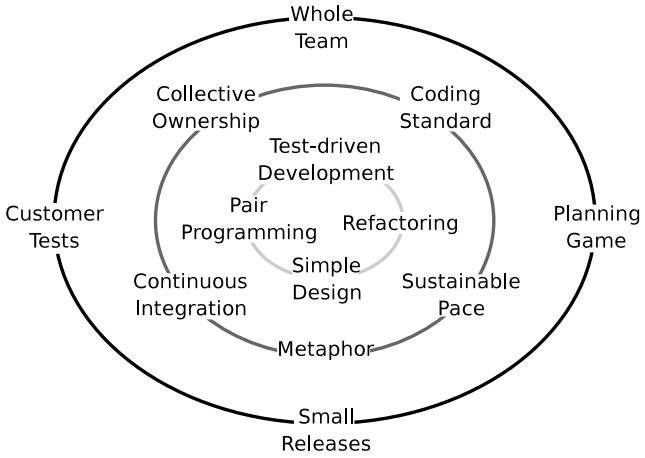


Figure 5.2: The 3 core cycles of the XP method. The outer cycle contains the steps that define the overall structuring and rules of an XP team. The middle cycle defines the rules and practices of the development team. The inner cycle defines the practical approach of the development team

Outer cycle

Whole team. XP removes the barriers between the customer and the development team. They all have authority to take part in critical decisions made during the development process and share the responsibility for the direction of the development. The development team consists of different roles: e.g. developers, testers, a project manager and others. All roles are filled by different team members at different times.

The planning game. The planning of every development step in XP is done through a

"planning game" between the customer and the developers. The customer expresses his or her goals trough "user stories", specifying the overall behaviors of the software. The developers then take these stories and estimate the costs. This gives the customer the ability to prioritize the stories and in this way control the direction of the project. When a step is done, the developers evaluate the cost estimates of the program and use this data to improve their estimations in the next step.
Small releases. XP teams work with frequent releases. Every iteration takes two to three weeks to complete and end with delivering a tested and working sample of code to the customer. This way the customer can test the code and plan the output of the next iteration.
Customer tests. The customer develops acceptance tests, that can test if the delivered software lives up to the user story. These tests are automated so the developers can run the tests frequently and use them to test the state of the current "user story".

Middle cycle

Coding standards. To ensure that the code does is not in different code writing standards and to ensure readability, the team is to follow one coding standard throughout the development process. The choice of standard however, is not nearly as important as consistently following it.
Sustainable pace. To ensure that the same amount of work is put into every iteration of a program, the team works at a sustainable pace throughout the development process. This is also known as the 40-hour week. This is done to avoid employee "burnouts" and mistakes made due to overtime work, and ensures that the team in the long run can develop better software in shorter amounts of time.
Metaphor. It is important that the whole team works with a common analogy, a vision of how the program works, also known as a "metaphor". This is to ensure that everyone knows the right place to look up a function or put a functionality they are about to add.
Continuous integration. This means that all code must compile and pass all tests before it is added to the system. Every time code is added to the system, it must be able to pass all those criteria too, meaning continuous additions demand multiple builds of the whole system. This is often done several times a day.
Collective code ownership. An XP team as a whole, owns all the code in the system. This means who originally wrote the code is not important and is often forgotten in the design process. Anyone can change any code at any time without asking anyone for permission. This demands a revision control system, when two or more pairs work on the same code.

Inner cycle

Test-driven development. The first thing the developers do when they have set the next intermediate goal, but before writing the code, is to write an automated test. The test is made to verify the code they write satisfies the intermediate goal. When the code satisfies the intermediate goal, the test is added to the overall test suite. This is run continuously, providing feedback on the development system.
Design improvement (refactoring). When the team discovers shortcomings or deficiencies in the code, they refactor the code to make it incrementally better. With the simple design from the start and the small improvements by induction, XP teams tend to produce code with good design characteristics.
Simple design. The design of the system is always to be kept as simple as the functionality allows. The design is not to extend further than the next iterations features.
Pair programming. All code delivery to the customer is written by pairs. Two developers

always work at one workstation with one keyboard; one writes the code and the other supports in whatever role appropriate at the given time. The developer not writing the code, helps detecting typing errors and comes with implementation suggestions. The roles may change at any given time.

5.1.3 Our method

In a software production of the size we plan to develop, and when working as a group of 7 developers, we need an overall structure and method to control our work. We have chosen to define our own method based on XP, with an overall structure that resembles the Waterfall method.

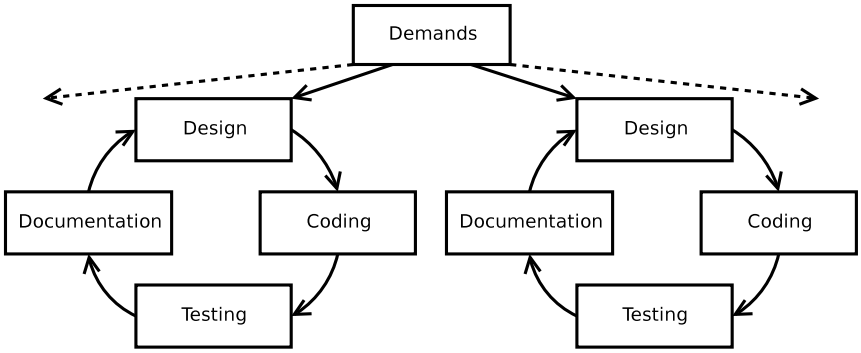


Figure 5.3: The overall structure of the development method we have chosen.

In figure 5.3 we show the overall structure of our development method. The first step, "demands", is inspired by the "requirements" phase in the Waterfall method, and derives from a mixture of earlier steps. This step is static throughout the development process and is built on the requirements, we have found through our case analysis and product outline. Our demands affect the design of our program design and is used to control, that the final tests of the entire product returns the data we planed.

The design, coding, testing, documentation and design process, is an iterative process inspired by XP. From the requirements, we have an overall idea of the functionalities we have to develop and a general product design. This is broken down into small steps, which can be designed, coded and tested separately. When a step passes the tests we have planned in the design phase, it is marked as completed. When this step is recorded in the documentation, we continue to the next small iteration step. Nothing is committed to the system, before it can compile and has passed the tests defined in the design phase.

Our code writing method is built on the concept of XP. We work in pairs in all parts of the iterative process, while the demands phase is defined in the group. Pair programming is also used as a mean to heighten our programming skills. By placing an experienced programmer with a less experienced programmer, we should save time on repairing erroneous code and at the same time share knowledge within the group. This is based on the fact, that we will use the programming phase as a knowledge sharing- and learning process, opposite of only producing a product. It is not possible for us to have the "customer" as part of our development team, because we do not have an actual customer. This means that steps like "the planning game" and "customer tests" are not used and " the whole team" only includes our development group.

By combining Waterfall and XP we get the opportunity to sort out some of the disadvantages from both the methods and use only the advantages. For instance if ,when using Waterfall, there is a design flaw in the "blueprint", it can be necessary to start all over with the project, but from our point of view a strict design plan is essential. This way we have a goal to strive for, and something to compare with. Thereby it is easier to see if we have met the objectives set at the beginning of the design phase. Using our method we have the opportunity to use multiple development strings. With this we mean multiple groups working parallel with each other.

5.2 Demand specification

Throughout the preceding sections we have implicitly introduced a number of demands to our program design. However, to specifically know what our product must be capable of, we have to list a series of demands, we later can test the prototype against. At the end of the problem analysis, we implied formulated general demands to the product; however they are not clear enough to be used in a demand specification. This means we have to specify the demands more thoroughly in order to know whether the program provides a correct output, according to the problems we wanted to solve in the problem formulation.

In our program outline in section 3.4.4, we mentioned that we want the output to be usable in practice. If a truck is sent on a route, which requires a larger quantity of goods to be delivered than a truck can carry, the truck would have to refill somewhere on the route. To prevent sending a second truck to refill the first one, or having the first truck return to the storage to refill, we will have to introduce demand 1.

Demand 1: No route can exists such that any truck available is incapable of handling it.
A route must not contain a larger quantity of goods, than a single available truck can carry.

A truck can not be at two locations at one time. If our program outputs such routes, we do not satisfy our goal from the program outline in section 3.4.4, which is to produce an output usable in practice. This amounts to demand 2.

Demand 2: No two routes assigned to the same truck can exists in the same span of time.
A truck can only handle one route at a time.

We want to deliver as many orders as possible in the available time. Since we cannot skip orders beyond what demand 5 allows, what we want to do is to minimize the length of the routes. This is necessary because the number of trucks is limited, and otherwise it is uncertain if the trucks have time enough to deliver all the orders. In theory section 4.3.1 about Christofides' algorithm, we know that the algorithm should return a Hamilton circuit, which is at most 50% longer than the optimal minimum circuit. This results in demand 3, which satisfies the problem 6.

Demand 3: The length of each route should be kept at a minimum.
The length of each route should be at most 50 % longer than the shortest route, since the number of trucks available are limited.

A truck driver should not exceed the regulations for driving and resting hours, as it is strictly prohibited by law. These rules are very complicated, so we have reduced them to the simple demand 4, based on the information given to us during the case in section 3.2.2.

Demand 4: Obey driving regulations.
It should be possible for a truck to drive and deliver cargo on any route assigned to a truck within 10 hours. If a truck is assigned more than one route, the total time taken to drive these routes can not exceed 10 hours.

Every order has to be delivered before its deadline is exceeded. In most cases this is the day after the order is entered into the system, which is done immediately after the customer has placed the order. A few customers are located so far away from the storage, that the orders have their deadline extended by 24 hours extra, before they are to be delivered. This means we have different deadlines. The amount of orders might be too comprehensive to handle on a single day, since orders with a 48 hour deadline are also considered when making routes. We can allow that some or all of the 48 hour deadlines are not handled during route planning, but the rest of the orders must be planned for delivery the following day. All this results in demand 5, which supports problem 7 on page 12

Demand 5: Route deadlines
All orders which are due within the next 24 hours must be delivered. Orders with a deadline which exceeds 24 hours does not have to be delivered, but can wait to the next day.

Every route should take as much advantage of the truck it is assigned to as possible. While keeping demand 4 in mind, a truck should carry as much load as it can when leaving storage. This means that if a route is made with a relatively small quantity of goods, the smallest available truck able to carry the goods should be used. This is based on problem 6 and amounts to demand 6 and 7.

Demand 6: Use the smallest trucks for the smallest quantities of goods.
The truck assigned to a route should be the smallest available truck with sufficient freight capacity.

Demand 7: Utilize maximal truck capacity
Every time a truck leaves storage it should, if possible, carry as much cargo as possible for delivery. However this demand can be overridden by demand 4, as this takes higher precedence, since that demand is mandatory to obey by law.

As problem 4 states, the output by the program has to be ready for use before the trucks are ready to be loaded. Thus the prototype should be able to run with as many as 500 orders in the time span between 21:00 and whenever the first truck needs to start loading the next day. This results in demand 8.

Demand 8: The product must have a reasonable runtime
The time it takes for the program to run, using an input with as many as 500 orders, must be less than the time between 21:00 and whenever the first truck has to start loading the next day.

As we now know the strict demands to the program, we are ready to begin the program design.

5.3 Program design

In this section we will describe the beginning of the developing phase of our program. This will describe our thoughts and plans for this program. We learned in our theory chapter, that to be able to process our input data it would be a good idea to translate the input data into

a graph. This graph can be built as a model of a real situation; the model is built as a map, with the customers and the storage represented as vertices, and the road between positions represented as edges in a graph. We need to split this graph into a number subgraphs. Every subgraph will contain a route for a truck, where the route through the vertices of the subgraph must be as short as possible. The magnitude of the subgraph is also limited by the fact, that the cargo to be delivered to the customers cannot exceed a trucks capacity, as described in demand 1. The routes in the subgraphs are then to be constructed using the Christofides algorithm, previously described in section 4.3.1. As a result of splitting the original input graph into a number of smaller subgraphs, the amount of vertices in a given input graph is likely to be very limited. As the amount of vertices is limited, the amount of machine operations needed to create a route, can most likely be done within a reasonable time. Initially the runtime of our product is specified as a demand from the "customer", which means that if the product can run and provide an acceptable output within that time, the product performs satisfactory.

Our early plan for the program is as follows: The first step of route generation, is to choose a number of orders, which is located strategic well in relation to each other. The orders is chosen by a stating point and in section 5.3.2 on page 41 we describe different ways of selecting orders. We continue choosing orders until there is enough to fit a truck.

The procedure of choosing orders should try to build routes which uses as much capacity of the trucks as possible and let the smallest truck which can handle route, drive it. This is nessesary to satisfy demand 6 and 7. If no trucks have time for a route, this procedure returns an empty route. This tells the program, that we can not make a complete output with the available trucks.

After the orders are chosen, we use Christofides' algorithm on the extracted subgraph containing the customers and the storage to build a route. The route is saved and the customers associated with the orders, are removed from the order map. This process then continues until all orders are included in a route, and the order map is empty.

Demand 2 of the demand specification requires that every truck only has one route at a time. To comply with this demand, we keep score of the available driving hours left on each truck. When a route is completed, the time it takes to complete the route is substracted from the available driving hours of the truck which the route was assigned to. This also allows us to check whether a truck has time for a given route, so that we can comply with demand 5.

Some of the customers are located far away from the storage, and these customers have different deadlines than the rest. The normal deadline for delivery is 24 hours, but these customers have a 48 hour deadline. Our program should still try to deliver the cargo to these customers, but if the trucks available can not reach every customers on the graph a given day, some of the customers with the 48 hour deadlines will be removed from the graph. They will be postponed for delivery/route generation the next day instead, where these orders then gets a 24 hour deadline.

We now have the components to make a rough description of the program in the form of a flowchart, where the various elements of the program are depicted as they depend on each other. This flowchart, which can be seen on figure 5.4 on the following page, will help us later on when we are going to identify the functions needed and how these should interact. The following sections describes the elements of the program.

5.3.1 Initialization of the program

The first step of the program is to acquire the data later used to compute routes. The input data it receives is a list of customers represented by; a customer ID, a coordinate set and the cargo ordered by the customer. This data effectively represents points in the two-dimensional

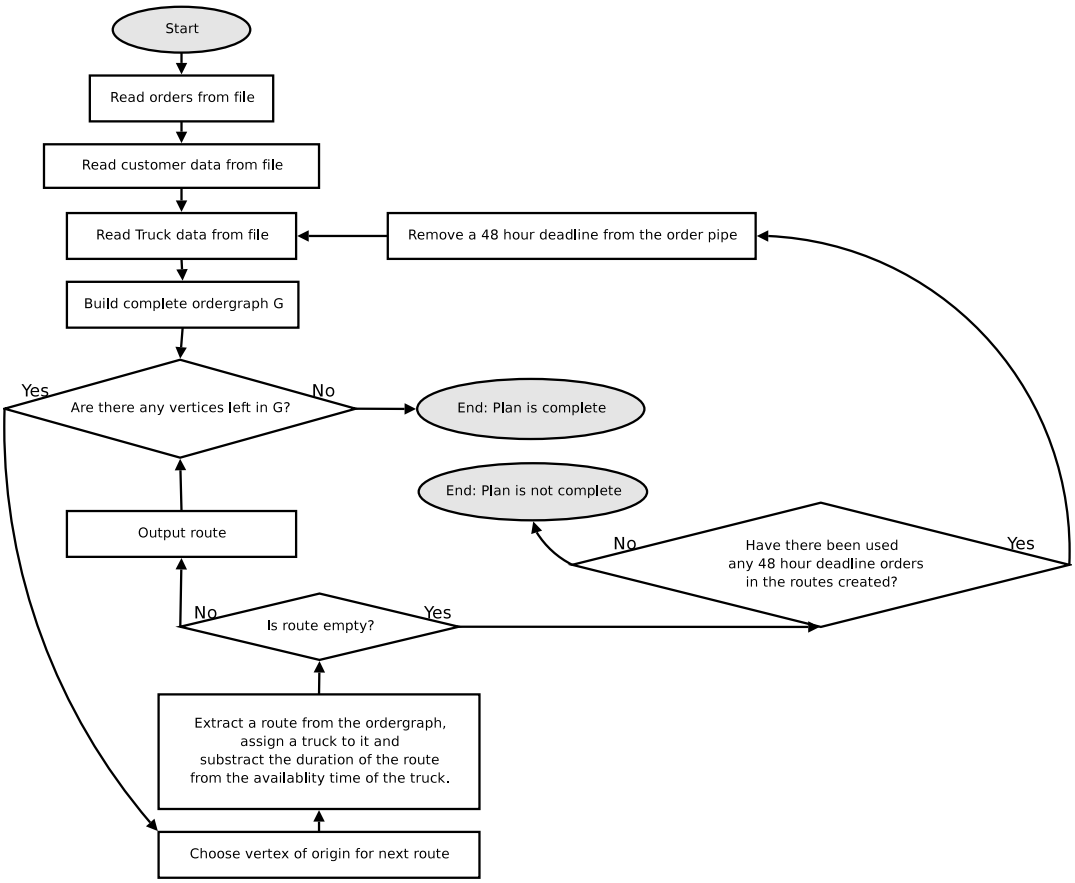


Figure 5.4: This figure shows the design of our program. The operations the program has to be able to do are shown, along with which order to do it in to produce a satisfactory output.

plane. This two dimensional plane contains the entire set of customers, and is defined as the order graph. This graph is then made into a complete graph by connecting all vertices to each other with edges. The weight of the edges is found by calculating the euclidean distance between all the vertices in the graph, and assigning the distance to each edge. The reason for doing this, opposed to manually giving each edge between the customers a fixed weight, is that it allows the prototype to automatically calculate these weights. This means the weight between two points is not a realistic measure for the time it takes to drive from one customer to another. However, the advantage is that we do not have to define the weight of each edge, every time we want to use another input dataset. The prototype will calculate the weight of the edges for us. This will save us a lot of work, when we are going to test our prototype during the coding phase and later when the prototype is finished, as we are going to thoroughly test it. Once all the edges are found and the weights are calculated, the order graph will be a complete, weighted graph. This allows us to easily find the weight value of a given edge, when we later are going to find routes in the order graph.

5.3.2 The route generation process

One of the problems we could encounter in our program, is that some times there can be more orders in the route subgraph, than one truck can service. The subgraph generated for the route could be too large, causing the time it takes for the truck to service all the customers to exceed the available driving hours of the truck. Another limitation would be if the maximum cargo capacity of the truck were exceeded, as defined in the demand 1 on page 37. We need to find an efficient method of finding a subgraph. We have found three simple ways of finding subgraphs, each with advantages and disadvantages, which we need to consider, before picking one for the program.

Picking a starting vertex

The first step of creating a subgraph is picking a start vertex. We have come up with three ways of picking the starting vertex of the subgraph; picking a starting vertex randomly, picking the closest vertex or picking the furthest vertex from storage. Each type of starting vertex has advantages and disadvantages, which we need to consider, before picking one to use in our product. The three starting vertices are characterized by being either; randomly chosen, the customer closest to storage or the customer furthest away from storage.

"Random customer" Picking a random starting vertex will ensure that customers are equally likely to receive a delivery as ordered, since there is no preferred vertex of start for a route. The disadvantage is that there will be starting vertices randomly scattered in the input graph, and as such once subgraphs have been picked out of the input graph there might occur "holes" without any customers. This might leave some customers isolated, which will demand additional resources if a truck is to visit that vertex before deadline. As optimizing the delivery process is a goal of the project, this is not a desirable scenario.

"Closest customer" Picking the closest costumer to storage as the starting vertex of the subgraph will ensure that a lot of deliveries can be made in a short period of time. The trucks can quickly get back to storage and refill, however it might become a problem to reach the customers further out before the deadline for deliver has been reached. Because of the driving hour regulations, it might not be possible to drive out to a costumer and get back once a truck already has been out on a route. This means that in a worst case scenario, some customers will not receive their goods on time which is the main goal of the delivery process.

"Furthest customer" Picking the furthest costumer to storage as starting vertex of the subgraph will ensure that the customers in the fringe areas will get their goods delivered on time. Once deliveries in the fringe areas has been done, the trucks are then free to take up routes in areas closer to storage. This means the trucks are more likely to use up all their drive hours every day, while delivering as much freight per day as possible, since most of the time spent is spent driving freight. The disadvantage of this method is that the customers closest to storage might be neglected if all resources are used driving goods to the customers far away from storage.

Extracting a subgraph for a route

Once a starting vertex has been picked we have come to three ways of picking a subgraph in the input graph. As such we need to initially pick one method for our prototype. Each of the three different methods of generating a subgraph, has its own characteristic way of doing it.

We have dubbed the three methods "The Radius Method", the "Minimum Spanning Tree Method" and "The Snake Method".

"The Radius method" Once a starting vertex has been picked, a subgraph containing the storage, the starting vertex and the edge between them is formed. Subsequently the vertices closest to the starting vertex and the edge between them and the starting vertex are added to the graph. This is done until a truck has reached its storage capacity. When the truck have reached its maximum storage capacity a route is generated; if the accumulated driving time of the route exceeds the maximum driving time of the truck, the vertex furthest away from the starting vertex is removed from the subgraph. Then a new graph is generated if the accumulated driving time still exceeds the maximum driving time of the truck, another vertex is removed. A graphical representation is seen on figure 5.5.

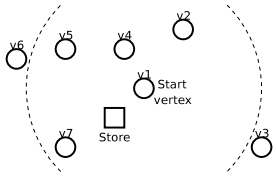


Figure 5.5: Initially the subgraph consists of the storage and the starting vertex. An imaginary radius based on the weight between edges is then expanded from the starting vertex. If a neighboring vertex is within the radius, it is added to the subgraph. This procedure continues until the maximum number of customers a truck can service, have been found or the route exceeds the allowed driving time of the truck.

"Minimum Spanning Tree Method" Initially the subgraph consists of the storage and the starting vertex. A minimum spanning tree is then formed with the starting vertex as the initial vertex. The vertices and edges between them are then added to the subgraph. This is done until a truck has either reached its maximum load capacity, or the time it takes driving the route has reached the driving hours maximum as earlier defined. A graphical representation is seen on figure 5.6.

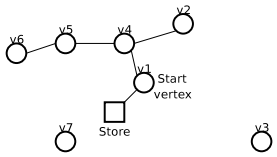


Figure 5.6: In the Minimum Spanning Tree edition, the subgraph initially consists of the storage and a starting vertex. A Minimum Spanning Tree is then built from the starting vertex using Prims algorithm.

"The Snake method" Once a starting vertex has been picked, a subgraph containing the storage, the starting vertex and the edge between them is formed. Then the vertex closest to the starting vertex, and the edge between them, is then added to the subgraph. Subsequently the vertex closest to the last chosen vertex and the edge between the two is added to the

subgraph. This is done until the storage capacity of a truck is exceeded or the time it takes to drive the route exceeds driving hours regulations. A graphical representation is seen on figure 5.7.

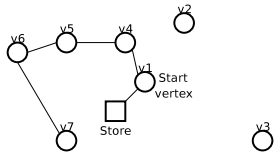


Figure 5.7: The subgraph initially consists of the storage and the starting vertex. The vertex closest to the vertex last added to the subgraph is then added to the new subgraph. This procedure continues until the cargo capacity of a truck is reached, or the driving time of the truck is exceeded.

The chosen method

Of the three methods of picking starting points, we find choosing the customer furthest from storage as the most efficient one. Picking this method will give us a good idea of how much time is left for driving the route. The time it takes to get back to storage, is at most twice that of the time it takes to drive to the starting vertex. This is true, as the starting vertex is the vertex furthest away from storage. In contrast it would be required to calculate an estimation of the possible end customer and the distance back to storage, if we used the other two methods of picking starting points. The other advantage seems to be, that once all the fringe customers have gotten their deliveries, there will be customers close to storage that a truck can spend the rest of its driving time delivering cargo to. To generate subgraphs we are initially going to use the "Radius method", as it will ensure the customers are nicely clustered. This means that appropriate "chunks" of the input graph is systematically removed in the fringes, ensuring only few holes in the modified input graph. The "Minimum Spanning Tree edition" works very similar to the "Radius Method" except the tree formed would look slightly different as not all vertices are connected to the starting vertex. It is hard to tell which of the three subgraph selection methods is most efficient, but we have a notion that the "Minimum Spanning Tree Edition" might provide some odd results, if used under certain conditions. If it is combined with the "Closest customer" starting vertex method, it might construct a Minimum Spanning Tree, where the route crosses back and forth past the storage. The "Snake Method" looks like it would generate long snake like subgraphs in which routes are to be made. The result from using the snake method seems illogical at times and it might be confusing for a later user of the route, who might question its efficiency. A very peculiar trait of this method, is that the subgraph it creates looks very much like something, which easily can be made into a finished Hamilton circuit. This means it would not be necessary to use Christofides' algorithm on it. However if we used this method, it would be quite hard to control, if it is in fact a short path satisfying demand 3 of the demand specification.

Once a subgraph has been made, a complete weighted graph is constructed using the vertices in the subgraph, which is then fed into Christofides' algorithm.

Route extraction with the radius method

Algorithm 6 on page 45 is used to extract a route from our order graph. The algorithm works as described in "The Radius method" in section 5.3.2. The algorithm takes an order graph,

a storage vertex, a start vertex and a truck list as input. The subgraph selection is done by picking the vertex closest to the start vertex and adding it to a route graph. This is done until the amount of cargo on the route is enough to fill the largest truck available. When this criteria is satisfied, the algorithm checks if there exists a truck, which can complete the route within its remaining driving hours. If not the algorithm removes a vertex from the order, thereby reducing the cargo to be delivered. Then it checks if there is a truck available to drive the route. This procedure is done until the cargo is reduced enough for an available truck to take the route. When there are more trucks available to take a route, the algorithm selects the truck with the smallest capacity able to handle the route, and subtracts the route length from the avilable driving hours of the truck. When all these steps are completed successfully, the algorithm removes the vertices of the route graph from the order graph, and returns the route. The storage vertex which was removed from the order graph, is immediately added again, as it it used later to extract a new route subgraph. If there is no truck available to take a given route, the algorithm returns an empty route. The algorithm uses the functions and procedures defined below.

$\mathbb{T}$ is the set of all trucks.
$capacity : \mathbb{T} \rightarrow \mathbb{R}$
$distanceleft : \mathbb{T} \rightarrow \mathbb{R}$
$subtracttimefromtrucktime(Distance,\ Truck)$

Capacity returns the capacity of a given truck. $Distanceleft$ is a function that returns distance, which the truck has time left to drive. $Subtracttimefromtrucktime$ subtracts the time it take to drive *distance* from the available driving hours of a truck.

5.4 Development summary

By using the algorithms found in theory section 4.3.1 and the knowledge obtained until now, we have designed a plan for a software solution, we believe capable of solving the problems presented in the problem formulation in section 3.3. The next step is then to create a prototype to test if our program design is in fact able to satisfy the criteria of the demand specification.

Algorithm 6 ExtractAndFindRoute$(s, z, (V, E, w), T)$

s : Start vertex.
z : Storage vertex.
(V, E, w) : The complete order graph.
T : The set of trucks available.

$U \leftarrow \{s, z\}$ ▷ *U is the vertices for the route.*
$B \leftarrow \{\{s, z\}\}$ ▷ *B is the edges for the route.*
$t \leftarrow \max(T, \text{capacity})$
while $\sum_{u \in U} \text{orderquantity}(u) < \text{capacity}(t) \wedge |U| < |V|$ **do** ▷ *Selects orders to the route by increasing radius from the startvertex, until truck is full or there is no vertices left.*
  $v \leftarrow a, \{s, a\} = \min(\{e | e \in E \backslash B \wedge s \in e\}, w)$
  $U \leftarrow U \cup \{v\}$
  $B \leftarrow B \cup \{\{s, v\}\}$
for $a \in U$ **do** ▷ *Add edges to B until the graph (U, B) is complete.*
  for $b \in U$ **do**
    if $a \neq b$ **then**
      $B \leftarrow B \cup \{\{a, b\}\}$
$C \leftarrow \text{Christofides}((U, B, w))$
$Y \leftarrow \left\{ t \in T \;\middle|\; \sum_{c \in C} \text{orderquantity}(c) \leq \text{capacity}(t) \wedge \sum i = 0^{i=|c|-1} w(\{c_i, c_{i+1}\}) \leq \text{distanceleft}(t) \right\}$
while $Y = \emptyset \wedge |U| > 2$ **do** ▷ *If there are no trucks available which can handle the order in its current form, or there is only start and storage vertex in graph*
  $v \leftarrow a, \{s, a\} = \max(B, w)$
  $U \leftarrow U \backslash \{v\}$
  $B \leftarrow B \backslash \{\{e | v \in e\}\}$
  $C \leftarrow \text{Christofides}((U, B, w))$
  $Y \leftarrow \left\{ t \in T \;\middle|\; \sum_{c \in C} \text{orderquantity}(c) \leq \text{capacity}(t) \wedge \sum_{i=0}^{i=|c|-1} w(\{c_i, c_{i+1}\}) \leq \text{distanceleft}(t) \right\}$
if $Y = \emptyset$ **then**
  $C \leftarrow ()$
$y \leftarrow \min(Y, \text{capacity})$
subtracttimefromtrucktime$(\sum_{c_i \in C} w(\{c_i, c_{i+1}\}), Y)$
for $v \in U$ **do** ▷ *Remove the vertices of the graph (U, B) from the graph (V, E, w).*
  $E \leftarrow E \backslash \{\{e | v \in e\}\}$
  $V \leftarrow V \backslash \{v\}$

The circuit C is now the route and the truck y is the smallest available truck which can handle the route.

CHAPTER

6

Evaluation

In this chapter we describe how we have implemented a prototype based on the plan of the program in section 5.3, and how it performs according to the criteria we set for it during the demand specification in section 5.2. The reason for making a prototype of the program, is to prove that the concept we have developed during the development process, is in fact usable. We have implemented the prototype in ANSI C. At the beginning of this section, it is described how we have chosen to represent the different types of data used in the prototype, along with a description of how we have implemented Prims algorithm. Then the overall structure of the prototype is defined, showing what files the program uses and how they interact with each other. Lastly the content and format of the input and output files is described as they are used in the prototype, and the various tests we have performed on the prototype are provided. These tests consist of a demand specification test and a time test, each performed to check if the demands of the demand specification are met and if the prototype can provide a valid output within a reasonable amount of time.

6.1 Prototype structure

We have opted to design our program in modules. This means we have designed the source code of the prototype, such it entirely consists of a number of smaller, independent functions. Splitting the prototype into modules enables us to program the various subparts independently of each other, making it possible to simultaneously program the prototype. This also gives us the ability to test the functions as they are programmed, making sure each part works as intended, when they are assembled at the end of the programming process. The program is divided into files, such that the main file of the program calls the functions it needs for constructing the output from the other files of the program. These files contains implementations of various parts of the program, such as the theory we got from math and graph theory and the functions to read data used by the main file. These files is represented graphically in figure 6.1 on the facing page.

First the prototype reads the data from the three files OrderBase.txt, CustomerBase.txt and TruckBase.txt. Then the orders and amount of trucks are forwarded to the algorithms mentioned during the development chapter to construct the different routes. These routes are stored in individual files called Route_1.dot, Route_2.dot and etcetera, as graphs in dot code which Graphviz[1] can compile to a graphical representation. Together with the routes a file is created with all order locations depicted along with the storage. To ease the testing phase, the following files are created as well; Route_report.txt, Truck_report.txt and Routepipe_report.txt. They consist of accumulated data from the program, for example each trucks utilization ratio and other information.

[1] A set of programs available at http://www.graphviz.org

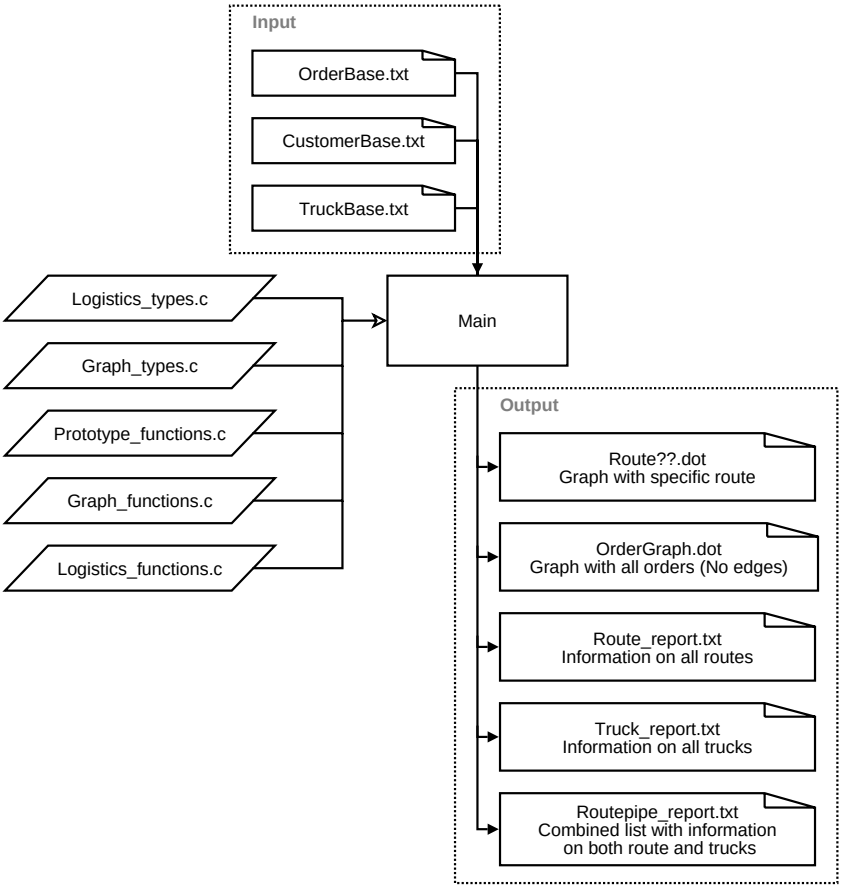


Figure 6.1: The files in the top contain input data. The ones on the left is files written in C. And finally the files at the bottom is the output.

6.1.1 Data structure

To make it more logical arranged and more transparent, we have chosen to divide the data into two main categories; logistics and graphs. To describe the structure of the data in the prototype, we use ANSI C structs. Our choice in data types is built on the assumption, that the compiler used to compile the prototype supports up to 32Bit signed integers (-2147483648 to 2147483647). To group the data types we have defined, we have chosen to make lists of pointers (arrays). An example of these lists, is the TruckList defined in this section. It contains an integer that defines the amount of trucks in the list and a pointer to the first in a list of trucks. Another more complicated list is the VertexSet defined later in section. Again it contains an integer that defines the amount of vertices in the set and a pointer to the first element of a list of elements. But instead of pointing directly at a vertex, the elements are pointers to vertices. This i done to avoid copying the vertex information when creating new lists and graphs and to ensure that the vertices are comparable throughout the program.

Logistics

```
struct Customer
{
    int Id;
    int X,Y;
    char *Name;
};
typedef struct Customer customer;
```

The customer type is used to store all the information we need to represent a customer. Id is an integer value used as a reference to the customer, which comes from the customer base which is a list of all the customers. X and Y are the coordinates of the customers geometric position in the 2d plane and Name is a string containing the name of the customer. It is easy to add more information about the customer to this struct as needed. The functions using the list, just need to be modified accordingly.

```
struct Truck
{
    int Id;
    double Capacity;
    int SecondsLeft;
};
typedef struct Truck truck;
```

The truck type is used to store information about a single truck. It contains an Id as a integer value for later reference to the truck. Capacity is a value that describes the size of the truck and is comparable to the size of an order, as described later in this section. It is used to calculate how many orders the truck can actually manage. SecondsLeft describes how many seconds the truck, and driver are still allowed to be out driving routes, and is used to determine if the truck can be assigned another route.

```
struct TruckList
{
    int Size;
    truck *Trucks;
};
typedef struct TruckList trucklist;
```

TruckList is a list that makes it easier to locate the trucks available in the prototype. Size is an integer value that tells how many trucks are on the list. Trucks is a list of pointers to the trucks, used to locate the trucks on the list.

```
struct Order
{
    int CustomerId;
    customer *Customer;
    time_t Deadline;
    double Quantity;
};
```

```
typedef struct Order order;
```

The order struct represent an order in the prototype, and contains all the information needed to handle an order. The CustomerId is the Id of the customer, who are to receive the order. Customer is a pointer to the customer. Deadline is a time stamp that contains information of when the order needs to be delivered to the customer. Quantity is the size of the order and tells how much space it will need in a truck. It is comparable to capacity in the truck struct.

```
struct OrderPipe
{
    int Size;
    order *Orders;
};
typedef struct OrderPipe orderpipe;
```

An orderpipe is a list of orders that makes it easier to locate the orders. Size tells how many orders are in the system. Orders is a list of pointers to the actual orders in the orderpipe.

```
struct Route
{
    path *Circuit;
    truck *Truck;
};
typedef struct Route route;
```

A route consists of a path as described later, which contains an ordered list of the vertices attached to the route. The circuit defines in which order the orders should be delivered. Truck is a pointer to the truck assigned to the route.

```
struct RoutePipe
{
    int Size;
    route *Routes;
};
typedef struct RoutePipe routepipe
```

A routepipe is a list of routes, containing pointers to the Routes and an integer value Size, which defines the amount of routes in the routepipe.

Graphs

```
struct Vertex
{
    order *Order;
};
typedef struct Vertex vertex;
```

Vertex is in the graph theory group and is our representation of a vertex. A Vertex is just a pointer to an order, where all the information we need for a vertex, can be found in the

order struct. Making this struct as a wrapper[2] for the order struct makes it easier to add more options later if needed.

```
struct Edge
{
    vertex *Init;
    vertex *Term;
    double Weight;
};
typedef struct Edge edge;
```

This is our representation of an edge in a graph. It consists of two vertex pointers, Init and Term; the initiating and terminating vertices in the edge. This essentially makes it a directed edge, but we can also work with it as undirected. The Weight double gives us the ability to attach a specific weight to an edge. In our prototype the weight value is automatically computed by a weight function, where the weight is the distance between the two vertices in the edge. If Weight initially is manually set to something different from 0.0, then the weight calculated by the weight function is overridden by this value.

```
struct VertexSet
{
    int Cardinality;
    int MultiSet;
    vertex **Vertices;
};
typedef struct VertexSet vertexset;
```

VertexSet contains a list of vertices. Vertices is a list of pointers to the vertices in the set. Cardinality is an integer value that defines the amount of vertices in the set. MultiSet is a boolean that defines wether or not the set is a multiset. If Multiset has the value 1, the set is a multiset. If Multiset has the value 0, it is not a Multiset. If a set is a multiset, it can contain a vertex more than once.

```
struct EdgeSet
{
    int Cardinality;
    int MultiSet;
    edge **Edges;
};
typedef struct EdgeSet edgeset;
```

EdgeSet contains a list of edges. Edges is a list of pointers to the edges in the set. Cardinality defines the amount of edges in the set. MultiSet is a boolean that defines wether or not the edgeset is a multiset.

```
struct Graph
{
    double (*WeightFunction)(edge *E);
    vertexset VertexSet;
```

[2] A wrapper is a function that does not do anything else than call another function.

```
    edgeset EdgeSet;
};
typedef struct Graph graph;
```

The graph type combines a VertexSet and an EdgeSet to describe a graph. The WeightFunction is a pointer to the function used to calculate the weight of the edges in the graph. This makes it possible to define how to calculate the weight of the edges, when constructing a new graph.

```
struct Path
{
    int Size;
    vertex **Vertices;
};
typedef struct Path path;
```

Path is an ordered list of vertices used to describe the order of the vertices in a route. The integer Size defines the amount of vertices in the list of vertices.

6.1.2 Functions description

As mentioned in section 6.1, we have divided our program into multiple modules and functions. The following section serves as an example of how we implemented our pseudo code in the prototype. In the example we have used Prims algorithm seen in algorithm 2 on page 27. We choose to describe our implementation of Prims algorithm, because it is a widely known algorithm and relatively easy to overview. Throughout this example, references to Appendix A.5 on page 73 are made where the ANSI C source code can be found. The source code for this and the rest of the implementation, is included on the Appendix CD-ROM.

Prims algorithm

Our Prims function receives a pointer to a graph and a pointer to a minimum spanning tree, which point at the graph structure in which the minnimum spanning tree is to be stored. Finally the function also receives a pointer to the start vertex in the graph. These inputs are also reflected in the C header for the function seen below.

```
void Prims(graph *Graph,graph *MinimumSpanningTree,vertex *StartVertex)
```

When implementing line 1 in the pseudo code we have to find a way to implement adj(v, b, E) in ANSI C. This is described below. Because this function needs a set of edges in which to test for adjencency, we have to iterate though all the edges in the edgeset to find all the edges that contains the start vertex which is seen on line 33 - 45 in the C code in appendix A.5.

On line 2 of the pseudo code we have to find the minimum edge in a edgeset. We have implemented C versions of the *min* function defined in definition 4 and a weight function which returnd the weight of an edge. These functions are also described below. When the minimum weighted edge has been found, the pointer to the edge is stored as in line 48 of the C code.

Line 3 of the pseudo code, introduces the addition of edges to our MST which is a set of edges. In C we can not just add an element to a set, because in C there is no such thing as a set. C only has lists known as arrays. To emulate the implimentation of sets the

function AddEdgeToEdgeSet, which is depicted in appendix A.1, only adds the edge if it is not allready present in the list in the edgeset struct. The edgeset can identify itself as a multiset, which allows for multiple instances of the same element in its list. Line 4 of the pseudo code needs the addition of a vertex to a set of vertices, this is done with a similar function called AddVertexToVertexSet.

In line 5 of the pseudo code, we compare two set of vertices. I our C implementation we used a different condition by comparing the size of the vertexsets, since, in the current context, when and only when the two sizes are equal the sets will be equal. The specific condition used can be seen in line 58 of the C code.

To implement line 6 of the pseudo code we constructed the set of edges in which one vertex was inside the MST and the other was outside it. From this set of edges the one with the minimum weight was chosen. In our implementation we have made this in two steps. First we make an "edgesetbuffer" to hold the edges from which to extract the minimum weighted. This "edgesetbuffer" is filled during an iteration through all the edges in the graph. When edges consisting of a vertex in our tree and one not inside the tree are found, they are added to the "edgesetbuffer". When this iteration is done, we simply choose the edge with the minimum weight, with the implementation of *min* mentioned above. This process can be seen in lines 64 through 86 of the C code.

Finally the C implementation stores the MST in the graph structure which was passed by pointer in the argument list.

```
int Adj( vertex A, vertex B, edgeset Edges )
```

This function, seen in appendix A.4 on page 72, gets two vertices as input plus a set of edges, and returns a boolean. The function iterates though all the edges in the edgeset and checks if the two vertices are represented in the edge. If they are both represented in an edge in the edgeset, the vertices are said to be adjacent in the edgeset and the function will return "1". If they are not both represented in an edge the function will return "0"

Euclidian weight

The EuclidianWeight function, which is depicted in appendix A.2 on page 70, simply receives an edge consisting, which it uses to calculate the euclidian distance between the initial and terminal vertices, which it returns as a double. The C header for this function is depicted below.

```
double EuclidianWeight(edge *E)
```

Minimum edge in set of edges

The function MinEdge, depicted in appendix A.3 on page 71, receives a set of edges and a pointer to a function for calculating the weight of an edge in the set. In the prototype this function is the EuclidianWeight function which is described above. The MinEdge function simply iterates though all the edges in the edgeset, while calculating the weight of each edge. Each time a weight smaller than the current minimum weight is found the current edge is set as the currently smallest edge. The C header for this function is depicted below.

```
edge *MinEdge( edgeset Set, double (*Function)(edge*) )
```

6.1.3 Prototype Input

As shown on figure 6.1 on page 47 the program gets input from 3 files; OrderBase.txt, CustomerBase.txt and TruckBase.txt.

OrderBase.txt

This file contains the order information and is structured like this:

```
CustomerId  Quantity    Deadline
0           3           1148279401
1           1.5         1148279401
```

Here an order is placed on one line, with the information separated by a tab, starting with the customer id. This is used to identify the customer in the customer file. The second element on a line is the quantity of the order. The last element is a time stamp in seconds, that defines when the order is to be delivered. Each line in this file is stored as an Order struct, as described in section 6.1.1.

CustomerBase.txt

This file contains the customer information and is structured like this:

```
CustomerId  Name            X       Y
0           Harald Nyborg   1532    -2341
1           Kvikly          10030   2134
```

Each line in the file contains the information of a customer, with the information separated by a tab starting with the customer id. The second element is the name of the customer, while the third and fourth elements are the X and Y coordinates of the customer represented on a 2D plane. Each line retrieved from this file is stored in a Customer struct, as described in 6.1.1.

TruckBase.txt

This file contains the trucks available and is structured like this:

```
TruckId     Capacity
0           15
1           30
```

The last input the program reads, is the truck file. This information is stored in the memory for the route planning. The information is kept in one line in the file with the information separated by a tab starting with the truck id. The second element of the line is the maximum load capacity the specific truck can handle. Each line retrieved from this file is stored in a Truck struct as described in 6.1.1.

6.1.4 Prototype Output

As shown in figure 6.1 on page 47, the program writes its output to 5 or more different files. These files are; Route_??.dot, OrderGraph.dot, Truck_Report.txt, Route_report.txt and Routepipe_report.txt. These files contain a graphical representation of the routes; a summary of how well the trucks have been used; descriptions of each route, and an overview of the general route plan, including how much time was used to run the program.

Route_??.dot

These files each contain a route as a graph. The question marks in the filename represents the id of the output file. For example, a file called "Route_13" would be the thirteenth route generated in a given input order graph. The route files are made in Graphviz format, so it is easy to create a graphical representation of the route. An example of a route output file converted to a graphical representation, can be seen on appendix B.1. Every customer and the storage for the specific route are depicted on the figure, along with the distance between each customer on the route. At the bottom of the figure, the total distance and the amount of vertices in the route is presented.

OrderGraph.dot

This file is an overview of all the orders in one single graph with no edges. Each order is plotted in the graph with its location in relation to the storage. Again the output is in Graphviz format for easy representation and readability.

Truck_Report.txt

This file is a summary of all the trucks used to drive the routes. An example of the contents is shown in the table below.

```
Truck Report
--------------------
Total number of trucks: 5
Truck no.   Size    Time Left (min) Utilization ratio
0           15      451.5           24.74
1           30      22.2            96.29
2           30      8.6             98.57
3           30      8.2             98.64
4           30      8.7             98.56
--------------------
Total Utilization ratio:83.36
```

It is seen that truck number zero has a capacity of 15 cargo units and still has 451.5 minutes left, which could have been used on the road. The utilization ratio is the amount of time used, compared to the total time available for each truck. The utilization ratio is a report, listing how well each truck is used compared to the driving hours limitations of 10 hours.

Route_Report.txt

This is a full list of all the routes created and some additional information. Below is an example of what the report of a route could look like. The below list is a segment of a file containing all the route descriptions.

```
Route number:            9
Assigned truck id:       0
Number of orders:        6
Length of route:         57.067 km.
Complete goods quantity: 5.000
```

```
----------The Route----------
Acer 31          1.000000
Plantronics 60   1.000000
Storage          0.000000
Iveco 86         2.000000
BMW 84           1.000000
Acer 31          1.000000
----------End Route----------
```

Route number is the specific route number, identifying the route. Assigned truck id identifies the truck assigned to the route. The number of orders, including the storage, is shown along with the length of the route and the total amount of goods transported. The Route describes the route in list form. Note that the storage is not necessarily the first and last item on the list. This is because of the way we generate the route, however the route is still valid, just not in the order it would be driven. It would be left to the administrator or driver to start the route at the right place, e.g. storage, and then drive to the rest of the customers in order, ascending or descending the list as appropriate.

Routepipe_Report.txt

This file contains summary information on all the routes created by the prototype. A comment in the end of the file, states if the orders were successfully delivered in time by the assigned trucks.

```
RoutePipe Report
--------------------
Total number of orders:   100
Complete Cargo quantity:  187.000

Route no.  Truck no.  Customers  Length (km)  Time (min)  Quantity (kolli)
1          0          19         145.287      384.3       30.000
2          1          16         151.753      362.1       28.000
3          2          18         124.103      348.9       29.000
4          3          19         128.941      364.7       29.000
5          2          12         91.005       249.2       18.000
6          1          12         81.138       237.3       19.000
7          3          13         63.636       226.3       23.000
8          0          7          50.570       150.7       11.000
-------------------
All orders included in route plan.
-------------------
Total Time Used: 3 sec.
```

6.2 Testing the prototype

The prototype is now ready to be tested according to the demands, specified in section 5.2. Due to complications, with memory allocation in the prototype, we have decided not to implement the function needed to remove a ”48 hour deadline” order. We found it more important to test the main functionalities of our program design, than use a lot of time debugging a faulty feature. This function would essentially restart the program after excluding

a single order from the total set of orders. However, this feature is not deemed necessary for testing of the remaining capabilities of the program design. The lack of this function means, that we can not test demand 5 using the current implementation of the program design.

To verify whether our program design would perform as expected, we expose the prototype to a series of tests. These tests are based on the demand specification in section 5.2. The input data used for these tests are randomly generated, but follows the structure described in section 6.1.3.

6.2.1 Demand specific tests

To properly determine if our prototype performs as intended, we have to extensively test it according to the criteria of the demand specification in section 5.2. We have chosen to use the same test data throughout this entire test to verify, that demand 1, 2, 3, 4, 6 and 7 in the demand specification are respected. Demand 8 will later be tested separately, in section 6.2.2, since it needs to gradual extend the amount of test data. Demand 5 will not be tested, because the conditions to be tested have been excluded from our prototype as described in section 5.4. The results of the test can be found on the Appendix CD-ROM in the directory "testdata/demandtest".

Method

To test each demand we have constructed a test with a basic data set, from which it should be easy to extract the data results. These results will then be held up against the demands of the demand specification, to see if they satisfy the individual demands. To test demand 3, we will try to manually improve the routes made by our prototype, to see if we can find better routes than the prototype. These routes are then to be compared with the output of the prototype. This should provide us with some measurement of how well the prototype performs, when compared with routes laid by hand using human logic.

Test results

The following list is the Truck Report and Routepipe Report from the test. A graph of the routes combined into one single figure is seen on figure 6.2 on the facing page. The test data consists of 30 orders with a total quantity of 68 pallets to be delivered. The customers are placed within ±50 kilometers on both axes, where the co-ordinate (0,0) denotes storage. In this test we have kept the number of orders, customers and trucks at a low quantity, since we manually need to check the data output from the prototype.

```
Truck Report
--------------------
Total number of trucks: 2
Truck no.   Size    Time Left (min)     Utilization ratio
0           15      273.9               54.35
1           30      7.9                 98.68
--------------------
Total Utilization ratio:76.52

RoutePipe Report
--------------------
Total number of orders: 30
Complete Cargo quantity:     68.000
```

```
Route no.  Truck no.  Customers   Length (km)   Time (min)   Quantity (kolli)
1          1          14          144.049       332.9        30.000
2          1          11          107.711       259.2        21.000
3          0          9           85.265        212.3        14.000
4          0          4           44.832        113.8        3.000
--------------------
All orders included in route plan.
--------------------
Total Time Used: 1 sec.
```

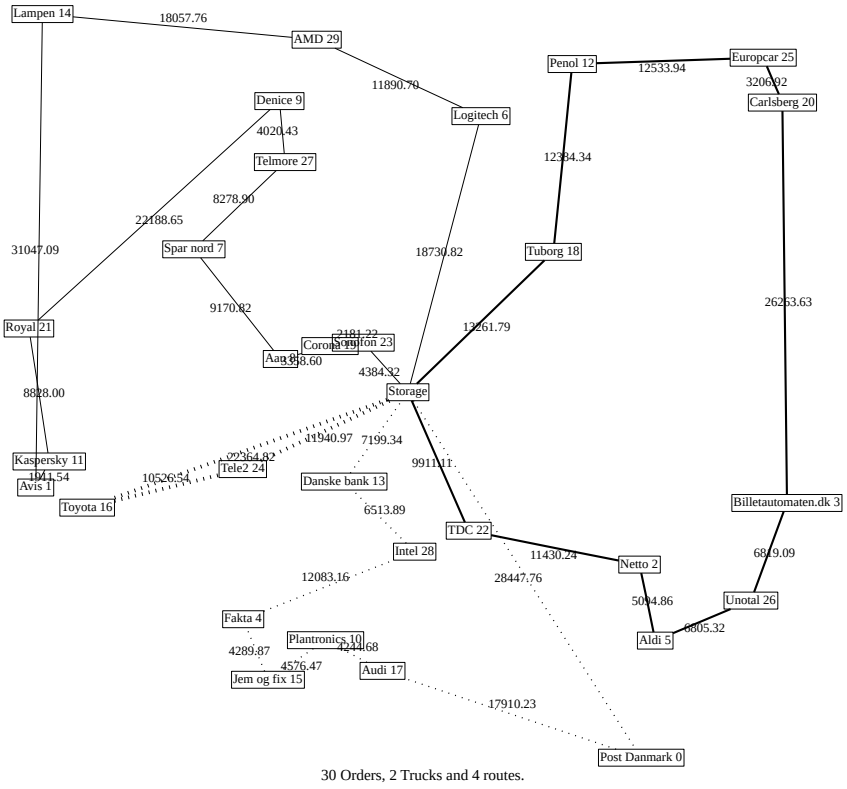


Figure 6.2: The dotted lines represent routes serviced by the small truck and the solid lines represents routes serviced by the large truck.

Testing demand 1 In this test, truck number 0 is a small truck with a capacity of 15 pallets. The other truck is a large truck, with a capacity of 30 pallets. As seen in the RoutePipe Report, none of the routes contains more cargo to be delivered, than the truck assigned to it can handle. This satisfies demand 1 in the demand specification.

Testing demand 2 and 4 Both trucks in the test have serviced two unique routes each, and it is seen that no truck is on two different routes during the same timespan. When truck

number 1 returns from its first route, it is allowed to drive for an additional 267 minutes. The second route assigned to truck 1, has a calculated driving time of 259 minutes, so at the end of the day it has 8 minutes of driving time left. Truck 0 has 273 minutes driving time left, when both routes are completed. This satisfies demand 2 and 4 in the demand specification. None of the two trucks drive more than 10 hours, and as such complies with the simplified driving hour regulations specified in demand 4.

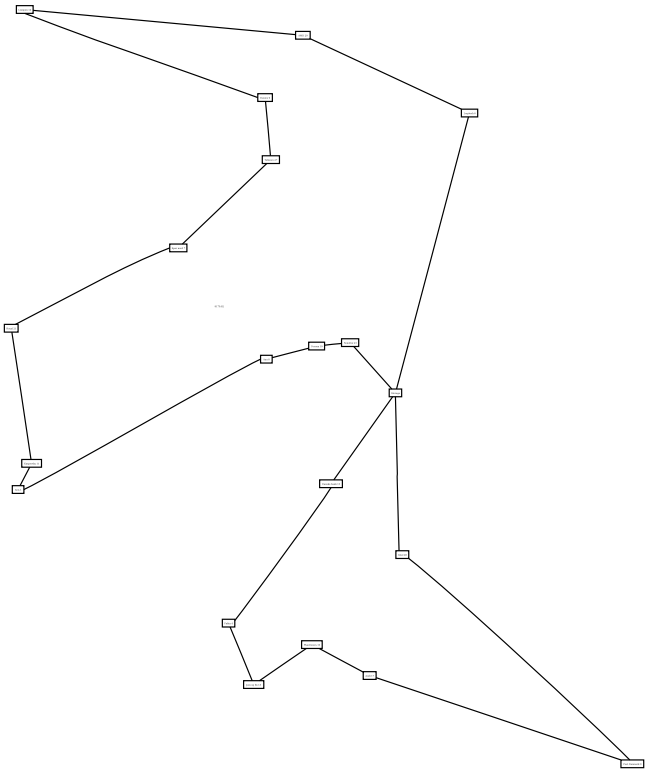


Figure 6.3: Our version of route 1 and route 3, total length reduced with 21 km.

Testing demand 3 To test if demand 3 of the demand specification has been satisfied, we have chosen to manually improve one of the routes made by our prototype. Route number 1 looks, at first glance on the graphical output, as the route that is most in need of optimization. Figure 6.3 shows the result of our manually optimized edition of route 1. The manually laid route is 130 km, while the original route 1 is 144 km. We have also been able to make a optimization of route 3, this route we were able to reduce to 205 km. This only gives a total optimization of approximately 2.28%. According to demand 3 on page 37, the routes laid by the prototype should be minimal, and as this only diviates 2.28% from an optimized route, we belive this satisfy demand 3 on page 37.

The best way of ensuring the route we have found is the shortest possible, would be generating all possible routes and finding the one with the lowest total length. Because of the large amount of possible routes it would be an insurmountable task to do manually[3].

[3]With 13 vertices this amounts to $(13-1)! = 479001600$ possible routes, see section 4.3.5

Therefor we can not ensure that our manually laid route is the shortest possible.

We have not been able to improve the other two routes beyond what the prototype gave as output.

Testing demand 6 In this test there were 30 orders, with a total amount of 68 pallets to be delivered. Our prototype divided the orders into 4 routes, as it can be seen in the Routespipe Report and on figure 6.2 on page 57. Two of the routes have a quantity below 15 pallets; one on 14 pallets and one on 3 pallets. The two other routes had a quantity above 15 pallets; one on 30 pallets and one on 21 pallets. The two routes with a large quantity of pallets, have been assigned to the large truck and the two small routes to the small truck. This satisfies demand 6, since the trucks are assigned routes which fit their capacity.

Testing demand 7 As seen in the Routespipe Report, the first route generated by the prototype, uses the full capacity of the large truck. When we look at the second route, we see the prototype has not been able to use the full capacity of the truck assigned to the route. This is because when the truck has driven the two routes it only has 8 minutes of driving time left. This means our prototype has been forced to remove some of the customers from the route to ensure the truck will not violate demand 4. If we look at route number 3 we can see that it has not used the full capacity of the truck assigned to it. This is probably because the next order, the prototype would have assigned to that route, would overload the truck. This means the prototype has removed one of the customers and closed the route, leaving the rest of the orders for the last route. This leads us to the conclusion that demand 7 has been satisfied.

6.2.2 Time of operation

Demand 8 in the demand specification in section 5.2, requires that a program made based on the flowchart on figure 5.4 in section 5.3, using the algorithms from section 4.3.1, should be able to produce a complete set of routes containing 500 orders during a night. To test whether or not this is possible, we have performed a series of time tests on the prototype.

There is one feature from the flowchart which is not implemented in the prototype. This feature is the inclusion of a subset of the orders, which has a deadline exceeding 24 hours from the current time. The process supposed to do this shown on figure 5.4 on page 40. The implementation of this feature is bound to have a serious effect on the time of operation. It is later seen, that these are somewhat insignificant, when put next to the large differences between the actual time of operation of the prototype, and the limit specified in demand 8.

Method

A series of test are made with the prototype, using an increasing amount of orders and an increasing amount of trucks. The orders are placed within ± 99 kilometers on both axes. The first series of tests are made with 2 trucks, and sets of orders with accumulated order amounts ranging from 10 to 200 in increments of 10 orders. Then a second series is made with 4 trucks and a similar set of orders. These series of tests are made repeatedly, increasing the amount of trucks with 2 between each series, until the final series is made with 20 trucks. The reason for choosing the maximum of 20 trucks, is that previous results has shown us, that 200 orders can be handled with 20 trucks. If we chose a smaller amount of trucks, we might never complete a set of routes with 200 orders, and thus never get the time required

to make this set. The tests are all made on the same computer[4] to get comparable data sets, where the relation between time of operation of the different sets, are not affected by differences in hardware configuration or operation system.

Test results

The graph in figure 6.4 shows the result of the tests as a 3D representation. Judging from this graph, the amount of trucks has no significant effect on the time of operation. The complete dataset is available in appendix C.1 on page 76.

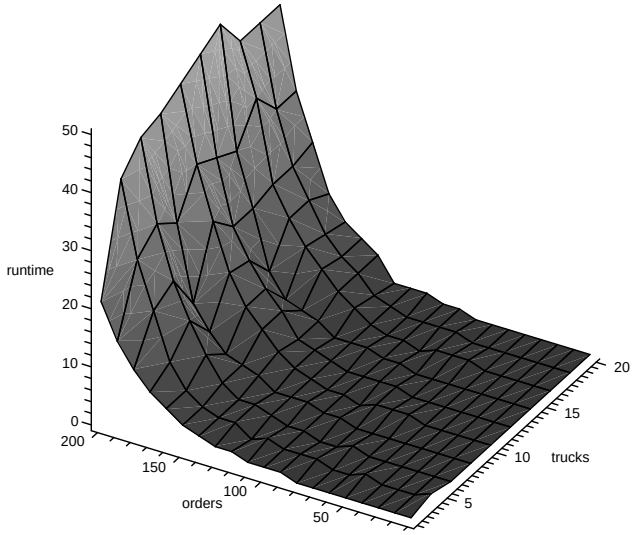


Figure 6.4: The left graph is a 3D representation of the test series from the test in section 6.2.2. It shows the time used in seconds to generate a set of routes for a different sets of orders and a different sets of trucks.

Due to the nature of our extraction algorithm, Christofides' algorithm is used the most number times when making the set of routes for a set of orders, with a set of trucks barely large enough to handle the orders. After first building a route which fills the largest truck, one order at a time is then removed until a truck with enough time to handle the route is found. After each order has been removed, Christofides' is used on the remaining route. This means, that when no truck has time left for anything but very small routes, Christofides' is used almost once for every order in the largest possible route a large truck could handle, without regards for the time factor. This results in the peak shown in the graph in figure 6.5 on page 62. The generation of routes for 200 orders succeeds, when 16 trucks are used. The prototype struggles to fit the 200 orders in small routes in the interval up to 16 trucks, until it finally succeeds at the point of having 16 trucks available. After this point the time to generate the routes seems to flatten, and the addition of extra trucks does not change the time of operation.

The complete set of raw data from the above mentioned tests is available on the appendix CD-ROM in the directory "testdata/timetest/var_trucks_orders". This directory also holds a Maple[5] worksheet, in which the graphs in figures 6.4 and 6.5 can be examined. The relevant

[4] Amilo K7600 2800+ (2.12 Ghz) with 512 MB RAM
[5] An advanced mathematics and computer algebra software package

data has been extracted and can be seen in the table in appendix C.1 on page 76. Based on the results seen in figures 6.4 and 6.5, it seems reasonable to use a very large number of trucks to perform tests, since the number of trucks does not have a significant effect on the time of the operations done. With this in mind the next test is performed with a much larger set of orders on a different computer[6]. This test was made with 100 trucks and two sets of orders: one with 500 orders and one with 1000 orders. The orders are placed within ±99 kilometers on both axes. The set of routes for 500 orders was completed after 17 minutes, and the set of routes for 1000 orders was completed after 269 minutes, or just about 4 and a half hour. The reason to this massive increase for only twice the amount of orders, is the number of operations made by the algorithms increases dramatically, as more orders are added. The complete set of raw data from these two tests are available on the appendix cdrom in the directory "testdata/timetest/large_orderset".

The conclusion is that demand 8, which requires the prototype to be able to build routes for 500 orders within one night, is met. In fact we can do so in 17 minutes. If a whole night was at the disposal, it could build routes for 1000 orders, meaning it can potentially handle twice the amount of orders required.

6.3 Evaluation of the development method

Throughout the coding phase we had strict guidelines, on which method to use during the development. This was done to ensure the program would not "get out of hand", when the code was written in different styles for each coder. Another advantage was that someone always knew how the various functions of the program interacted with each other. It also ensured that all parts of the program worked individually, as they were completed. At the same time everyone knew what the input of each specific input and output was.

As mentioned in the definition of our development method in section 5.1.3 we used pair programming, which was both to ensure a low error rate and to add a learning process to the coding. We sought to pair an experienced programmer with a novice, to avoid beginner errors and avoid common mistakes that can occur if you are used to other programming languages. This generally worked, as most people in the group tried to work both as a driver and a navigator. We shifted programming groups frequently, so nobody worked in the same group throughout the entire process. In general coding alone was not widely used and only debugging and other minor changes were allowed when working alone.

We used a Wiki[7] to keep track of which function or part of the program that was currently under development, and which state of completion[8] they were in. In doing so it was possible to maintain control over the process, and people knew what the next step of the coding process was. This largely removed all standstills in the process. Not only the current state of the different functions were in the Wiki, but also all the function parameters for input and output were listed there. This prevented function specific information from getting outdated or erroneous, because of double or triple documentation. By only having the data in one place, we insured that the code based on the information on the Wiki, would easier co-operate and result in less debugging.

As a conclusion on the development method, we find our combination of XP and Waterfall as an efficient solution. We believe we were able to use the concepts of both methods to balance out the disadvantages of both, and efficiently produce a working prototype.

[6] Ubuntu running on a Xeon 2.8Ghz CPU (Hyper-Threading disabled) with 1 GB RAM

[7] Essentially a web page where users can freely add and edit contents. See "http://www.wiki.org/" for the complete definition

[8] Not started/Started/Broken/Finished

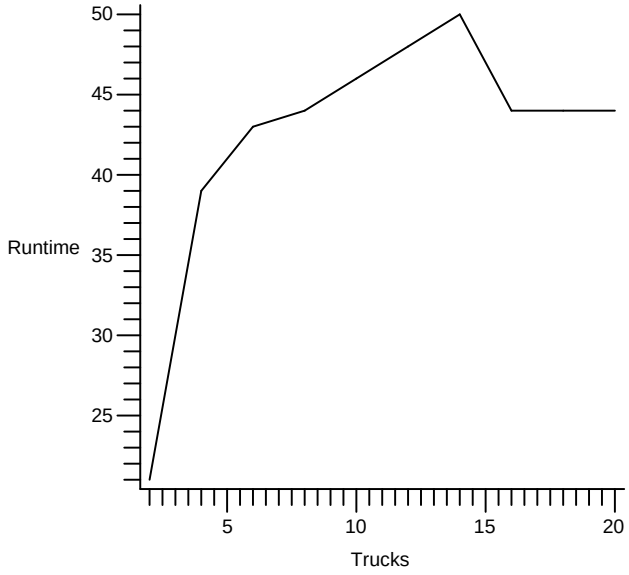


Figure 6.5: The right graph represents parts of the results of the test described in section 6.2.2. It shows the time used in seconds to generate a set of routes for 200 orders using different sets of trucks.

CHAPTER

7

Conclusion

The purpose of this project was identifying and solving specific logistic problems, involved in the delivery process of transporting cargo to the customers. The problems we chose to work with were; reducing the planning time of the routes (problem 1), optimizing the usage of the available trucks (problem 2) and including future deliveries if possible (problem 7).

Through an interview with Daka Bio-industies, we learned that they have had good results, concerning the solution to similar problems, using a software based solution. This lead us to the application of known algorithms and graph theory, which we included in our program design, used to solve TSP related problems. We developed this plan to solve the logistic problems described in our problem formulation in section 3.3.

We implemented a prototype of our program design using the programming language C, to test if the solution met the demands, which has to be met in order to solve the problems. These demands are listed in section 5.2.

The evaluation of our prototype showed, that the demands based on the problems "planning time" and "optimizing truck usage", were met to our satisfaction. Because of complications while implementing the prototype, we were not able to test the demand based on the "48-hour deadlines" problem. Although we have not confirmed, that our method for solving this problem results in a valid solution, it is still a part of our program design.

In the program design we have outlined different methods for route extraction as described in section 5.3.2. We have successfully implemented and tested one of these methods in our prototype, however we can not know if one of the other methods will produce better results.

During the implementation of our prototype, we used our own development method built on Waterfall and XP. By using this method we ensured, that at least two people knew how any given part of the source code worked. This also helped us keep the number of minor errors low, compared to what we have experienced in previous projects.

Based on the results of our test phase, we believe it is plausible that a fully developed software solution, based on our program design, could reduce the workload of a logistics employee.

7.1 Putting the project into perspective

We have now developed a program design, which is based on the problems from an interview with L.C.H. import. This software solution should be able to solve some of L.C.H.s problems, like easing the delivering process by planning routes for the trucks. The question is then; can they use our software solution? How much is missing at the current level of implementation of our prototype, before they can use it in practice? Which influence will such a solution have, if it is commissioned by L.C.H.? These are some of the questions we try to answer in this section.

If we take a look at our prototype, and put it into the perspective of L.C.H., then which parts of the program is ready for implementation and where does it need improvements? By now the program can read a list of input data, available trucks and a set of customers with orders. It would probably be necessary to translate L.C.H.s data, before the prototype can

handle it. Beyond that our prototype could, at its current state, be used as a tool by L.C.H. to help them plan their routes.

If L.C.H. implements our solution in their logistic department, they could save a lot of man hours on the planning process. By implementation, we of course mean a complete developed and programmed solution. The output routes may not necessarily be perfect when they are returned by the program, but one person can quickly get an overview of the routes, one at a time, and do the last improvements, before the truck leaves the storage. This way of validating the routes, is also used at Daka, as described in the third party interview in section 3.4.3.

7.1.1 Future improvements

Though we have a working prototype, there is room for improvement. If we implemented the various methods of extracting subgraphs described in section 5.3.2, we could compare how they perform in relation to each other. The subgraph extraction method the prototype uses on the present stage might not be the best possible, and it would be interesting to see results using other methods. The routes generated using the other subgraph extraction methods would also provide different looking output, which might be usable in different situations.

Remove crossing paths. Demand 3 specified, that the individual routes should be as short as possible. As we mentioned earlier, the prototype has a tendency to create crossing edges in the output graph, which makes the routes longer than necessary. Where the edges collide, resulting in crossing paths, the initial and terminal vertices of the two edges could be interchanged. This should shorten the overall length of the route, because the sum of the length of two opposite sides of a quadrangle are shorter than the sum of the lengths of the two diagonals.

Inclusion of 48 hour deadline orders. Demand 5 specified that customers with a 48 hour deadline should still be considered when generating routes, but currently the prototype can only plan routes for orders with a deadline the following day. The purpose of making it possible to include orders with a 48 hour deadline during the route generation process, is that it is practical to deliver these orders earlier. If the trucks drive in the general area of the customers in the 24 hour deadline plan, they might as well bring out cargo for the customers with 48 hour deadline as well.

Integration with existing software. Besides these additions to the program, there are other things we would like to improve. The method the prototype uses for receiving input orders could be improved, as the input currently is highly dependent on the syntax of the input files to work. Currently the input files are text files, where the data is manually typed in. If the data is typed in wrong, or does not obey the syntax the program fails or provides an invalid output. As L.C.H. currently uses an accounting system to handle their orders, it would be an advantage if we could get the prototype to extract the information it needs directly from this system. If designed properly it would eliminate human errors in the input file, and ensure the input files had the correct syntax.

Proper distances. Currently the distance between customers is calculated as an euclidean distance, and not according to actual distances. The distance between two customers is calculated from the coordinates on the graph, and not the actual time it takes to drive from one customers to the other. It does not take into account the landscape, road systems

and speed limits when calculating the distance, which means the current route results are somewhat inaccurate, if applied to a actual situation. This could be improved by chaining the prototype together with another software application, dedicated to calculating these distances according to actual conditions.

Graphical user interface. At present stage the prototype works entirely through a command console. The output of the data is a text report and a number of images that illustrate the routes made by the software. The output is split up into several files of different types consisting of graphical presentations and text files, meaning a user has to use different programs to view the entire output. Some kind of graphical user interface, where all the output is easily viewable, and the output is easily accessible, would make the program easier to implement in a company. A graphical user interface would also allow easy modification of the input, so an administrator from a centralized position, could make route plans for different storages using the same software. This would make the prototype very similar to what is described in the interview with Daka in section 3.4.3.

Webinterface and forecasts. Another feature could be an opportunity for the logistics employee to optimize routes via a web interface. Then the employee could do it at home, instead of sitting at work all night. A GPS addon which monitors the trucks positions, could tell the logistic manager where each truck is at every given moment, and how much of the route is completed. Another improvement, which could be very useful, is a function which remembers the amount of incoming orders from earlier years, and from these data make a forecast of how many trucks which are needed the following day. Finally an addition to the program could be made, to make it automatically uses the data of a weather forecast to adjust the traveling time of the routes, more accurately.

7.2 Source criticism

We have used a variety of different sources during this project, from which key information has been extracted. These sources form the basis of most of our theoretical considerations, and as such we need to asses these sources in a critical perspective. If we have used non-credible sources, the information used to write the report could be inaccurate or wrong, meaning the base of the entire project is faulty. The following section contains a short assessment of each source referred to in the report.

Indkøbs- og Materiale Styring (logistik) [2]
This source is written by Poul Erik Christiansen, who is an expert within the field of logistics. The book is directed at students of specialized technological (or technical) studies[1] as an educational textbook. The publishing house is "Erhvervsskolernes forlag", which is a publishing house related to a well respected educational institution. On this background we believe the information in the book is credible and as such can be safely used.

En mosaik af dansk logistikforskning [11]
This source is a compilation consisting of contributions from a number of scientists from Aalborg Universitet. The publishing house is Aalborg Universitetsforlag, which is a publishing house related to Aalborg Universitet. The book was written as part of a research project called 'Fremtidens forsyningskæde', and as such we believe the information provided within

[1] In Danish the education is called "teknonom"

the source is credible for use in our project.

Håndbog i Evaluering [6]
This source is a handbook in information gathering and evaluation, written as a collaboration of three different writers. We only had access to an excerpt of the book, and as such cannot evaluate the entire content of it. The writers are all of academic background, and therefor we evaluate the content of the publication is credible. On this background we believe we can use the information in this publication as valid information.

The Traveling Salesman: Computational Solutions for TSP Applications [4]
This source is a handbook for constructing computational solutions for TSP applications. The book is written by Gerhard Reinelt, who is an affiliate of "Institut für Angewandte Mathematik, Universität Heidelberg". The publishing house "Springer-Verlag" mainly publishes handbooks for various engineering crafts. On this background we deem the information used from this sources is valid and relevant for our project.

Discrete Mathmatics and It's Applications, fifth edition [7]
This source is used as an educational handbook in our courses of discrete mathematics and as such we regard this information as credible.

Politiet i Kolding: Bilag 3 til politi rapporten om ulykken i Seest [3]
This source is part of the official police report about the accident i Seest. It is made by the police in Kolding, which was leading the investigation of the accident and as such we believe the information provided within is credible for use in our project.

Extreme Software Engineering: A Hands-On Approach [9]
This book is written by Daniel H. Steinberg and Daniel W. Palmer, as a handbook for a software engineering course. It is used as source for description of XP in our development method section. Daniel H. Steinberg has published several books on the subject of teaching and working with eXtreme Programming, hence we believe this to be a credible source.

Human Aspects of Software Engineering [10]
This source is a book written for software developers, practitioners as well as students. It aims to increase software development team members insight in the human aspects of software engineering. The book is written by James E. Tomayko (D.A., Carnegie Mellon University) and associate Professor Orit Hazzan of Israel Institute of Technology, whom we believe to be a credible source.

W.W. Royce. Managing the development of large software systems [8]
This source is a paper written by Dr. Winston W. Royce (1929 - 1995), former director of Lockheed Software Technology Center. We found his paper on several websites, most of them was websites of different universities. On this background we believe the information in this paper is credible and as such can be safely used.

Retsinfo: Lov om ændring af lov om fyrværkeri og beredskabsloven [5]
Retsinfo.dk is the danish states online legal information system. On this background we believe the information on this website is credible and as such can be safely used.

BIBLIOGRAPHY

[1] Daka Bio-industries. `http://www.dakabio-industries.dk/`.

[2] Poul Erik Christiansen. *Indkøbs- og Materiale Styring (logistik)*. Erhvervsskolerns forlag, 1998. ISBN 87-7881-117-1.

[3] Politiet i Kolding. Bilag 3 til politi rapporten om ulykken i seest. `www.politi.dk/NR/rdonlyres/77AEA27B-57D3-4F28-BE65-46A8CC7C439C/0/Kolding_Bilag_3_Kronologisk_uddrag_af_haendelsesforlobet.pdf` D.20-03-2006.

[4] Gerhard Reinelt. *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer-Verlag, 1994. ISBN 3-540-58334-3.

[5] Retsinfo. Lov om ændring af lov om fyrværkeri og beredskabsloven. `www.retsinfo.dk/_GETDOCI_/ACCN/A20050106030-REGL` D.11-03-2006.

[6] Mikael Togeby; Jill Mehlbye; Olaf Rieper. *Håndbog i Evaluering*. AKF Forlaget, 1993. ISBN 87-7509-340-9.

[7] Kenneth H. Rosen. *Discrete Mathmatics and It's Applications, fifth edition*. McGraw-Hill, 2003. ISBN 0-07-119881-4.

[8] W.W. Royce. Managing the development of large software systems. `http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf`(pdf) D.02-05-2006.

[9] Daniel H. Steinberg and Daniel W. Palmer. *Extreme Software Engineering: A Hands-On Approach*. Pearson/Prentice Hall, 2004. ISBN 0-13-047381-2.

[10] Jim Tomayko and Orit Hazzan. *Human Aspects of Software Engineering*. Charles River Media, 2004. ISBN 1-58450-313-0.

[11] Aalborg universitet. *En mosaik af dansk logistikforskning*. Aalborg universitetsforlag, 2002. ISBN 87-7307-659-7.

APPENDIX

A

C functions

A.1 AddEdgeToEdgeSet function in C

```
1  void AddEdgeToEdgeSet( edge *Edge, edgeset *Set )
2  {
3      /* For safekeeping of the edges while reallocating */
4      edge **Buffer;
5
6      /* Iterator */
7      int i;
8
9      /* Boolean that decides if Edge should bee added */
10     int Addition = 1;
11
12     if( !Set->MultiSet )
13     {
14         i = 0;
15
16         /* look for the edge in the set */
17         while( i < Set->Cardinality )
18         {
19             /* compare the edges in an undirected set */
20             if( CompareEdges( (*Edge), (*Set->Edges[i]), 0 ) )
21             {
22                 /* prevent the edge from being appended, if it exists */
23                 Addition = 0;
24                 break;
25             }
26             i += 1;
27         }
28     }
29
30     if ( Addition )
31     {
32         /* Allocate memory at size of old EdgeSet */
33         Buffer = (edge**) malloc( sizeof(edge*)*Set->Cardinality );
34
35         i = 0;
36
37         /* Save the edges */
38         while (i< Set->Cardinality)
39         {
40             /* Copy edge pointers to buffer */
```

```
41              Buffer[i] = Set->Edges[i];
42              i += 1;
43          }
44
45          /* Free old EgdeSet memory */
46          free( Set->Edges );
47
48          /* Allocate memory for new EdgeSet with room for one more edge */
49          Set->Edges = (edge**) malloc( sizeof(edge*)* (Set->Cardinality + 1) );
50
51          /* Copy buffer to new EdgeSet */
52          i = 0;
53          while (i< Set->Cardinality)
54          {
55              /* Copy edge pointers back from buffer */
56              Set->Edges[i] = Buffer[i];
57              i += 1;
58          }
59
60          /* Append edge to new edgeset*/
61          Set->Edges[i] = Edge;
62
63          /* Increase cardinality on edgeset */
64          Set->Cardinality += 1;
65
66          /* Free buffer memory */
67          free(Buffer);
68      }
69  }
```

A.2 EuclidianWeight function in C

```
1  double EuclidianWeight(edge *E)
2  {
3      double Output;          /* The output */
4      int X0, X1, Y0, Y1;     /* The coord's of the customers */
5
6      if( E->Weight != 0.0)   /* If the edge weight is set, use that */
7      {
8          Output = E->Weight;
9      }
10     else
11     {
12         /*  Fetch coords from the customer belonging to the order of the initial
13             vertex of the edge */
14         X0 = E->Init->Order->Customer->X;
15         Y0 = E->Init->Order->Customer->Y;
16
17         /*  Fetch coords from the customer belonging to the order of the
18             terminal vertex of the edge */
19         X1 = E->Term->Order->Customer->X;
20         Y1 = E->Term->Order->Customer->Y;
21
22         /* The Euclidian distance  */
23         Output = sqrt( pow((double)(X0-X1), 2) + pow((double)(Y0-Y1), 2) );
24     }
25     return Output;
26 }
```

A.3 MinEdge function in C

```
1  edge *MinEdge( edgeset Set, double (*Function)(edge*) )
2  {
3      edge *MinEdge;  /* min fundne edge */
4      int i = 1;          /* iterator */
5
6      MinEdge = Set.Edges[0]; /*  start med første edge i edgesættet,
7                                  vi skal have noget at sammenligne med */
8
9      /* sammenlign med alle efterfølgende edges */
10     while( i < Set.Cardinality )
11     {
12         /* hvis den nye edge er "mindre" end den gamle */
13         if( (*Function)(MinEdge) > (*Function)( Set.Edges[i] ) )
14         {
15             MinEdge = Set.Edges[i];
16         }
17         i++;
18     }
19     return MinEdge;
20 }
```

A.4 Adjacent function in C

```
1  int Adj( vertex A, vertex B, edgeset Edges )
2  {
3      int i = 0;       /* iterator */
4
5      /* For easy reading we use these to store the vertices of the current
6          edge */
7      vertex *Init, *Term;
8
9      /* Run through the edges of the set */
10     while( i < Edges.Cardinality )
11     {
12         Init = Edges.Edges[i]->Init;
13         Term = Edges.Edges[i]->Term;
14
15         /* Comparisson by address of order */
16         if(
17             (Init->Order == A.Order && Term->Order == B.Order)
18             || (Init->Order == B.Order && Term->Order == A.Order)
19         )
20         {
21             /* A and B are adjacent */
22             return 1;
23         }
24         i += 1;
25     }
26
27     /* If we get this far A and B arent adjacent */
28     return 0;
29 }
```

A.5 Prims algorithm function in C

```
1  void Prims( graph *Graph, graph *MinimumSpanningTree, vertex *StartVertex )
2  {
3      /*  Note that in this implementation A is an edgeset,
4              it's simply more convenient */
5      edgeset EdgeSetBuffer;
6      edgeset T;                  /* Edges for the MST */
7      vertexset U;                /* Vertices for the MST */
8      edge *ShortEdge;      /* Buffer for the minimum weighted edge */
9      int i;  U.Cardinality = 0;
10     U.Vertices = (vertex **)malloc(1);
11     U.MultiSet = 0;
12     EdgeSetBuffer.MultiSet = 0;
13     EdgeSetBuffer.Cardinality = 0;
14
15     /* malloc something for the first addvertex to clear */
16     EdgeSetBuffer.Edges = (edge**) malloc(1);
17
18     T.MultiSet = 0;
19     T.Cardinality = 0;
20
21     /* malloc something for the first addvertex to clear */
22     T.Edges = (edge**) malloc(1);
23
24     U.MultiSet = 0;
25     U.Cardinality = 0;
26
27     /* malloc something for the first addvertex to clear */
28     U.Vertices = (vertex**) malloc(1);
29
30     /* Find all edges containing StartVertes and add them to A */
31     i = 0;
32     while( i < Graph->EdgeSet.Cardinality )
33     {
34         /* Add the edge if it contains StartVertex */
35         if(
36             CompareVertices( (*Graph->EdgeSet.Edges[i]->Init) , (*StartVertex) )
37             ||
38             CompareVertices( (*Graph->EdgeSet.Edges[i]->Term) , (*StartVertex) )
39         )
40         {
41             AddEdgeToEdgeSet( Graph->EdgeSet.Edges[i], &EdgeSetBuffer );
42         }
43         i += 1;
44     }
45
46     /* Find the shortest edge with StartVertex in it */
47     ShortEdge = MinEdge( EdgeSetBuffer, Graph->WeightFunction );
```

```
48
49      /* Add the shortest edge to the tree */
50      AddEdgeToEdgeSet( ShortEdge, &T );
51
52      /* Add the vertices to the tree */
53      AddVertexToVertexSet( ShortEdge->Init, &U );
54      AddVertexToVertexSet( ShortEdge->Term, &U );
55
56      /* While U != Graph->VertexSet */
57      while( U.Cardinality < Graph->VertexSet.Cardinality )
58      {
59          /* Find the shortest edge which connects the "already added"
60                  vertices with the "not added yet" vertices */
61          EdgeSetBuffer.Cardinality = 0;  /* Empty the edgeset buffer */
62          i = 0;
63          while( i < Graph->EdgeSet.Cardinality )
64          {
65              /* Add the edge if it contains a vertex in U and a vertex not
66                      in U */
67              if(
68                  (
69                      1 == VertexInVertexSet( Graph->EdgeSet.Edges[i]->Init, &U )
70                      &&
71                      0 == VertexInVertexSet( Graph->EdgeSet.Edges[i]->Term, &U )
72                  )||(
73                      0 == VertexInVertexSet( Graph->EdgeSet.Edges[i]->Init, &U )
74                      &&
75                      1 == VertexInVertexSet( Graph->EdgeSet.Edges[i]->Term, &U )
76                  )
77              )
78              {
79                  AddEdgeToEdgeSet( Graph->EdgeSet.Edges[i], &EdgeSetBuffer );
80              }
81              i += 1;
82          }
83          /* Note: At this point EdgeSetBuffer contains all edges connecting
84                  the already contructed Tree with the rest of the graph */
85          ShortEdge = MinEdge( EdgeSetBuffer, Graph->WeightFunction );
86
87          /* Add the edge and the vertices to the tree */
88          AddEdgeToEdgeSet( ShortEdge, &T );
89          AddVertexToVertexSet( ShortEdge->Init, &U );
90          AddVertexToVertexSet( ShortEdge->Term, &U );
91      }
92
93      /* Build the MST */
94      MinimumSpanningTree->VertexSet = U;
95      MinimumSpanningTree->EdgeSet = T;
96      MinimumSpanningTree->WeightFunction = Graph->WeightFunction;
97  }
```

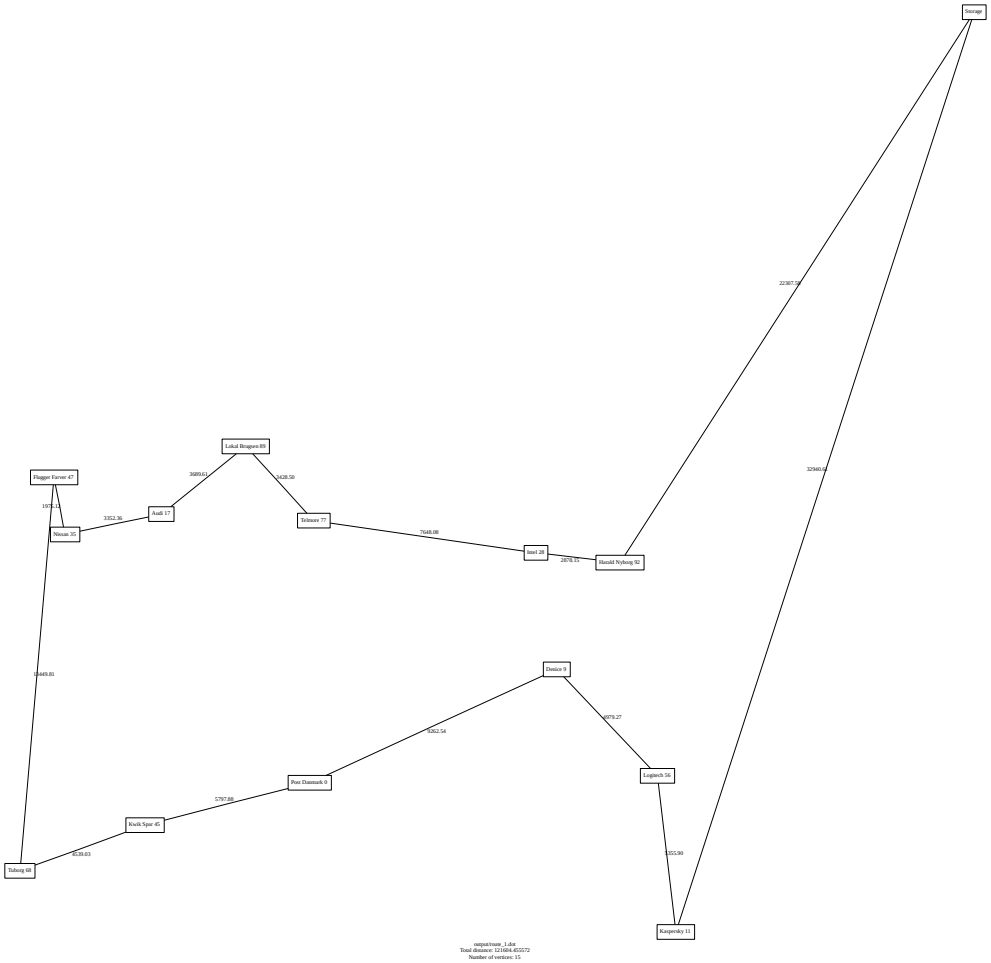
APPENDIX

B

Output examples

B.1 Route example

An graphical example of a route produced by the prototype.



APPENDIX

C

Test results

C.1 Results of the time test

The top row, in the table below, indicates the number of trucks available to the test and the leftmost column lists the number of orders. The numbers in between, is the time, in seconds, it took the prototype to complete the test, with the given input.

Orders / Trucks	2	4	6	8	10	12	14	16	18	20
10	0	1	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0	0	0
50	0	1	0	0	0	0	0	0	0	0
60	0	0	0	0	1	0	0	1	0	0
70	0	0	1	0	1	0	0	1	0	0
80	0	1	0	0	1	1	0	1	1	0
90	1	1	0	1	0	1	1	0	1	1
100	1	1	1	1	1	1	1	1	1	1
110	1	2	1	2	1	1	2	1	1	2
120	2	1	2	2	2	2	2	2	2	2
130	2	3	3	2	3	3	2	4	3	2
140	3	3	3	4	5	4	4	6	6	6
150	4	4	5	5	8	8	5	9	8	8
160	6	6	9	6	11	11	6	12	11	10
170	8	10	14	9	16	15	10	16	16	14
180	11	18	20	13	24	20	20	21	21	22
190	15	26	29	26	33	31	29	35	30	30
200	21	39	43	44	46	48	50	44	44	44