Multi-robot parcel sorting systems

Allocation and path finding

Bram van den Heuvel

Bachelor thesis

Multi-robot parcel sorting systems

Allocation and path finding

by

Bram van den Heuvel

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 26, 2018 at 14:00.

Student number:	4522974	
Project duration:	March 28, 2018 – July 26, 2018	
Thesis committee:	Dr. ir. L. J. J. van Iersel,	Delft University of Technology, supervisor
	Dr. C. Kraaikamp,	Delft University of Technology
	Drs. E. M. van Elderen,	Delft University of Technology

TU Delft

Multi-robot pakket sorteer-systemen

Toewijzing en route planning

door

Bram van den Heuvel

ter verkrijging van de graad van Bachelor of Science
aan de Technische Universiteit Delft,
in het openbaar de verdedigen op donderdag 26 juli om 14:00 uur.

Student nummer:	4522974	
Project duur:	28 maart 2018 – 26 juli 2018	
Thesis comissie:	Dr. ir. L. J. J. van Iersel,	Technische Universiteit Delft, begeleider
	Dr. C. Kraaikamp,	Technische Universiteit Delft
	Drs. E. M. van Elderen,	Technische Universiteit Delft

TU Delft

Preface

This thesis was written towards the completion of my Bachelor of Science degree at Delft University of Technology. During the Applied Mathematics program, discrete optimization has become one of my favorite subjects. It is fundamental, elegant, and applied.

The latter property is one I appreciate, for I prefer to use my mathematical knowledge to solve real-world problems. That is also what I've tried to do during this thesis project. While trying to learn more about the context of my research topic, which is logistics, I met Ronald Schepers. I thank him for selflessly helping me understand his industry better, and for showing me in which ways this project could be closer to real-world applications.

This project was supervised by Leo van Iersel. Our weekly progress meetings were, without exception, simply fun: every time, we dove straight into the solving of puzzles. Every so often, we were joined by Mark Jones, a Ph. D. student of his. I would like to sincerely thank them both.

Bram van den Heuvel
Delft, July 2018

Summary

The logistics industry is being modernized using information technology and robots. This change encompasses a new set of challenges in warehouses. Recently, some companies have started using robot fleets to sort products and parcels. This thesis studies those systems, and researches the combinatorial problems that arise within them.
Three main optimization problems are identified:

1. Finding an optimal layout of the sorting system on the warehouse floor
2. Allocating products or parcels to be sorted to robots
3. Finding paths that all robots can follow concurrently, without colliding.

These problems are considered one by one. The first problem is understood on an intuitive level, while the other two are considered more closely. For both problems, several algorithms are considered. Some utilize greedy heuristics while others model the problem at hand precisely using integer linear programming methods.

The algorithm's real world performance is then assessed using a simulation. Slow, ILP-based algorithms are found to produce optimal solutions for small instances. However, they don't scale well, and are unable to solve large instances. Greedy approximation algorithms solve all problem instance sizes tested, but produce solutions of lower quality.

Contents

1 Introduction

1.1. Logistics is changing

E-commerce has experienced a sustained, high growth in the last years, resulting in a growth of the number of parcels shipped (see Figure 1.1). This growth drives a number of related changes in the logistics industry, which in turn may affect one another. Some of these are captured by economies of scale; higher levels of efficiency are achieved because of improved utilization of resources, for example through improved demand forecasting. This in turn allows a broader range of products to be shipped for lower prices, recursively reinforcing growth.

Combined and among other things, globalization, the Internet, increasing use of information technology systems and automation are leading businesses to restructure their supply chains. Previously, supply chains would often consist of a large number of intermediate wholesalers, retailers and warehouses. These many-step chains are increasingly shortened by large, globally operating companies who account for multiple steps of the classical supply chain and then fuse these steps. Modern fulfillment centers unpack international shipments, temporarily store the products and ship directly to the end customer.

From a business perspective, having a large fulfillment center which avoids the many-step hierarchical distribution of products via wholesalers and retailers has several advantages. The shorter supply chain saves money, as this modern model vertically integrates several steps of the chain, allowing for more direct control of the chain and reduced overhead. Moreover, the shorter supply chain reduces the total time to market, decreases the total value of assets in the chain, and allows for more fine-grained control of supplies. This allows business to more flexibly change existing or introduce new products, while reducing waste.

1.2. A new set of requirements

The modern approach requires parcel handling processes which are quite different from the processes necessary in the hierarchical, many-step supply chain. Where before, facilities would perform few steps or even just a single step in handling a small, known and rarely changing set of products, they now perform many steps

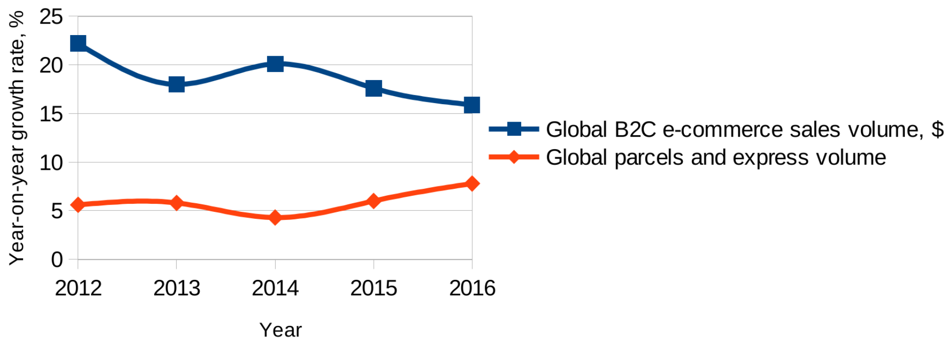


Figure 1.1: Global growth of e-commerce sales volume, from [3]. Global growth of number of parcels and express volume, from [9].

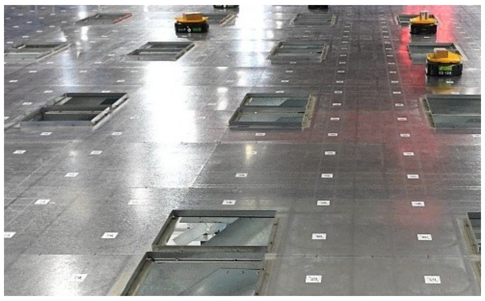


Figure 1.2: Surface area marked with barcodes

on large collections of products, potentially varying greatly in size, weight and handling requirements. These fulfillment centers often operate all day and night, and are heavily automated using information technology tracking products using barcodes.

Modern distribution and fulfillment centers need to perform many complex tasks in a dynamic way. Components of the handling process are [11]:

1. Receive: the intake of products, often arriving via trucks
2. Putaway: the moving of received products to (temporary) storage
3. Replenish: the repletion of products already in storage
4. Pick: the retrieval of products from storage
5. Sort / Pack: the collecting of picked products together and packing for shipment
6. Ship: the moving of sorted and packed products out of the facility.

Often, the picking and sorting phase of the handling process are managed by a single system. This thesis will explore one such system: a multi-robot parcel-sorting system. These systems are flexible, easily scalable, can perform complex tasks, require little human intervention and can as such operate every hour of the day.

1.3. Picking and sorting using robot fleets

A sorting system that uses robot fleets to move items has several components:

- A large surface area, marked with barcodes, for the robots to move around on, see Figure 1.2,
- A fleet of robots moving goods around, see Figure 1.3,
- A system for the loading and unloading of the robots, see Section 1.3.2,
- Charging stations for the robots.

The robots keep track of their location by scanning these barcodes. The area on which the robots move can take on many shapes, see Chapter 3. The robots then move goods around, thereby sorting the goods such that they can thereafter be shipped.

1.3.1. Robot fleets in modern fulfillment centers

There are several advantages to the application of robot fleet systems in modern distribution centers, with limited downsides. We will compare the fleet approach to a more classical sorting solution: that of a static sorting machine.

One clear advantage, is the expandability of the system. If more space is available, the surface area can flexibly be extended by simply placing more barcodes indicating locations. If there is enough surface area for an increased number of robots to function effectively, extra robots be placed into the system. Similarly, a system for the loading and unloading of the robots can be expanded. Likewise, the system could be downsized dynamically and in small increments. Static sorting machines have a fixed capacity and can be extended only to a small extent, they are built once and not designed for modification.

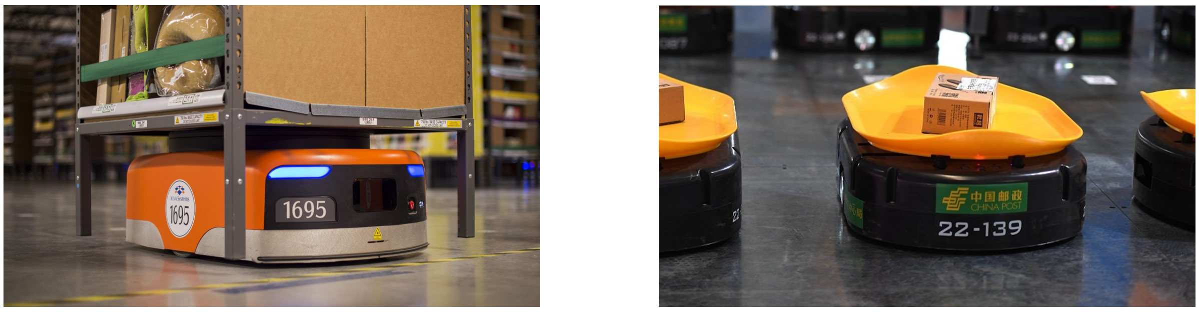

(a) Specialized for lifting heavy weights. (b) Specialized for light sorting.

Figure 1.3: Different robot types

Another advantage is the ability to more easily handle products or parcels of different sizes or weights. While a static sorting machine can handle a fixed range of items, part of a robot fleet can be upgraded to allow the handling of heavier parcels, for example. In this way, fleets can be more easily adapted to a diverse, changing product mix being processed in a facility.

Also in terms of reliability and robustness, robot fleets have an edge on static sorting machines. While the breaking of one or two robots of a fleet might slightly reduce its sorting performance, all functioning robots will still sort packages. A single, static sorting machine will however stop working completely at a malfunction. The predictability of warehouse components is important, and also in this respect one favors the robot fleet solution.

Despite all the above advantages, it depends strongly on warehouse conditions whether a fleet robot sorting solution has the overall advantage.

1.3.2. Storage and picking or picking and sorting

Robot fleets and used in fulfillment centers in two ways, where each application uses a different type of robot. In the first application, robots are used to interact with an enormous number of products or parcels which are stored on shelves in racks, see Figure 1.3a. There will be a large number of racks stored, with a number of robots that is small relative to the size of the space used. The stocking of shelves and picking from shelves happens at a “pick station” to where the robot moves the rack, see Figure 1.4. This robot can lift heavy weights and is expensive compared to the other type, which we discuss next.

The second application has no storage component and is instead used purely for picking and sorting. Small, light and relatively cheap robots carry individual products or parcels, see Figure 1.3b, to one of often many destinations. This applications involves no racks, so the surface area used is often minimized constraint by the sorting capacity the system needs to have, rather than the number of storage racks. Products are placed on the robots by personnel (Figure 1.5a), and after the robots have reached their destination, the parcel is dropped into a bin or hole in which several are collected (Figure 1.5b).

This thesis will investigate robot fleet sorting systems for the second application. While for the first, robots following shortest paths will rarely collide, this is not the case for the second application, where many more robots are active at once.

1.3.3. Available information

Several companies build robot fleet sorting systems. Examples of such companies are Geek+, Hikvision, Amazon Robotics and Tompkins. Information about performance, cost and other properties of these systems is not publicly available, let alone information about their inner working.

Similarly, to the knowledge of the author, no academic literature exists on the topic of non-storage sorting robot fleets, despite several interesting optimization problems that arise within them. While similar problems have been studied extensively, there often are specific requirements for feasibility which are not mentioned in the literature.

1.4. Research objective

This thesis aims to build an understanding of the robot fleet sorting systems without storage components, and the discrete optimization problems that arise within them. This will be done by studying existing systems

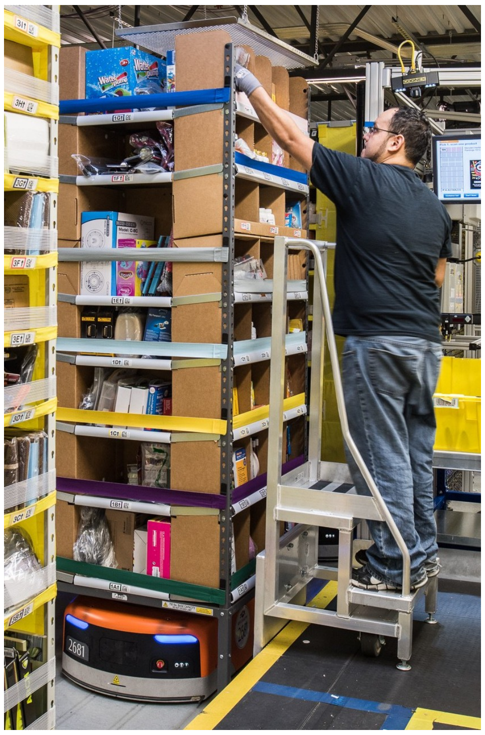
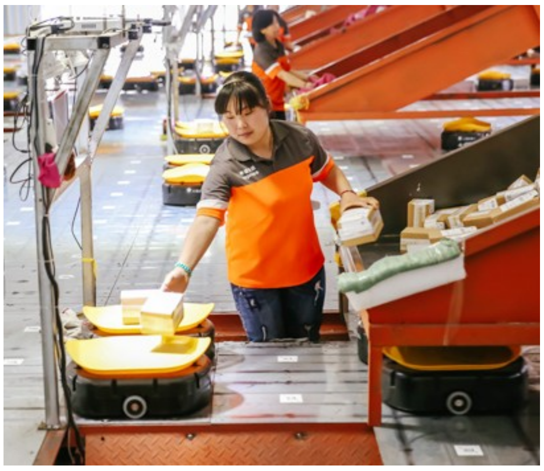
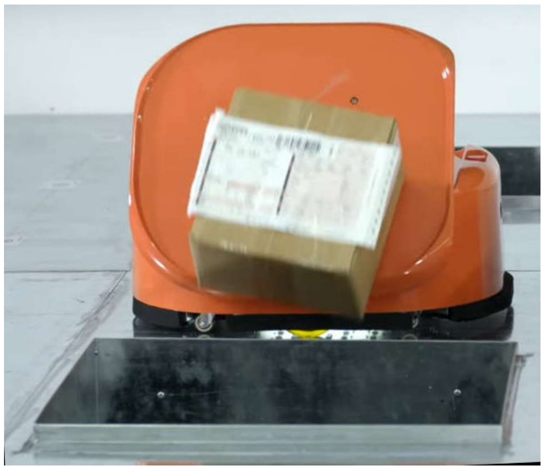


Figure 1.4: Collecting items from a rack at a picking station.



(a) Placing parcels on robots.



(b) Dropping off a parcel.

Figure 1.5: Loading and unloading system for sorting robots

using publicly available information, and by studying existing literature on related topics.

1.5. Outline

This first chapter introduced the topic under consideration. The second chapter will be used to define the substantive problems being considered in this thesis. This includes defining the problem scope, identifying of subproblems and additional requirements for solutions to these problems. The third, fourth and fifth chapters are dedicated to the theoretical treatment of these problems, becoming increasingly thorough in their analysis. The sixth chapter describes the simulation of several models of robot fleet sorting systems and analyzes the performance of several algorithms introduced in earlier chapters. Then, Chapter 7 will reflect on the results of the experiments and conclusions will be drawn. The eighth and final chapter discusses the methods used and identifies open research questions.

2

Scope and problem formulation

The previous chapter demonstrated that the fleet sorting solutions are quite flexible. They can often be adapted to an existing storage or sorting infrastructure, and the possibilities when creating a sorting system from scratch are enormous. In this chapter, we will narrow down the problems we will consider. We do this by first considering various levels of analysis and optimization.

2.1. A naive multilevel formulation

We will formulate the problem as a trilevel optimization problem. While this formulation is far too complex to solve, it helps us identify problems which we can reasonably analyze. The notation introduced will be used throughout this thesis.

2.1.1. Notation

All systems we will analyze will have a number of discrete locations. We denote the collection of these locations by V and the connections between them as $E \subset V \times V$. Because we are working on a plane, one can assume that the graph $G = (V, E)$ is planar. Some of these locations $v \in V$ allow for the dispensing of parcels, others for the removal of parcels from robots. The collections of these locations are denoted by S and T respectively, we sometimes call them the "sources" and "terminals" of G. In this way, we have $S, T \subset V$ while we require that $S \cap T = \emptyset$. Naturally, we must always have $|S|, |T| \geq 1$.

We denote a request from $s \in S$ to $t \in T$ by (s, t), and the collection of requests that a sorting system should address by R. We say that a request (s_i, t_j) is *satisfied* by a path

$$P = (s_i, v_1, \ldots, v_{n-2}, t_j) \in S \times V^{n-2} \times T$$

of length $|P| = n$. Working with discrete time, robots move to any direct neighbor of their current location v in G, where they will then arrive in the next time step. They can stay in the same place if necessary, so not all $v \in P$ need to be distinct.

The robots used in the system are denoted by $[M] := \{1, 2, \ldots, M\}$ for $M \in \mathbb{N}$, with $A : R \to [M] \times \mathbb{N}$ an injective function assigning requests to robots, with a specified order. For each request assigned, we will need to find a collection of satisfying paths with the requirement that no two paths use the same location at the same moment in time. We call such a collection $\mathscr{P}$. We will see in Section 5, that depending on several other choices, we may need more requirements.

Depending on the system, robots can either follow request paths only, or also follow paths connecting the drop-off point of a request by the pick-up point of the next one. We call such paths "return paths". We denote the cost of a collection of paths by $T(\mathscr{P})$. For the first case, we can define T as

$$T(\mathscr{P}) := \max_{r \in [M]} \sum_{P \in \mathscr{P}(r)} |P|,$$

the maximum sum of all paths lengths. Here, the set of paths assigned to robot r by $\mathscr{P}$ is written as $\mathscr{P}(r)$. The latter case requires more cumbersome notation, we will treat it in Chapter 4.

2.1.2. Objective function

From a set of requests R we deduce the sets S and T of start and end locations, these are given. The system to be designed should sort as fast as possible. That is, we want to minimize the latest moment in time the last parcel arrives at its destination, $T(\mathscr{P})$. This requires we have an assignment A for $\mathscr{P}$, which in turn requires we know G. As such, we can write the problem as a trilevel optimization problem in which we try to find the graph $G \ni S, T$ that allows for an assignment A on which a $\mathscr{P}$ is possible such that $T(\mathscr{P})$ is minimal over all possible G, A and $\mathscr{P}$. Or, more concisely:

$$\min_G \min_A \min_{\mathscr{P}} T(\mathscr{P}).$$

2.2. The three levels of optimization

The multilevel optimization problem derived is too complex to analyze. Yet, a clear overview of these levels will help scope this thesis. A layer is identified for each of the subproblems in the trilevel optimization problem. We refer to these as the graph, assignment and path layer from this point onwards.

2.2.1. The graph layer

As briefly mentioned in Section 2.1.1, we assume the system to work on a two-dimensional plane containing discrete locations on which robots can be located. Sometimes, we will call these locations vertices. Between some vertices, robots can move. Some of these are sources or terminals.

The problem of determining the optimal G knows many constraints, some cumbersome to express precisely in mathematical language. It contains the bilevel subproblem of assignment and path finding, of which we will see in chapter 5, that it is intractably hard. As such, precise analysis of this problem is unreasonable. Consequently, the decision was made to focus on the two other problems, and to consider this graph problem mostly out of scope for this thesis.

We would however like to understand some of the choices made in existing systems, as well as justify with a standard of intuitive reasonability the graphs used in further chapters. We will analyze this problem in Chapter 3. Almost all mention of this graph problem will be restricted to that chapter, and this one.

2.2.2. The assignment layer

Now taking the underlying graph G as part of the problem instance, a bilevel optimization problem remains: to find an assignment A of requests in R to robots in $[M]$ such that the total delivery time $T(\mathscr{P})$ for the optimal set of satisfying paths is minimal.

This problem, too, has a subproblem: the finding of optimal paths for a given assignment. After simplifying the subproblem to a simple heuristic, we will notice that this problem is a generalized version of a well-known intractable problem. We must rely on approximation and heuristics to find a reasonably good but suboptimal solution. The analysis takes place in chapter 4.

2.2.3. The path layer

The most inner problem is the minimization of total path time $T(\mathscr{P})$ for a set $\mathscr{P}$ of disjoint paths, taking the graph and assignment as given. There may be extra requirements for the set $\mathscr{P}$, as will be discussed in detail in Section 3.2.

We will study this problem more extensively than the previously introduced ones. Its hardness will be investigated in some detail, and several approximation and heuristics algorithms are provided, all in chapter 5.

Once we have derived several graphs in chapter 3, we will consider a variant of the problem where robots deliver a single parcel and are routed to a new source external from the system in chapter 5. In this case, assignment is not necessary because the first robot available will get the request assigned. We will then adapt this solution for multiple assigned parcels.

2.3. Further requirements

Chapter 1 introduced the potential for robot fleet sorting systems to be adapting, dynamic and fault tolerant. This desired property can be realized only when planning algorithms used can adapt easily and quickly to new information, such as a failing robot or a new request.

2.3.1. Fault tolerance

When a robot fails, one or more physical locations are occupied with a robot that can no longer move. This changes the space that the other robots are allowed to move on. We are especially interested in algorithms that handle this situation well, both on the assignment and path finding layer.

To handle a fault well means that the either the existing solution the algorithm produced can be easily adapted to feasibility for the new situation easily, or that the algorithm can calculate an entirely new assignment and path schedule from scratch sufficiently fast. This means that we are less interested in algorithms that take a long time to compute an initial solution which can't be easily adapted.

2.3.2. Online

Often, not all requests are known in advance. Adaptability of the algorithms used is important in this case. While new requests arrive, they continuously need to be assigned in a way that does not deviate too much from the optimal solution, which could only be computed if all requests were known in advance. Again, we have a special interest in algorithms that have this property.

3
The graph

All of the systems considered in the survey of deployed systems in Section 1.3.3 have their underlying graph structure in common: a square grid graph. This chapter attempts to provide an intuitive understanding of why, under certain logical assumptions, is an obvious choice. Then, we will provide three example graphs to be used for testing assignment and routing algorithms in future chapters.

3.1. Motivation of the square grid graph

Let us reiterate that we restrict our analysis to the two-dimensional case. We start by simply listing various reasonable requirements, from which we will then derive the chosen graph structure.

3.1.1. Requirements

It should be the case that:

1. **The region used should be connected**

   If the system being designed would consist of two separate components, they might as well be two different systems. Without loss of generality, we restrict ourselves to connected systems.

2. **All robots should have the same, convex shape**

   While robots of different shapes might be beneficial in case a set of parcels of varying sizes needs to be sorted, in practice it will rarely be beneficial to use robots of different sizes; this makes it hard to efficiently use the available space, plan routes and design pickup and delivery points. Moreover, production costs for identical robots will be lower because of economies of scale.

   We want the robots to be as small as possible while allowing enough space for the parcels to be carried. The parcels are virtually always rectangular, and as such it is intuitively clear that we should require robots to be convex at least.

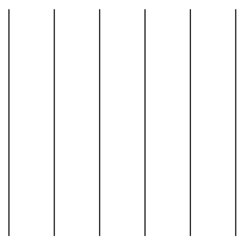
3. **The shape of the robots should be identical to the shape of the locations they occupy** We require the above for dense packing of robots on the graph.

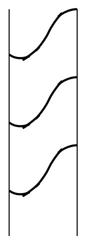
4. **The robots can drive in straight lines while using space optimally**

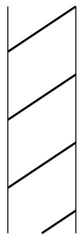
   While covering distances, the robots should be able to drive in straight lines without moving aside, except if another robot is occupying the (entire) space being moved to.

3.1.2. Derivation

Starting with requirement 4, it is clear that we should be able to divide the space on which the robots move in line segments, as depicted in figure 3.1a. There should be multiple robots on these strokes. As they are all


(a) Segmenting the available space in lines


(b) Example of stroke dividing


(c) Example of stroke dividing with straight lines

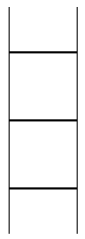

(d) Straight line division with maximal rotations

Figure 3.1: Deriving the square grid graph

shaped identically, we are now able to determine the robot shape by dividing the strokes using identical lines, as depicted in figure 3.1b. The convexity requirement prescribes that two adjacent robots should both have a convex shape, and as such the curve separating the two must be both convex and concave, or a straight line as seen in figure 3.1c. Of all possible stroke divisions using straight lines, we choose the one with lines perpendicular to the moving direction, as it is the only division resulting in a shape with four rotational symmetries, rather than two. As the robots can move in only one direction, this will allow for greater flexibility. See figure 3.1d.

This concludes the motivation for the square block graph, depicted in figure 3.2. We realize that both robots and the locations they occupy should be squares.

3.2. Extra path requirement

Because the robots are about as large as the locations they occupy, we need to formulate an extra requirement for sets of satisfying paths. This requirement will be used in chapter 5.

To see why we need this extra requirement, imagine two robots located next to each other, in the horizontal direction. If the left robot decides to move not further to the left, but to one of the other two possible locations, the robot located on the right can't move to the left directly. It needs to wait until the location where the left robot previously was, is completely empty. For this reason, we require that robots can only move to empty locations.

3.3. Notation

We introduce a vertex for each square location on the plane, and an edge for every two adjacent squares. These form sets V and E respectively. When we refer to "the graph" of a system, we refer to the graph $G = (V, E)$. The graph G is planar.

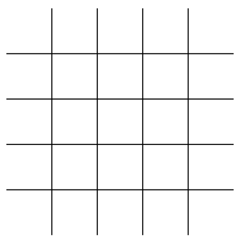


Figure 3.2: Square grid graph

3.4. Examples of graphs

Several scenario's are deemed relevant for testing of algorithms in the chapters to come. One can verify that they all adhere to the requirements outlined in the previous section, and as such use a square grid graph.

One might notice that for all three of the graphs, the terminals outnumber the sources by a substantial margin. The reason is that it is quite simple to mix the parcel streams that need to be sorted, and have them be picked up at the same sources.

The depictions of the graphs below are scaled down for clarity, but will are used in benchmarks with sizes of roughly 3750 locations.

3.4.1. Sources and terminals at the edges

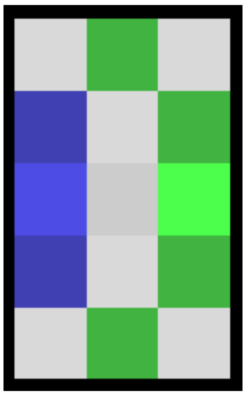
This design, also used by [2], has all sources on the left side with terminals on the top, right and bottom side of a rectangle. See Figure 3.3a for a depiction of this graph.

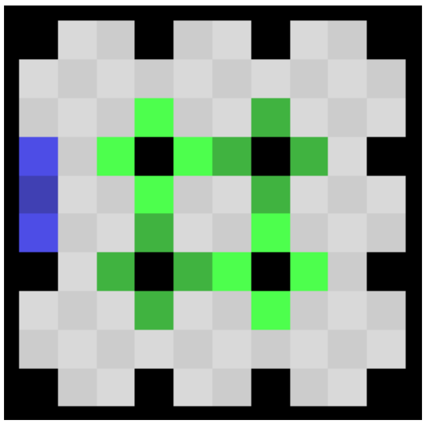
3.4.2. Terminals in the center

When a lower floor is available, one might create holes in the surface used by the robots. This design allows for robots to dispose the parcel into the hole from all four sides, as in figure 1.5b. See figure 3.3b.

3.4.3. Narrow aisles with terminals

When an relatively large number of terminals is needed, a design such as this one might be worth considering. It has all sources on the left side, with terminals on both sides of the a large number of narrow "aisles". See figure 3.3c.


(a) Rectangle with sources and terminals on sides


(b) Rectangle with sources on one side, and terminals in the middle

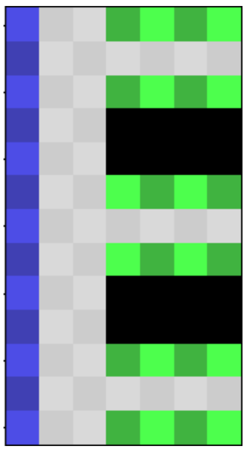

(c) Open space with sources and narrow aisles of terminals

Figure 3.3: Scaled down graph shapes. Sources, or pickup stations, are displayed in blue, while drop-off locations are displayed in green.

4 Assignment

In deciding which robot needs to deliver which parcel, we realize that we have encountered a typical assignment problem. As introduced in Section 2.2.2, we can't solve this assignment problem optimally; in it is contained a subproblem which appears computationally hard, as will be discussed in detail in the next chapter. A heuristic will be introduced, after a precise formulation of the problem.

4.1. Problem formulation

Assuming the graph G as given, we search for an assignment $A : R \to [M] \times \mathbb{N}$ from the set of outstanding requests R to the M robots $[M] = \{1, 2, \ldots, M\}$. The assignment A should result in minimal maximum total path lengths of all robots in $[M]$, so our objective function can be stated as:

$$\min_{A} \min_{\mathscr{P}} T(\mathscr{P}).$$

4.1.1. With or without return paths

As briefly mentioned in Section 2.2.2, there are two variants of robot fleet sorting systems; those where robots complete a request, and then follow a return path outside the graph, and those where robots move only on the graph.

Without return paths

In this variant of the problem, robots are queued outside of the graph, but near the loading points. After a robot is loaded, it follows a path over the graph to the drop-off location for the request. At this location, it leaves the graph. Finally, it requeues near a loading point. Such a system could be implemented with robots returning on another floor, either above or below the surface on which the robots deliver requests[1]

There are two sub variants to this problem; the number of robots available for the system is either limited or unlimited (that is, there are enough robots available to occupy all locations on the graph). Assignment for the unlimited case is trivial; one just assigns each request to a robot available at the relevant loading point. We won't treat this simple solution any further.

For the second sub variant, the number of robots is limited. As such, there may not always be an unoccupied robot to which a request can be assigned. We will use the time (or equivalently, path length) it takes the robot to return to a loading point equal to the length of the request path of the request just served, as a heuristic. A heuristic for the request path is introduced in Section 4.1.2.

Note that there is no cost related to the initial position of the robot, as we assume it to be at the loading point of the next request.

With return paths

For this variant, the robots all move on the graph without leaving it. All robots are located on the graph initially, as such there is a cost to reaching the loading point of the first request. The order in which the requests are served is now important, as there moving from the drop-off point of the previous request to the loading point of the next request takes time.

We denote requests $r_i \in R$ by (r_i^s, r_i^e) for the start and end locations, respectively. We have $r_i^s, r_i^e \in G$.

[1]On this "return floor", a similar path finding problem occurs as on the "delivery floor".

4.1.2. Heuristic for path length

Instead of defining the objective function in terms of the optimal collection of paths $\mathscr{P}$, a heuristic is used. We will use the length of the shortest path between two nodes in the graph, or equivalently, the taxicab metric as a definition of their distance:

$$u = (u_x, u_y), v = (v_x, v_y) \in G : d(u, v) := |u_x - v_x| + |u_y - v_y|.$$

This heuristic is accurate for systems with a single robot, but becomes increasingly inaccurate as the number of robots in the system grows. The probability that robots are able to follow a shortest path decreases. We denote the length of the shortest request path length of request r by $d(r)$.

4.1.3. Objective function

Without return paths

We use the heuristic to estimate the time it will take a robot to complete the assigned request. For convenience, we define $A^{-1} : [M] \to 2^R$ to be the sequence of requests $\{r_1, \ldots, r_l\}$ assigned to r, in order. The total number of requests assigned to r is l. The total duration $t : [M] \to \mathbb{Z}_{\geq 0}$ for robot r is now stated as

$$t(r) = \sum_{k \in A^{-1}(r)} 2d(k^s, k^e), \tag{4.1}$$

the sum of all minimal request and return paths.

With return paths

The initial location or robot r is denoted by s^r. If a specific end location is desired (e.g. a charging station), we denote this location by e^r. This time, the cost of completing request i after request j is given by the distance of the end point of i and the start point of j. We choose to include the request lengths in the term including the start point of the request.

$$t(r) = d(s^r, A^{-1}(r)^s_1) + d(A^{-1}(r)_1) + \sum_{i=1}^{l} \left(d(A^{-1}(r)^s_i, A^{-1}(r)^e_i) + d(A^{-1}(r)_1) \right) + d(A^{-1}(r)^e_l, e^r) \tag{4.2}$$

For both variants, we now attempt to minimize objective function

$$\max_{r \in [M]} t(r). \tag{4.3}$$

A more generalized version of this problem has been studied extensively, and is often called the Pickup-and-Delivery Vehicle Routing Problem (PDVRP), or Pickup and Delivery Problem (PDP). Often, constraints such as time windows for requests and vehicle capacity are included for this problem.

4.2. Exact algorithms

4.2.1. Without return paths

We recognize this problem as a minimal makespan problem with identical machines. Often called the job shop scheduling problem, minimal makespan searches for an assignment of n jobs of varying processing times to m machines. We can view the parcel deliveries as jobs and the robots as the m machines. This problem is NP-hard. It can be solved optimally by dynamic programming, or using an ILP formulation.

4.2.2. With return paths

Offline and exact

We now consider the variant with return paths, firstly its offline version, and realize that this variant is not as straightforward to analyze. We first consider the case when $M = 1$. For this case, we can state the objective function as

$$d(s, a^s_1) + \sum_{i=1}^{l} d(a^s_i, a^e_i) + d(a^e_l, e)$$

where $(a^s_i, a^e_i) := (A^{-1}(1)^s_i, A^{-1}(1)^e_i)$ and $(s, e) := (s^1, e^1)$. We realize that this problem is very similar to the MINIMALHAMPATH problem, where a minimal Hamiltonian path is to be found. It is known that this problem is NP-hard, and as such also the more general PDVRP problem is NP-hard. We set s, e as the start and endpoints of the graph, and consider the requests to be served as vertices which should be on the path. The graph is connected except for the s and t vertices, as every request order is feasible. The cost $c_{i,j}$ is set equal to $d(a_i^e, a_j^s) + d(a_j)$, with start costs being equal to $c_i^s := d(s, a_1^s) + d(a_1)$ and end costs equal to $c_j^e := d(a_j, e)$.

Because the problem is NP-hard for $M = 1$, we conclude that it is NP-hard in the general case with varying M, which is equivalent to a generalized version of the MINIMALHAMPATH problem with M paths, where the graph is non-metric in general.

Exact algorithms for the general PDVRP and PDP problems don't scale well: at most, instances of a few tens of requests have been solved optimally [7]. Some approximation algorithms have been designed, but they are often quite involved. We instead attempt to find a sufficiently fast ILP formulation which is far less general than the general PDVRP problem, and instead formulates the problem at hand specifically.

An ILP formulation

This ILP formulation will often be referenced in the approximation section of this chapter. It uses the following parameters and variables. Note that each feasible solution will always have at least one request assigned to each robot, which for reasonably close start positions and a large number of requests is not limiting.

• $x_{r,i}^s$	for $r \in [M], i \in R$	whether robot r serves request i first
• $x_{i,j}^r$	for $r \in [M], i, j \in R$	whether robot r serves request i before request j
• $x_{r,j,e}^e$	for $r \in [M], j \in R, e \in E$	whether robot r serves request j last and ends at location e
• $c_{r,i}^s$	for $r \in [M], i \in R$	cost of robot r serving request i first, $c_{r,i}^s = d(s^r, k_i^s) + d(k_i)$
• $c_{i,j}$	for $i, j \in R$	cost of serving request i before request j, $c_{i,j} = d(k_i^e, k_j^s) + d(k_j)$
• $c_{j,e}^e$	for $j \in R, e \in E$	cost of serving request j last and ending in location e, $c_{j,e}^e = d(k_j^e, e^r)$

We have the $x^s, x, x^e \in \mathbb{F}_2$. The model uses flow conservation, with other constraints ensuring correctness:

$$\forall r \in [M], \forall i \in R: \quad x_{r,i}^s + \sum_{j\in R} x_{j,i}^r - \sum_{j\in R} x_{i,j}^r - \sum_{e\in E} x_{r,i,e}^e = 0 \tag{4.4}$$

$$\forall r \in [M], \forall i \in R: \quad x_{i,i}^r = 0 \tag{4.5}$$

$$\forall j \in R: \quad \sum_{r\in[M]} \left(x_{r,j}^s + \sum_{i\in R} x_{i,j}^r \right) = 1 \tag{4.6}$$

$$\forall r \in [M]: \quad \sum_{i\in R} x_{r,i}^s = 1 \tag{4.7}$$

Constraints 4.4 ensure flow conservation, while constraints 4.5 ensure that the same request won't served twice consecutively. Constraints 4.6 require that all requests are served at exactly once, and constraints 4.7 require that each robots has a first request to serve.

As the above constraints do not yet avoid cycles, such as, for example, $x_{i,j}^r = x_{j,i}^r = 1$, we generalize the node position variable technique often used in TSP, to force all requests assigned to a single robot to belong to the same path. No other subtour elimination techniques were considered due to time limitations.

• T_r	for $r \in [M]$	total number of requests served by robot r
• t_i^r	for $r \in [M], i \in R$	position of request i in the path of robot r

The accompanying constraints are as follows:

$$\forall r \in [M]: \quad T_r = \sum_{i\in R} \sum_{j\in R} x_{i,j}^r + 1 \tag{4.8}$$

$$\forall r \in [M], \forall i, j \in R: \quad t_j^r \geq t_i^r + 1 + (t^r + 1)(x_{i,j}^r - 1) \tag{4.9}$$

$$\forall r \in [M], \forall i \in R: \quad t_i^r \leq (T_r - 1)(x_{r,i}^s + \sum_{j\in R} x_{j,i}^r) \tag{4.10}$$

$$\tag{4.11}$$

where the $T_r \in \mathbb{R}$ and $t_i^r \in \mathbb{Z}_{\geq 0}$. Constraints 4.8 simply count the total number of requests for each robot, while constraints 4.9 ensure that the counters t_i^r are incremented each time a transitions takes place, but not otherwise. Lastly, constraints 4.10 make sure that counter t_i^r is forced to zero in case request i is not assigned to robot r, and is smaller than the total number of requests served by robot r otherwise.

The objective function, which is the total completion duration for all requests, is stated as

$$\max_{r\in[M]} \sum_{i\in R} \left(c^s_{r,i} x^s_{r,i} + \sum_{j\in R} c_{i,j} x^r_{i,j} + \sum_{e\in E} c^e_{i,e} x^e_{r,j,e} \right) \tag{4.12}$$

and should be minimized.

This formulation has $O(\max(M|R|^2, M^2|R|))$ variables and $O(M|R|^2)$ constraints. Performance in practice is assessed in Chapter 6.

4.3. Non-exact algorithms

For large instances, methods solving NP-hard problems exactly will inevitably be impractical to use. As such, we consider formulations which scale better in this section.

4.3.1. Without return paths

Depending on whether all requests are known in advance, we can apply an approximation algorithm. If all requests are known in advance, we can do offline approximation using, for example, the $O(|R|log|R|)$ SortedGreedyLoadBalance which first sorts all requests r by their $d(r^s, r^e)$ distance, and then iterates over all requests once, repeatedly assigning a request to the robot with the lowest total request distance. SortedGreedyLoadBalance has an approximation ratio of $\frac{4}{3}$. If not all requests are known in advance, GreedyLoadBalance can be applied. It is identical to SortedGreedyLoadBalance, but doesn't sort the requests in advance. It has an approximation ratio of 2, and is an $O(|R|)$ algorithm. If the number of requests is sufficiently large and the variance of their lengths is sufficiently small, even a random assignment to each robot with equal probability will suffice in practice, as the differences in total request length among the robots will tend to zero.

4.3.2. With return paths

Using a non-optimal, feasible ILP solution

Because it is trivial to find a feasible solution for the ILP of Section 4.2.2, it can be supplied to an ILP solver. In case the optimal solution has not been found after the maximum time available, the best feasible but non-optimal solution (or more precisely, a feasible solution not proven to be optimal) can be used instead. Often, this solution will be close to the optimal relatively quickly.

Using the exact ILP online

The exact ILP formulation of Section 4.2.2 can also be used in an online fashion. Strategy Replan [6] repeatedly recalculates the optimal solution as more requests arrive, which requires only a minor addition to the objective function. For each robot, the duration to become available should be added to the relevant term. This algorithm has an approximation of $\frac{5}{2}$ [6]. A variation of this algorithm called Ignore requires less computations as it waits until all currently assigned requests are completed, and only then recalculates with all requests available at that moment. Somewhat surprisingly, it too achieves a $\frac{5}{2}$ approximation ratio [6].

When the offline, exact ILP formulation turns out to be impractical to use, the requests can be split in several batches, to be solved independently. This can either be done sequentially, as described above, or in parallel by ignoring the time a robot becomes available and using the ILP formulation of Section 4.2.2 without modification. Parallel computation may be favored as for a large number of batches, the cost of ignoring the availability times tends to zero. Moreover, a large number of machines can be used to compute at once. Again, a $\frac{5}{2}$ approximation ratio can be achieved. The lowerbound for the approximation ratio of online assignment is 2.

Makespan MinHamPath

A heuristic method which is less easily analyzed analytically but potentially very useful, is the following algorithm consisting of two phases. In the first phase, requests are assigned to robots using one of the methods in Section 4.3.1. In the second phase, the request order is calculated for each robot. This can be done using, for example, the ILP formulation of Section 4.2.2 with $M = 1$.

5

Path finding

This chapter will treat the finding of paths as outlined in Section 2.2.3. First, we repeat the problem formulation and introduce a simpler version, suitable for hardness analysis. Then, we will consider two algorithms solving the problem, after which we will integrate the extra requirements formulated in Section 3.2.

5.1. Problem formulation

Provided a graph G and a assignment A, the aim is to find a path of vertices in V for each robot which starts at the robot's initial location and visits all relevant pickup and delivery points, after which it should end at the assigned location. Naturally, the robots shouldn't collide and as such shortest paths are insufficient.

In order to facilitate an investigation of the hardness of the problem, we will first consider the simpler problem, in which for robots $r \in [M]$ concurrent paths need to be found with a simple start- and endvertex $s^r, e^r \in V$.

In Section 2.2.3 the above problem was posed as an optimization problem, minimizing the total time T used for the completion of all paths. We will first consider the decision variant of this problem only: "For some T, does there exist a collection of satisfying paths $\mathscr{P}$?". We refer to this problem as "the satisfying path problem".

5.2. Time graph

In order to facilitate convenient reasoning about the robot's paths over the graph G in discrete time, we introduce the "time graph". This graph consists of many layers, one for each moment in time, and several extra vertices and arcs representing endpoints. Each layer is isomorphic with G, while for each s^r and e^r an extra vertex is introduced, with arcs outgoing towards the same vertex in each of the layers for the s^r, and arcs incoming from the same vertex in each of the layers for the e^r.

We will assume the available time T to be finite and as such, the amount of layers to be finite. We start counting the discrete time at $t = 0$. Formulating the time graph more precisely, we have

• $V_l = \{v_t \mid v \in V, 0 \leq t \leq T\}$	nodes in the layers
• $V_s = \{s^r \mid r \in [M]\}$	nodes from which paths start
• $V_e = \{e^r \mid r \in M\}$	nodes at which paths end
• $E_l = \{(u_t, v_{t+1}) \mid (u,v) \in E, 0 \leq t \leq T-1\} \cup \{(u_t, u_{t+1}) \mid u \in V, 0 \leq t \leq T-1\}$	arcs within the layers
• $E_s = \{(s^r, s^r_t) \mid 0 \leq t \leq T\}$	arcs from the starting points into the layers
• $E_e = \{(e^r_t, e^r) \mid 0 \leq t \leq T\}$	arcs from the layers to the end points

With $V^T = V_l \cup V_s \cup V_e$, $E^T = E_l \cup E_s \cup E_e$ and $G^T = (V^T, E^T)$ we have time graph G^T. The graph is directed and acyclic, which can be seen easily by realizing that the arcs represent the possible move actions by the robots, and that time only moves forward.

Because the $v \in V$ are located on the x-y-plane, it can be convenient to refer the vertices in V^T using x-y-t-coordinates (e.g. "vertex $(3,4,3) \in V^T$").

Paths from s^r to e^r in G^T correspond to paths, potentially containing cycles and repeated nodes, in G. As robots can't be positioned at the same location at the same time, we require that paths through G^T are vertex disjoint. We conclude that it is possible to model the problem of finding a set of satisfying paths which are completed before time T, as the vertex-disjoint path problem in a directed acyclic graph (DAG).

Alternatively, one can rewrite the vertex-disjoint problem to an edge-disjoint problem, by "splitting" each vertex v into vertices v^i and v^o, having all incoming arcs of v point to v^i and all arcs leaving v leave v^o, while introducing a new edge (v^i, v^o). This edge will force the vertex-disjointness.

5.3. Hardness of the vertex-disjoint path problem in DAGs

This section reports on several research directions which were explored in attempting to determine the hardness of the above problem. We prefer a polynomial time algorithm for solving the vertex-disjoint path problem on time graphs. It is however not known whether it exists, for the general case, the vertex-disjoint path problem is NP-hard [1].

5.3.1. Dynamic programming

The vertex disjoint path problem in DAGs can be solved using a "pebbling game", described in detail by the authors in [8]. In short, a pebble is placed in each of the starting points of paths in the graph, after which one-by-one the pebbles are moved along the arcs following a set of rules, disallowing two pebbles to be at the same vertex. The paths pebbles trace, are the paths sought after.

The pebbling game is a dynamic programming approach, which costs $O(|V^T|^M)$ and as such scales poorly for even a small number of robots.

5.3.2. Tree-width

The bound on the runtime of the pebbling approach of the previous section is simple to obtain: there are only $(V^T + 1)^M$ ways of putting M or fewer pebbles on the graph [8]. Potentially, the specific layered structure of the time graph can be exploited to obtain a better bound.

The dynamic programming approach would work better on DAGs of small tree-width. For each layer, the number ways one can put M or fewer pebbles on the graph is bounded by $O(w^M)$, when following the rules as outlined by [8]. Here, w denotes the tree-width of G^T.

As T is often large in practice compared to $|V|$, tree decompositions of which we suspect they are minimal are those where each subset X_i contains one of the couples of adjacent layers of the graph. More precisely, take $\mathscr{X} = \cup_{i=0}^{T-1} X_i$ with $X_i \supseteq \{v_i | v \in V\} \cup \{v_{i+1} | v \in V\}$. While this may appear to be a tree decomposition using which tree-width can be bounded to $w = 2|V|$ and as such the dynamic programming runtime to $O((2|V|)^M)$, this is in fact not the case. For both endpoints of all arcs in E_s and E_e to be included in a single one of the X_i, we require that for $0 \leq i \leq T-1 : V_s \cup V_e \subset X_i$. As such, $w = 2|V| + M$ and the pebbling algorithm has worst-case runtime $O((2|V| + M)^M)$.

5.3.3. Multi commodity integer flow in DAG

Alternatively, we can model the problem as a multi commodity integer flow problem in a DAG: using the same time graph, we introduce the M commodities $[M]$ with source s^i and terminal e^i. All commodities have a demand of 1.

If working with vertex capacities, one can set all vertex capacities to 1, forcing the disjointness at the vertices. If not, replace each $v \in V^T$ by vertices v^i and v^o, where all incoming arcs of v now point to v^i instead, and all arcs leaving v now leave v^o. Finally, introduce an arc (v^i, v^o) with capacity 1. As values for flow are allowed to be integer only, vertex disjointness is guaranteed.

The general multi commodity integer flow problem is NP-hard, even in DAGs [10]. As there are no obvious improvements we can make to this formulation for the satisfying paths problem, no information about the complexity of the problem is gained.

5.3.4. As a linear program

While providing less insight into the problem at hand, a linear program formulation would imply the existence of a polynomial time algorithm.

Several experiments with a linear program formulation were conducted. The formulation used is similar to the binary program of Section 5.4.2, but doesn't contain any integrality constraints. Solutions found using the Simplex method always returned integer solutions for small instances, but never for large instances for which the usage of interior point methods was necessary. The Simplex method doesn't run in polynomial time, while the interior point method does.

Clearly, for the small instances of the linear program tested an optimal solution of integer values exists. While the same is possibly true for large instances, this doesn't imply that a polynomial time algorithm exists. Even if non-integer solutions can be found in polynomial time, it needs to be possible to derive from that solution one that is integer (and if that were true, with the same objective function value).

It is unclear how this could be done in polynomial time. More importantly, this isn't possible in the general case: the vertex-disjoint path problem is NP-hard even when G is planar [5]. If a linear program formulation of the problem exists, it should exploit graph properties specific to this problem.

As all of the above approaches to finding a polynomial time didn't yield results, it seems plausible that the satisfying path problem can't be solved by a polynomial time algorithm. Attempts to prove the hardness of the problem were unsuccessful.

5.4. Approximation

Since no polynomial time algorithms are known, and possibly don't exist, we instead decide to use heuristic algorithms. While the problem was posed in its decision form before, we now consider its optimization variant as we move on to a more practical approach.

5.4.1. Greedy path finding

A first algorithm to be explored is a greedy one. In an arbitrary order, it finds a shortest path for each robot through the time graph; from the first start location, via all request pickup and delivery locations, to the specified end location.

The A^* algorithm is used to find paths, first between the start location and the first request pickup location, then between the first request pickup location and the first request delivery location, etc. The heuristic used for distance between the $v \in V^T$ is the taxicab metric, which is accurate. Since only vertices adjacent to the shortest path are considered, finding a shortest path in G^T using A^* is computationally cheap and of complexity $O(|V^T|)$. The graphs used for testing are close to square, and as such the shortest path has approximately length $O(\sqrt{|V^T|})$ in the worst case.

In order to avoid collisions, this greedy algorithm repeatedly finds paths reaching all locations, and then removes all vertices in that path from the time graph. In this way, any further paths found in the time graph will be disjoint from paths already found.

While this method will prove useful in practice, it lacks important properties. Most importantly, it won't always find a feasible solution. Depending on the graph, several robots which have reached their final destination might block access to a location which yet has to be visited by another robot. This problem can be mitigated by choosing a set of end locations $\{e^r\}$ such that $G \setminus \{e^r\}$ is connected, and sending the robots to the $\{e^r\}$ after delivery of the last assigned parcel.

Additionally, this algorithm is not fault tolerant: when a robot breaks, all paths have to be recalculated. Recalculating however is cheap.

Extra requirement for satisfying paths

Section 3.2 notes the slightly complicating constraint that robots should move to empty locations only. This constraint is easily incorporated into this algorithm. Vertices v_t in G are only considered for paths if also the vertices v_{t-1} and v_{t+1} are available, and instead of removing only the path vertices v_t from the time graph, the algorithm is adapted to also remove the vertices v_{t-1} and v_{t+1}. In this way, one is certain that no other robots will occupy the adjacent locations over which is being moved.

5.4.2. Using an integer linear program

More sophisticatedly, the following method uses an integer linear program to optimize the next several moves. It "looks ahead" T moments in time, and finds the first T nodes of a non-colliding path for each of the robots. It is fault tolerant, as it simply recomputes every step without any further long-term state. The formulation has the following variables and constants:

• $\mathscr{X}$		set of all locations
• $X_r^t \subseteq \mathscr{X}$	for $0 \le t \le T, r \in [M]$	set of locations robot r could reach within t time steps
• $x_{r,i}^t$	for $0 \le t \le T, r \in [M], i \in X_r^t$	whether robot r is at location i at time t
• $c_{r,i}$	for $r \in [M], i \in X_r^T$	distance to goal of robot r at location i
• F		possible movements on G, or $\cup_{\{u,v\}\in E}\{(u,v),(v,u)\} \cup \{(v,v) \in V\}$

As well as the following constraints:

$$0 \le t \le T, \forall r \in [M]: \quad \sum_{i \in X_r^t} x_{r,i}^t = 1 \tag{5.1}$$

$$0 \le t < T, r \in [M], i \in X_r^t: \quad x_{r,i}^t \le \sum_{(i,j) \in F: j \in X_r^{t+1}} x_{r,i}^{t+1} \tag{5.2}$$

$$0 \le t \le T, i \in \mathscr{X}: \quad \sum_{r \in [M]: i \in X_r^t} x_{r,i}^t \le 1 \tag{5.3}$$

Constraints 5.1 ensure that at each moment in time, every robot is located at a position. Constraints 5.2 ensure that if a robot is located somewhere at time t, it will also be located at one of its neighboring nodes at time $t+1$. Lastly, equations 5.3 ensure that no robot occupies the same location.

Note that the number of $x_{r,i}^t$ variables is greatly reduced by defining them over the sets X_r^t, rather than the entire set $\mathscr{X}$. In practice, $\mathscr{X}$ is large (thousands of vertices), while for $T = 1, 2, 3, \ldots$ we have $|X_r^T| \le 5, 13, 25, \ldots$.

Objective function

We want the robots to move closer to their destinations. However, it isn't obvious how exactly costs should be assigned to locations for $T > 1$. It is for example possible to minimize the total distance the robots have to their goals after T steps:

$$\min \sum_{r \in [M]} \sum_{i \in X_{r,i}^T} c_{r,i} x_{r,i}^T.$$

This however doesn't guarantee that the robots will always move forward. Suppose for example that a robot is $T-1$ steps removed from its goal. Then, there is equal cost to waiting 1 time step and moving in the following $T-1$. Clearly, the objective function must involve also the $x_{r,i}^t$ for $t < T$. I have been unable to find an objective function which doesn't have a similar problem for $T > 1$. For $T = 1$, the objective function above behaves as desired: robots move as much towards their destinations as possible within a single time step.

Extra requirements for satisfying paths

As in Section 5.4.1 we need to make sure that robots move only to empty locations. This can be done by introducing the following constraint:

$$0 \le t < T, r \in [M], i \in X_r^t: \quad x_{r,i}^t + \sum_{s \in [M]: s \ne r, i \in X_s^{t+1}} x_{s,i}^{t+1} \le 1. \tag{5.4}$$

6 Experiments

The algorithms in the previous chapter were introduced and analyzed from a theoretical perspective, from which real world performance may deviate substantially. This chapter is devoted to experimentation by simulation of a multi robot parcel sorting system.

First, the requirements for such a simulation are analyzed, after which the technical implementation is discussed and compared with the requirements. Then, all previously mentioned assignment and path finding algorithms of chapters 4 and 5 are put to the test.

6.1. Simulation

6.1.1. Requirements

The experiments should be useful and conducted following scientific norms. In pursuit of that goal, the following concrete requirements for the simulation have been identified, in order of importance:

- **Correctness**
  This most important requirement demands little explanation. If results are incorrect, they are not useful.

- **Representativeness**
  Even if results are correct, they are not necessarily useful: the experiments being conducted should be representative of real-world problems.

- **Reproducibility**
  Being able to easily demonstrate a result is important. As such, any simulation outcomes should be completely reproducible.

- **Software quality**
  The quality quality of the software should be high, for it can be more easily read and used by others, be more easy to debug, and is less likely to crash.

6.1.2. Technical implementation

Programming languages and software used

The great majority of the simulation is implemented using systems programming language Rust. This language has several properties which make it especially suitable for this purpose:

- **Correctness**
  Rust's is memory safe, implying it doesn't allow null pointers, dangling pointers or data races. This helps the programmer to write code without mistakes [4].

- **Performance**
  Rust is a compiled, low-level language with abstractions which are zero-cost. Its speed is comparable to that of C++, and as such it is very fast. An implementation used in the real world would use a programming language of similar speed, and in this way the language choice contributes to the representativeness of results.

- **Software quality**
  The Rust compiler imposes strict requirements on the programmer, which makes “taking shortcuts” less tempting.

The linear programs were formulated using AMPL, a mathematical modeling language. It allows the programmer to write models more concisely, and on a higher level of abstraction which reduces the likelihood of mistakes. The linear program descriptions it generates are used as input for the combinatorial solver Gurobi. At the moment of writing, Gurobi is the fastest linear program solver on the market, although performance can vary significantly between solvers for specific instances.

The programming language Python was used with library MATPLOTLIB to visualize the simulation. A visualization of the simulation allows for visual verification of correctness, to complement verification done in programming logic.

Software design
Within the simulation software, a clean dichotomy has been put in place which separates the algorithm implementations from the simulation. Repeatedly, the simulation presents the latest system state to the algorithm, which then decides on the next state. This back-and-forth between the simulation and the algorithms allows for extensive verification of the logical correctness of instructions provided by the algorithm being tested, as well as ensures new algorithms can be created and tested in identical circumstances as previous ones. Moreover, Rust's typesystem allowed the writing of the simulation in such a way, that static analysis of the software guarantees that the algorithms can not influence the simulation; they can only provide instructions which are then verified by the simulation, and executed if valid.

The simulation and the algorithms implemented are completely, bit-by-bit, reproducible. All pseudorandom number generators can be initialized with seeds, after which their behavior will be completely predictable. This allows the reproduction of correctness results across different machines, but performance might still differ.

All simulation, algorithm and testing code totals approximately 4500 lines. The software is publicly available on GitHub[1].

Hardware
The simulation was ran on a high-end consumer laptop, with an Intel i7-6700HQ processor and 16 GB memory of available memory. The i7-6700HQ is a quadcore processor with hyperthreading, which results in 8 available cores being presented to the operating system. This processor is clocked at 2.8 GHz, with a peak frequency of 3.5 GHz.

Graphs used
For these algorithm benchmarks, the test graphs as mentioned in Section 3.4 are used. These graph shapes and sizes or similar are used in real-world systems [2].

6.2. Assignment algorithms

In this section, performance of the exact ILP formulation of Section 4.2.2, the greedy heuristic method of Section 4.3.1 for the return path case, and the MAKESPAN MINHAMPATH algorithm of Section 4.3.2. All ILP algorithms are supplied with an initial solution, as pro vided by the greedy minimal makespan heuristic.

6.2.1. Benchmark details

All algorithms were allowed to run for an equal amount of time: 60 seconds. Note that this doesn't include the time for the simulation to verify the actions determined by the robots. The graph sizes used are roughly equal in the number of vertices: $|V| \approx 3750$. This size was chosen after [2]. For details on the graphs used, see Section 3.4.

[1]`https://github.com/vandenheuvel/disjoint-path-routing`

6.2.2. The results

Performance indicator

For some problem instances tested, the exact ILP formulation finds an optimal solution, but for others, it doesn't. For this reason, the optimal value is not always known and as such can't be used to compare with.

Instead, we will use a different metric. We divide the objective value T by the number of requests and multiply by the number of robots used, to get the average number of steps required by a robot to complete a request: $\frac{T}{|R|}M$. Finally, we normalize using the average request length, as defined by Section 4.1.2, of the instance $\frac{1}{|R|}\sum_{k\in R} d(k)$ as it can differ significantly with the different graphs. We call this metric $\alpha := \frac{TM}{\sum_{k\in R} d(k)}$, a lower bound for assignment.

If a value is missing, no value is reported. Tests are performed only if the number of requests is greater than the number of robots.

The number of robots is displayed on the horizontal axis, and the number of requests on the vertical axis.

Pickup and delivery on sides **Pickup and delivery in middle** **Narrow delivery paths**

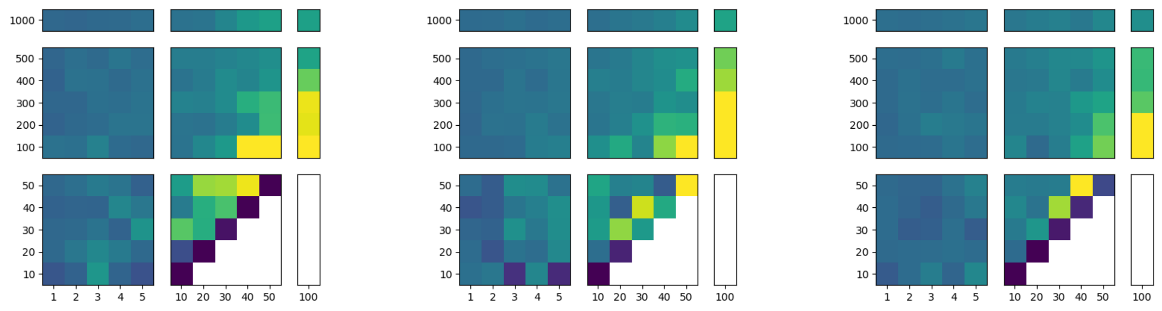
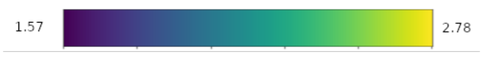


Figure 6.1: Greedy heuristic method



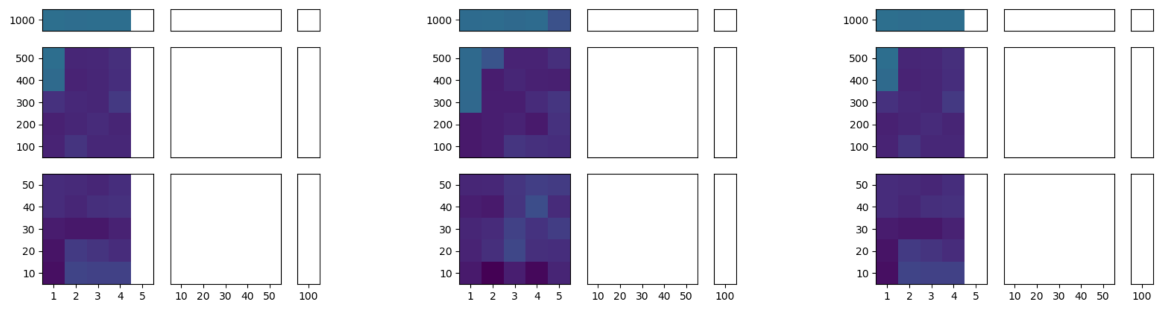


Figure 6.3: MAKESPAN MINHAMPATH

The relative integrality gaps for the exact ILP formulation are displayed in figure 6.6.

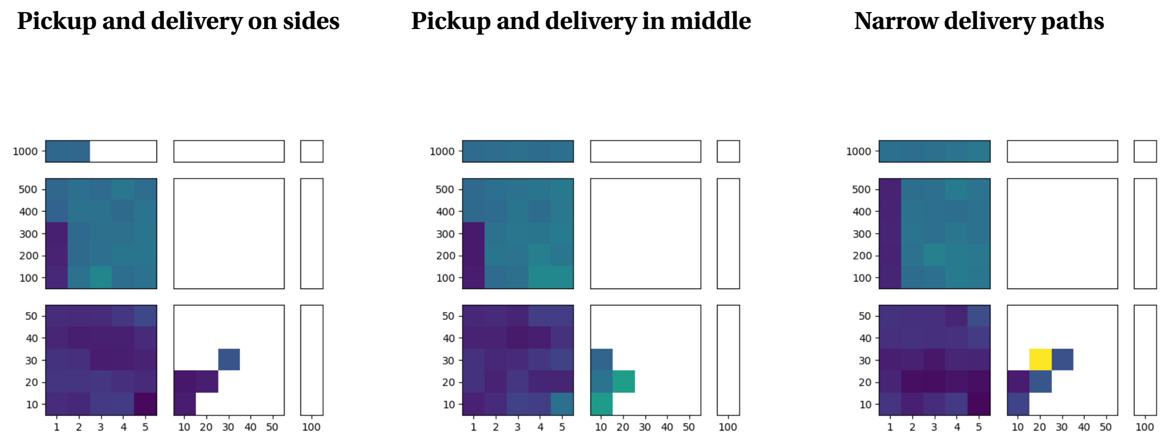


Figure 6.4: Exact ILP formulation

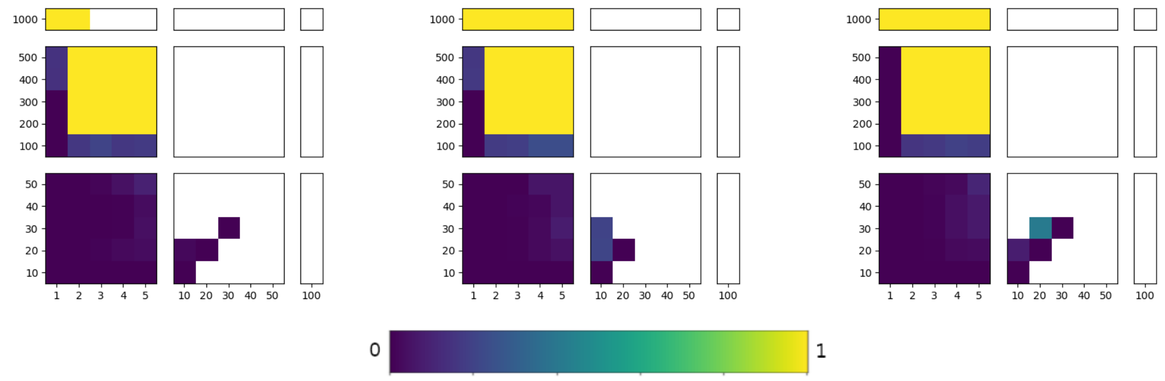


Figure 6.6: Relative integrality gaps for the exact ILP method with undetermined values indicated by yellow

Missing results
When the ILP-based algorithms failed to determine a result, the cause was often an out-of-memory error. Only rarely could the integer linear programs be solved at all on the hardware used, and not yield a solution within the available time.

6.3. Path algorithms

While the number of requests assigned is of importance for assignment, it is irrelevant for path algorithms as they only move towards their next goal. For this reason, tests in this section will all be carried out with a fixed $\frac{|R|}{M}$ ratio.

Both the greedy algorithm as introduced in Section 5.4.1, and the ILP formulation of Section 5.4.2 with $T = 1$ are put to the test.

We take the summed length of all requests and intermediate paths to be satisfied as a reference value, again measured as defined in Section 4.1.2, because the optimal collection of satisfying paths is unknown. We then consider how the total number of steps compares to this lower bound, by computing the ratio between the two. We call this ratio β.

6.3.1. Benchmark details

The algorithms were permitted to run for 1 second per time step. As an initial solution was provided to the ILP-based algorithm, it always found a feasible solution within than time.

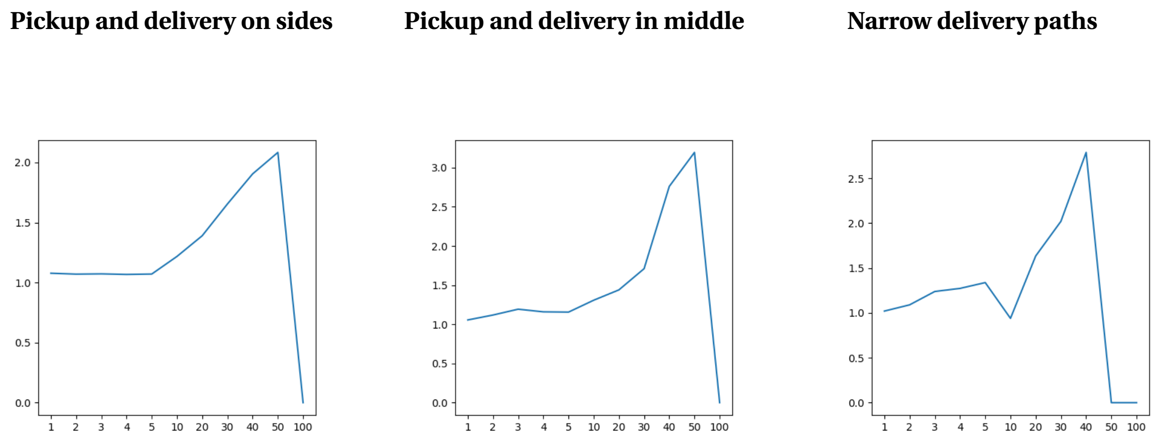


Figure 6.7: Greedy algorithm's ratio β for a varying number of robots

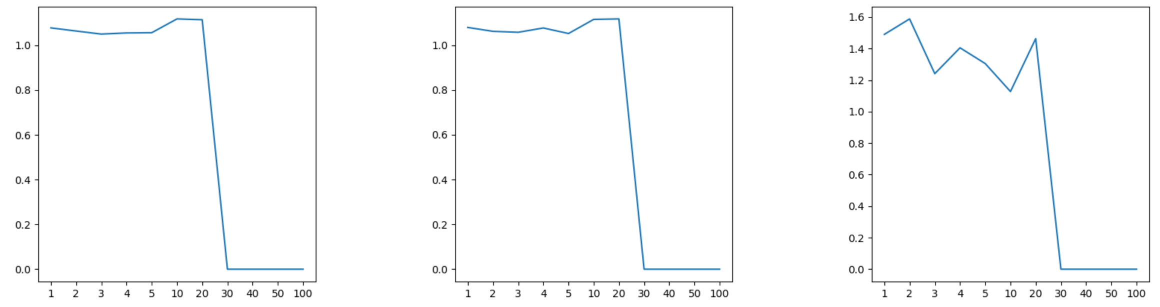


Figure 6.8: ILP-based algorithm's ratio β for a varying number of robots

Missing results
Again, missing results (indicated by zero values) for the ILP-based algorithm were due to a memory error. Missing values for the greedy algorithm were caused by hitting the limit for the number of total time steps in the simulation, which was 5000.

7 Conclusions

From the results of Chapter 6, we will now briefly draw conclusions.

7.1. Assignment algorithms

7.1.1. Feasible solutions

It becomes clear that the ILP-based algorithms don't scale well in regards to the number of robots. Very few problems having 10 or more robots were solved. The algorithms appear to scale significantly better with respect to the number of requests to be assigned. Note that the MAKESPAN MINHAMPATH algorithm isn't able to solve any instances with 10 or more robots. The second phase of the algorithm, which is the expensive phase, can be performed completely in parallel using multiple machines. As the large instances required more memory than was available, this method has the potential to perform significantly better when tested with multiple machines.

The exact ILP formulation has $O(\max(M|R|^2, M^2|R|))$ variables and $O(M|R|^2)$ constraints. As such, the theoretical analysis isn't well reflected in the real-world experiments.

The greedy algorithm is however able to find feasible solutions for all tested sizes of the problem.

7.1.2. Solution quality

The quality of solutions found by the exact ILP formulations is high, as the relative integrality gaps are very low: only rarely above 0.2. With this information, we can compare the values for α found among the different algorithms. As is to be expected, the solution quality goes down as the problem instances grow for the ILP-based algorithms. Noteworthy is the performance of MAKESPAN MINHAMPATH for large numbers of requests, as it often performs better than the exact ILP formulation.

While the greedy algorithm is always able to find a solution, the quality of those solutions is clearly lower. The algorithm especially suffers when the ratio $\frac{|R|}{M}$ is low. This is to be expected: differences in request length won't be "balanced out" as the algorithm assigns more and more requests to the robots. As the previously mentioned ratio grows, we notice that the solution quality approaches $\alpha \approx 1.9$. This only slightly better than a random assignment, which is roughly $\alpha \approx 2.0$.

7.2. Path algorithms

Reviewing the simulation results for path algorithms, we find that the ILP-based algorithm performs as well as the greedy algorithm for small instances, but outperforms the greedy algorithm for instances with a larger number of robots. This is to be expected. The greedy algorithm is however able to handle instances with a larger number of robots, which the ILP-based algorithm can't.

The ILP-based algorithm performed significantly worse on the "narrow delivery paths" map. For $M = 1$, the performance value α should be the same for both the algorithms, but it isn't. Possibly, a mistake was made in program logic.

8 Discussion

8.1. Main limitations of this thesis

The ILP-based path planning algorithm was only experimented with to a limited extent: the algorithm works only for $T = 1$. This is regrettable, as it has the potential to perform well. Sadly however, the time available for this thesis was limited.

The metrics used to assess algorithms' performance could have been examined in more detail. While the chosen metrics are intuitively reasonable, no other measures were explored.

There are numerous improvements that still can be applied to the algorithms. An interesting example is the providing of good initial solutions to the ILP-based algorithms, as this could improve the quality of final solutions. A simple feasible solution that could provided is the one were all robots stay in the same place, but using heuristic methods better initial feasible solutions could be found. Moreover, probabilistic algorithms have not been explored.

The extra path requirement of Section 3.2 is a bit too strict. Instead of requiring that robots move only to empty locations, one could allow for robots to move in the same direction, while being adjacent. As this requires one to incorporate orientation into the model, the stricter requirement was used for simplicity.

8.2. Further research

8.2.1. Model extensions

Robot types

Using different robot models in a single sorting system might be beneficial in case there are different parcel types to be sorted, but the distribution of the parcels over different sorting systems is inefficient. This model expansion requires the introduction of parcel and robot kinds, as well as the compatibilities between the different kinds.

Robot capacities

If efficient pickup and drop-off methods can be designed which allow for multiple parcels to be moved around by a single robot, more efficient and complex pickup and delivery strategies could be employed. In this case, using more complete pickup and delivery problem models are needed. Similarly, time windows could be introduced.

8.2.2. Speed

During this thesis, the robots were assumed to move the same distance in each time interval. More realistically, robots would accelerate and decelerate, and in this way could achieve higher speeds when driving in the same direction for a longer distance.

The speed component appeared fundamentally incompatible with the discrete nature of the already chosen model and for this reason, speed was neglected.

8.2.3. Storage

The focus of this thesis was on sorting, but the combination of sorting and more long term storage is just as interesting. Sometimes, warehouses have far more storage racks than robots, and the robots move around and underneath the storage racks. In that case, algorithms determining which product should be stored in which rack are necessary, as well as different assignment and path routing algorithms.

8.2.4. Battery charging

All robot models discussed in Chapter 1 have a battery inside, powering them for several hours. Simulations over long time spans could incorporate battery levels and charging time.

Bibliography

[1] Sanjeev Khanna & F. Bruce Shepherd Chandra Chekuri. An $o(\sqrt{n})$ Approximation and Iintegrality Gap for Disjoint Paths and Unsplittable Flow. *Theory of Computing*, December 2005.

[2] Francesco Mauro. Towards the design of an effective and robust parcel-sorting system. August 2017.

[3] Zoran Kalinić. *E-commerce in EU and Serbia: Current trends and perspectives.* February 2015. ISBN 978-83-62511-14-3.

[4] The Rust Programming Language. The Rustonomicon. July 2018.

[5] Guyslain Naves and Andras Sebo. Multiflow Feasibility: an Annotated Tableau. *Research Trends in Combinatorial Optimization*, 2009.

[6] Jorg Rambau Norberg Ascheuer, Schen O. Krumke. Online Dial-a-Ride Problems: Minimizing the Completion Time. 2000.

[7] Katya Scheinberg. Vehicle Routing: Problems Methods, and Applications. Second Edition. 2014.

[8] John Hopcroft Steven Fortune and James Wyllie. The Directed Subgraph Homeomorphism Problem. *Theoretical Computer Science*, 1980.

[9] IPC; Various. Year-on-year percentage change in global parcels and express volume from 2012 to 2016, 2017.

[10] Xiaoqian Sun Weibin Dai, Jun Zhang. On solving multi-commodity flow problems: An experimental evaluation. *Chinese Journal of Aeronautics*, October 2016.

[11] Vincent Weinschenk. Analysis, Design & Consultancy of Collaborative, Robotic & Automated Solutions. `www.wherehows.com`, Heidijk 2B, 5251 BM Vlijmen, The Netherlands, February 2018. Copyright by WHEREHOWS Logistic Consultants.