TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Computer Systems

[IASM]
Amina Manafli 195995IASM

APPLYING TSP HEURISTICS TO SOLVE WAREHOUSE ORDER PICKING PROBLEMS

Master's Thesis

Supervisor: Uljana Reinsalu
Researcher
Co-Supervisor: Stefano Fiorenza
Software Engineer, Java mentor

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutisüsteemide instituut

[IASM]
Amina Manafli 195995IASM

PROOVIREISIJA ÜLISANDE HEURISTIKA KASUTAMINE LAOTELLIMUSTE TÄITMISEKS

Magistritöö

Juhendaja:	Uljana Reinsalu
	Teadur
Kaasjuhendaja:	Stefano Fiorenza
	Tarkvarainsener, Java mentor

Tallinn 2021

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, the literature and the work of others have been referenced. This thesis has not been presented for examination anywhere else.

Author: Amina Manafli

2021-05-21

Abstract

In this thesis the warehouse routing problem is addressed in the context of conventional parallel aisle warehouse system with multiple blocks. Exact algorithms for this NP-hard Steiner traveling salesman problem (TSP) exist only for specific warehouse configurations. We investigate whether reformulating and solving the problem as a classical TSP will yield good results using common TSP heuristics and state-of-the-art TSP solvers. To accomplish this, we introduced a preprocessing stage that converts the routing problem into an instance of a traditional TSP. Additionally, we evaluate the influence of the warehouse layout design on the performance of the selected and implemented TSP algorithms.

This thesis is written in English and is 66 pages long, including 6 chapters, 28 figures, and 10 tables.

Table of Contents

List of Figures

List of Tables

1 Introduction

Warehouses are an essential component of any supply chain. Without agreeing on the location, construction, and management of warehouses, no supply chain design and management is complete. Warehouses are also used not only as storage facilities, but also as value-addition facilities. Several warehouses have manufacturing, storage, and maintenance processes on-site. Warehouse decision models are critical to a company's viability. Existing literature indicates that learning warehouse architecture and management concepts will play a critical role in optimizing operational performance.

Warehouse's typical operations include the following processes: receiving, put-away, processing customer orders, order-picking, accumulating and sorting, packing, cross docking, and shipping. Many research papers have been written about these processes, for example a survey by Cormier and Eng [1] and paper by Gu, Goetschalckx, and McGinnis [2] covered considerable number of topics in almost all of the aspects of the warehouse operations, including warehouse design models, picking policies, batching policies, storage assignment policies, and issues about costs of space, leasing, and inventory.

Challenges that warehouses face are continuously changing through the years and require new solutions - supply chains are shortening and becoming more integrated, globalization of the operations is increasing, while customers are more demanding and technology changes are occurring rapidly. To meet these demands, companies are implementing novel approaches such as warehouse management systems (WMS).

Warehouse management systems may be standalone or integrated into an ERP (Enterprise Resource Planning) or supply chain execution suite. The primary function of a WMS is to coordinate the transfer and handling of products within a warehouse. The WMS may be paper-based, RF/wireless-based, or a hybrid of the two.

Table 1. Typical distribution of warehouse operating expenses.

Function	% of annual operating expense
Picking	55%
Shipping	20%
Storage	15%
Receiving	10%

Real-world order-picking system design is often complicated by a variety of external and internal factors that influence design decisions. Goetschalckx and Ashayeri [3] mention such *external factors* as: marketing channels, customer demand pattern, inventory levels, overall demand for a product, and state of the economy. They also describe system characteristics, organisation, and operational policies of order-picking systems as *internal factors*. The organisation and operational policies are: routing, storage, batching, zoning and order release mode (five in total) [4]. In this work we focus on a variety of routing and storage policies, as well as the influence of storage policies on the results of routing process.

Bartholdi, Hackman, and Technology. [5] pointed out that in a typical warehouse management system the order-picking operation cost accounts for 55% of warehouse operating cost (Table 1) and order-picking itself may be further broken down as the following activities in Table 2. It's easily noticeable that travelling comprises the biggest part of the order picking costs, which is itself the most expensive part of the warehouse operating expenses (Table 1).

Based on these figures, we can draw the conclusion that reducing the travel distance when receiving a picking order is a critical problem that deserves careful consideration. In relation to reducing travel time in the picking process, we are concerned with the following question: *how to minimize warehouse routing time?*

Table 2. Activities that compose order picking process.

Activities	Order-picking time (%)
Travel	50%
Search	20%
Pick	15%
Setup	10%
Other	5%

1.1 Problem Formulation and Motivation

There are various ways to decrease handling time, for example by introducing new handling machines to the warehouse or by changing the design of the picking areas. However, less costly and radical methods are commonly available for reducing the handling time, for example through continuous improvement in the operational procedures.

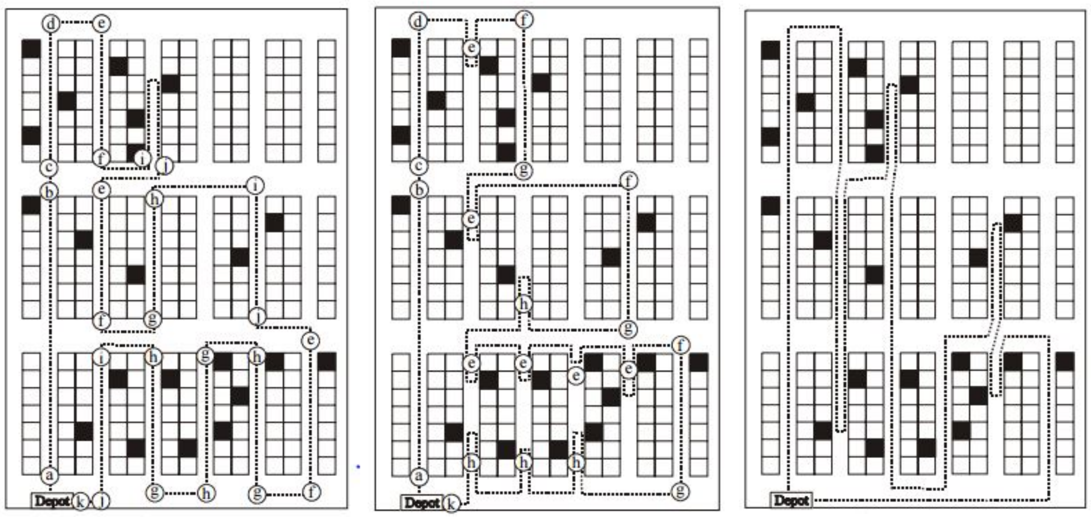
"The productivity of the order picking process depends on factors such as the storage systems, the layout and the control mechanisms" [6]. It can be improved by reducing the time needed for picking an order which, as shown in Table 2 mainly comprises of travelling at around 55% of the total order-picking time. Previous works have also shown that picking and storage strategies are closely interrelated, meaning that decisions on the storage policy have a major influence on order picking performance (Petersen and Schmenner [7] and Petersen and Aase [8]).

In this thesis we limit the scope to systems that implement picker-to-parts order picking method, consist of multiple parallel aisles and blocks, and all orders include more than one product at a time. These systems form the very large majority of picking systems in warehouses worldwide, over 80% of all order-picking systems in Western Europe according to Le-Anh and De Koster [9].

The problem of sequencing and routing order pickers in conventional rectangular warehouse systems with multiple parallel aisles classifies as a *Steiner Traveling Salesman Problem*, a variant of the classical Traveling Salesman Problem which is one of the most important Combinatorial Optimization Problems and belongs to the class of NP-hard problems.

To solve the Steiner TSP means finding a tour of minimum length that contains every non-Steiner node. We can represent the problem by a graph that contains two types of vertices: vertices that must be visited (representing order locations) and Steiner vertices that are not required to be visited and represent the intersections between aisles and cross-aisles. To solve this problem various approaches were proposed by different authors, predominantly focusing on dedicated heuristic algorithms such as S-shape, return, composite, midpoint and largest gap heuristics Roodbergen and Koster [6]. Examples of the three most common of these algorithms are shown in Figure 1.

Another approach used in Theys, Bräysy, Dullaert, *et al.* [10] uses the fact the the Steiner TSP is a special kind of the classical TSP and, therefore, can be reformulated into the latter by recalculating the distances between any two pickup locations and removing Steiner nodes. This work published by Theys et. al. focuses on comparing the state-of-the-art Lin-Kernighan-Helsgaun heuristic to the dedicated heuristics for routing order pickers. In our work, we use a similar approach to test the performance of three common TSP



(a) S-shape. (b) Largest Gap. (c) Aisle by Aisle.

Figure 1. Example of dedicated routing methods [6].

heuristic algorithms and two TSP solving tools in the context of a conventional multi-parallel-aisle warehouse with the goal to determine, how the less sophisticated TSP solving algorithms perform on the Steiner TSP when compared to the optimal and optimized TSP solving tools such as Google OR-Tools and Concorde TSP solver.

This thesis was inspired by the aforementioned paper by Theys, Bräysy, Dullaert, *et al.* [10], as well as my internship with Kuehne + Nagel International AG, a global transport and logistics company whose employees, including my co-supervisor Stefano Fiorenza, were kind enough to provide valuable insights and domain knowledge related to various warehouse design and management aspects necessary for successfully conducting the thesis.

In this work we first examine how three traditional TSP heuristics work on the problem of order picker routing in standard multiple-block warehouses (RQ1). To accomplish this, we adapt the routing and sequencing problem of order pickers in traditional multi-parallel-aisle systems to a classical TSP during the preprocessing stage.

Second, we investigate the impact of warehouse layout and order list size on the outcomes of these heuristic algorithms (RQ2). We also perform a case study on one of the Kuehne + Nagel's existing warehouse configurations and discuss whether our solution could be integrated in such warehouse.

Finally, we examine whether a sophisticated heuristic is needed for routing order pickers in warehouses, or whether an existing tool will suffice to produce optimal and near optimal results.

The implementation and detailed explanation of the preprocessing stage that makes the use of TSP heuristics on the routing issue possible is also considered a contribution of this work.

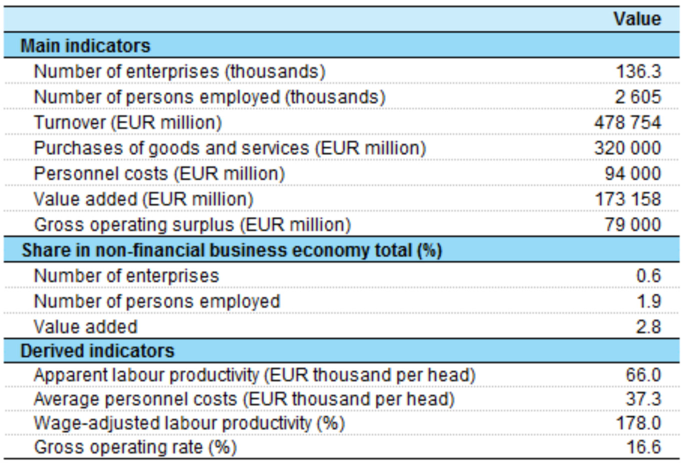
This study starts with a summary of the associated context subjects, such as warehouse operations including picking problem and storage assignment, traveling salesman problem and steiner traveling salesman problem formulations in section 2. Section 3 examines related literature and state-of-the-art papers on the thesis subject. Section 4 is devoted to describing the implementation of the preprocessing stage and TSP algorithms, followed by an experimental results section (4) in which we discuss and compare the simulation results and attempt to answer the research questions described above. In this segment, we also discuss our study on the Kuehne + Nagel warehouse case and assess the applicability of the work done on this case. Finally, we wrap up the report with a review of the outcomes and possible changes for future work.

2 Background

This section provides an overview of the theory underlying the thesis subject, introducing information on warehouse design and operations, combinatorial optimization, NP-complete problems, and Travelling Salesman Problem that is required to understand the following sections.

2.1 Warehouse

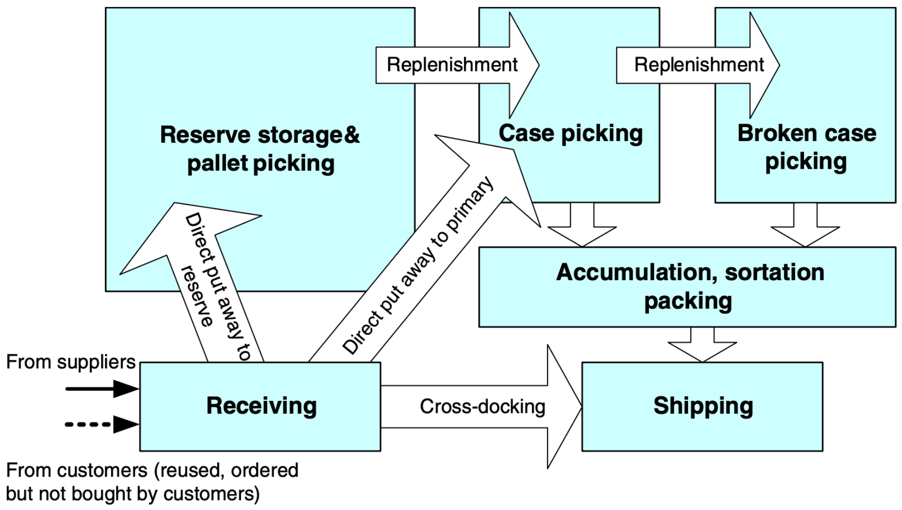
"Warehouses form an important part of a firm's logistics system allowing storage or buffering of products at and between points of origin and points of consumption" [4]. According to Warehousing and Transport Support Services Statistics provided by NACE [11] as of 2012 there existed 136.3 thousands warehouse facilities in the European Union, employing over 2.5 million people with a turnover of 478 754 million Euros (Figure 2).



	Value
Main indicators	
Number of enterprises (thousands)	136.3
Number of persons employed (thousands)	2 605
Turnover (EUR million)	478 754
Purchases of goods and services (EUR million)	320 000
Personnel costs (EUR million)	94 000
Value added (EUR million)	173 158
Gross operating surplus (EUR million)	79 000
Share in non-financial business economy total (%)	
Number of enterprises	0.6
Number of persons employed	1.9
Value added	2.8
Derived indicators	
Apparent labour productivity (EUR thousand per head)	66.0
Average personnel costs (EUR thousand per head)	37.3
Wage-adjusted labour productivity (%)	178.0
Gross operating rate (%)	16.6

Figure 2. Key indicators, warehousing and support activities for transportation (NACE Division 52), EU-27, 2012 - Source: Eurostat (sbs_na_1a_se_r2).

Most supply chains need product items to be stored and/or buffered. This means that warehouses play a very important role in a company's logistics performance. For that reason it's important to understand the inner-workings of the warehouses in order to minimize operational costs.



Replenishment
Replenishment
Reserve storage& pallet picking
Case picking
Broken case picking
Direct put away to reserve
Direct put away to primary
Accumulation, sortation packing
From suppliers
Receiving
Cross-docking
Shipping
From customers (reused, ordered but not bought by customers)

Figure 3. Typical warehouse functions and flows Smith [12].

Ackerman [13] describes some of the common tasks that a warehouse performs in addition to *storage* as follows:

Stockpiling, is the practice of using a warehouse as a reservoir to manage excess demand. This may happen in two cases: either there is seasonal production and level sales, or there is steady production and seasonal sales. In either case, the warehouse is where supply and demand are balanced.

Product mixing is the process of assembling half-finished products into customer-ordered products.

Consolidation is the act of gathering different products for a consumer order. The consumer can need goods that are manufactured in different locations or by different manufacturers.

Distribution is the inverse of consolidation. It, like consolidation, is largely justified by the freight savings realized in higher volume shipments. The seller pushes finished goods to the consumer by distribution, while the buyer pulls stocks through aggregation.

According to Smith [12] this list can be expanded by adding *receiving, inspection, inventory control, replenishment, order picking, checking, packing and marking* processes.

Unloading goods from the transport carrier, reviewing the inventory record, and inspecting for any quantity or quality inconsistencies are all part of the *receiving* operation.

Inspection is the stage in the flow that controls whether the purchased order is met and if the shipped goods satisfy the minimum quality requirements that the customers requested. The primary aim of inspection is to maintain data communication so that data regarding stock levels and stocking locations of products in the warehouse are up to date.

Order picking is the most important task in most warehouses. It entails obtaining the correct number of the correct items for a series of customer orders.

Inventory control entails moving goods to their proper locations in the factory after they have been received. It is important to keep track of which goods are kept where in order to monitor the positions and amounts of products on hand.

Order picking process involves the process of grouping and planning the customer orders, releasing orders to the order picking area, picking the products from storage locations and the disposal of the picked articles [4].

After the order has been picked the order should be checked through the *checking* process to assure that the order fulfills the customer order regarding quantity and quality.

After the orders have been selected and reviewed, the process flow is completed by *packaging* the orders for shipping and *marking* them according to customer demands or company policy.

2.1.1 Warehouse Layout

Warehouse design, as defined by Roodbergen, Vis, and Taylor Jr [14], is comprised of three interdependent parts: "the form of *systems* used, the *layout*, and the *policies* that govern all operating processes". To choose the system, product characteristics must be considered, for example larger and heavier products tend to be stored on pallets in pallet racks, while smaller products may be stored on shelving racks that have a lower storage capacity and require less floor space. More variations are described in Hackman, Frazelle, Griffin, *et al.* [15] in detail.

"For all warehouse systems a *layout* must be determined such that each system can achieve the required performance" [14]. In Figure 4 a typical order picking area is graphically depicted, while in Figure 7 some common questions of order picking system design are defined. The layout of the warehouse pick area can be defined through the following parameters: number of *aisles*, number of cross aisles that break down aisles into *sub aisles* and form *blocks*. "A *depot* functions as the central point where orders are administered to employees" [14].

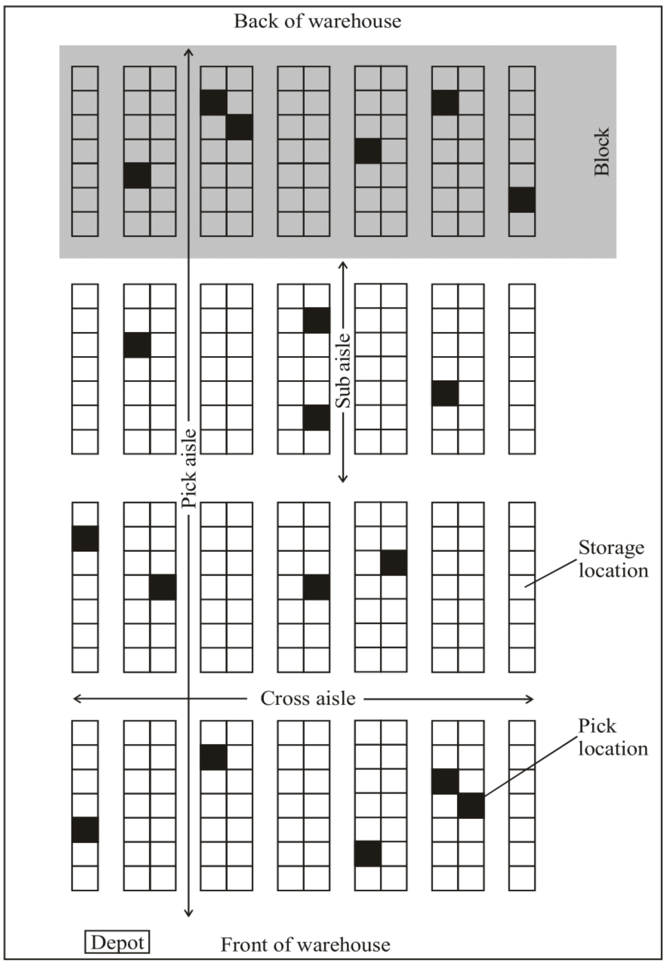


Figure 4. Illustration of a Typical Order Picking Area.

Other layouts exist and feature diagonal cross aisles and picking aisles that are not parallel, such as "Fishbone" or "Flying V" layout (Figure 6). These layouts appear in the literature, for example in the work by Pohl, Meller, and Gue [16], but because the parallel aisles with one or multiple blocks layout is most common in real life (Figure 5), we'll be focusing on such warehouse layout in this work.

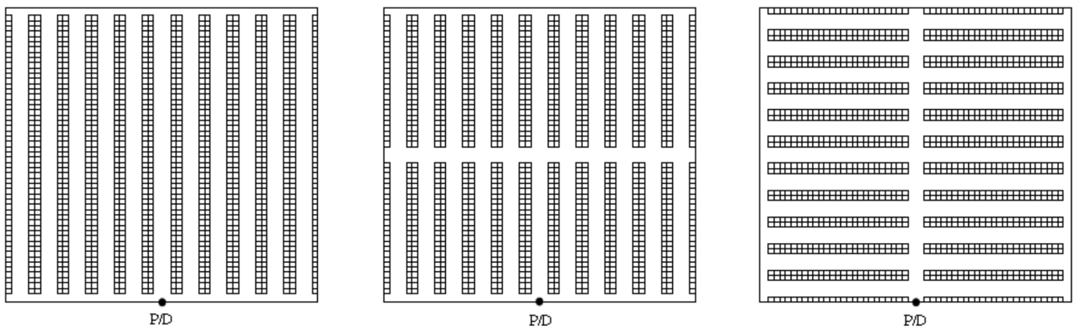


Figure 5. Traditional Warehouse Layouts.

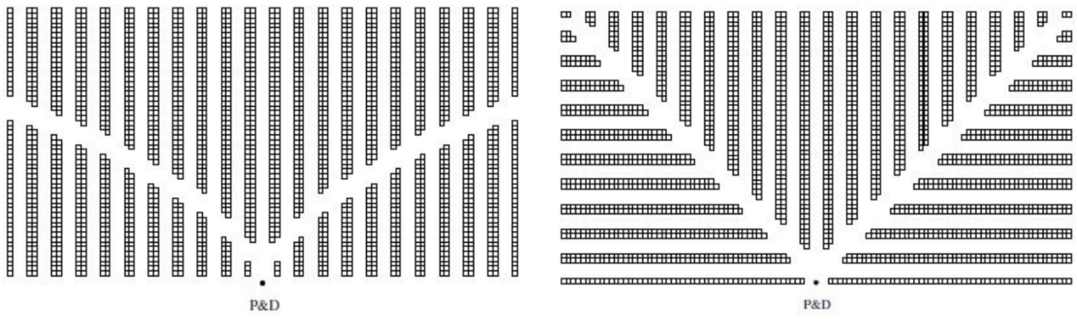


Figure 6. Flying-V (left) and Fishbone (right) Layouts.

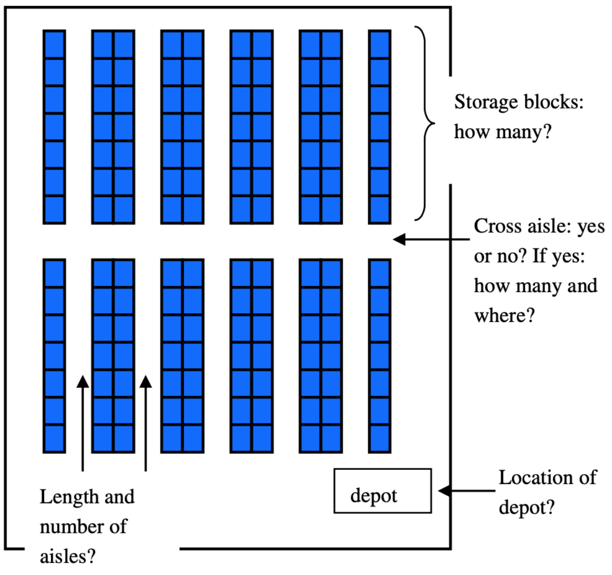

Storage blocks: how many?
Cross aisle: yes or no? If yes: how many and where?
depot
Location of depot?
Length and number of aisles?

Figure 7. Typical layout decisions in order picking system design Tompkins, White, Bozer, *et al.* [17].

2.1.2 Storage Policies

After the products are received at the warehouse they must be assigned storage locations. A *storage assignment policy* is a set of rules for determining this allocation. A large number of ways to assign products to storage location exists, and there are five ways that are used most frequently: random storage (Petersen and Schmenner [7], Choe [18]), closest open location storage (Hausman, Schwarz, and Graves [19]), full turnover storage, class-based and dedicated storage.

Random Storage. In the random storage policy, incoming goods are allocated to a warehouse location that is chosen at random from all suitable empty locations with equal probability. Because of its simplicity, this storage policy is often used in practice; for example, in dynamically evolving product assortments where insufficient data is available to evaluate demand frequencies of individual goods De Koster, Van der Poort, and Wolters [20].

Full Turnover-based Storage. "All products are ranked from most frequently to least frequently requested and all locations are ranked from best, usually meaning that they're closest to the depot, to worst" Roodbergen [21]. The products are then assigned to the available locations by matching these two rankings.

Dedicated Storage. In this policy each product has a fixed location in the warehouse it's assigned to. The downside is that even when the items are out of stock they're allocated a storage space, causing inefficient storage usage when compared to a random storage.

Class-based Storage. The concept of class-based storage is based on the Pareto's method. The goal is to organize products into classes so that the most frequently incoming/outgoing class accounts for 85% of the total turnover but only contains about 15% of the products stored. For each class dedicated storage spots are assigned in a random manner within each area.

"Generally, the number of classes equals three, which may give about 85% of the potential efficiency gains of full turnover-based storage" [19]. This allows storing frequently incoming/outgoing items closer to the depot and reducing the travel time required to get them while retaining some of the flexibility of the random storage policy within each class area.

Figure 9 illustrates the four variations of class-based (ABC) storage policies and Figure 8 shows the two common ways of implementing the Within-Aisle and Across-Aisle storage policies. A-items (black) account for products which turnover rate are high and the number of locations is rather small. C-items (light grey) represent products which average storage time are much longer than the storage time of A-items. B-items (dark gray) is an in-between category, concerning turnover rate and space needed [22].

"*Across-aisle storage* is an ABC storage policy in which A-items are allocated to the front-most position of each pick aisle. C-items are kept at the back of each pick aisle, while B-locations are stored in between" [21]. De Koster, Le-Duc, and Roodbergen [4] advocate for this method.

For *within-aisle storage*, all items in a pick aisle belong to the same item class. The A-items are located in the pick aisles closest to the depot. Jarvis and McDowell [23] support this approach.

For *nearest-aisle storage* all items in a subaisle belong to the same class. "The subaisles with their center closest to the depot contain the A-items. This is actually a variation on the within-aisle storage rule" [21]. This policy is the most effective for multiple block layouts, otherwise within-aisle storage is implemented.

With *nearest-location storage*, the A-items are assigned to the locations that are closest to the depot. "This policy is closely related to the method of diagonal storage Petersen and Schmenner [7], which defines class boundaries based on diagonal lines in the picking such that the A-items are closest to the depot" [21].

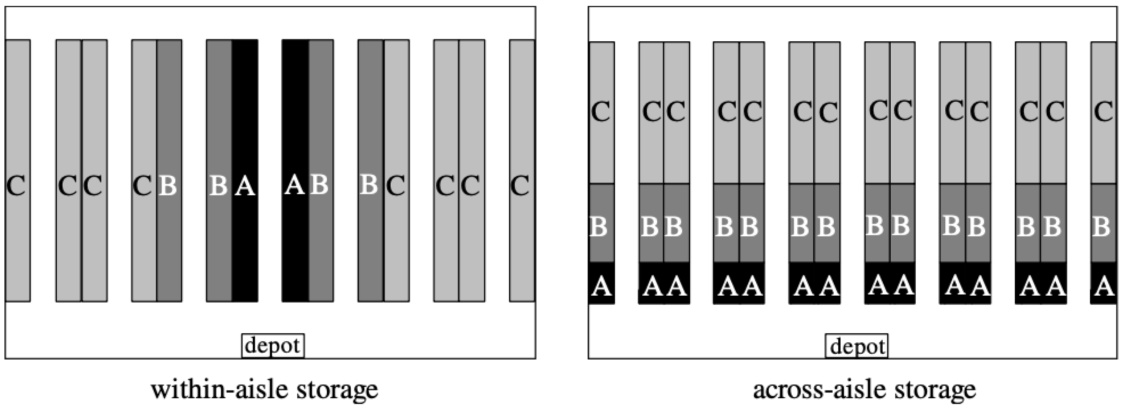


Figure 8. Two ways of organising the warehouse for within-aisle and across-aisle storage policies [4].

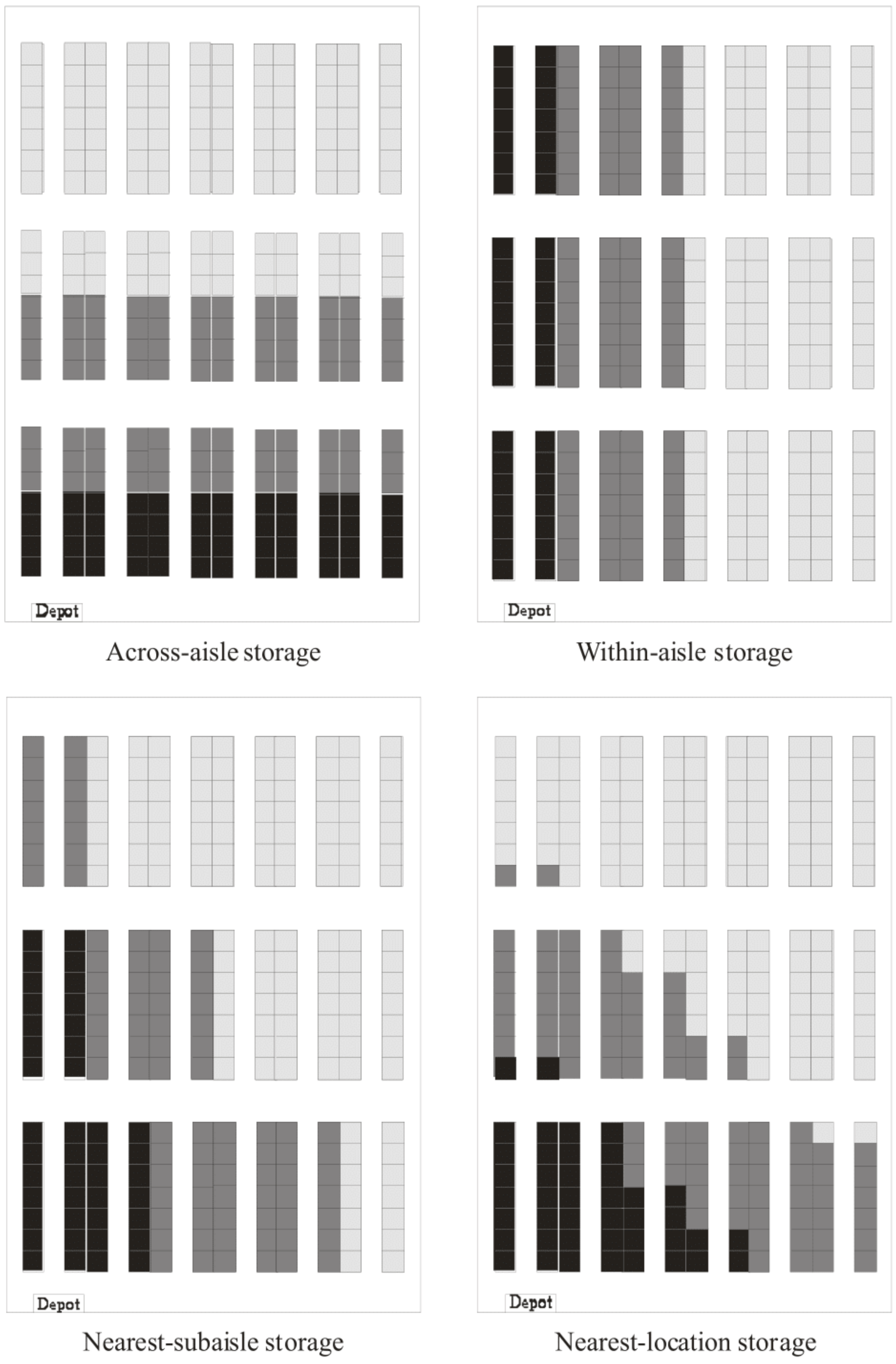


Figure 9. Example of Four ABC-Storage Assignment Policies [14]. Black locations indicate A-items, dark gray locations indicate B-items and light grey locations indicate C-items.

2.1.3 Order Picking Process

Order picking, as defined earlier in the work, entails grouping and planning customer orders, issuing orders to the order picking area, picking items from storage locations, and disposing of the picked articles [4]. Order picking is not a simple or cheap to automate and despite all of our technological advancements, order picking is still largely a manual activity Ackerman [13].

Order picking can be manual, power-assisted, automatic or it may be a combination of these methods. Order picking also varies in terms of order distribution among pickers; it can be discrete (single order) or batch picking.

Discrete (single order) picking requires the picker to assemble the total order before moving on to another one. An advantage of such picking method is that it maintains single order integrity and therefore avoids repacking and provides fast customer-order service. *Batch picking* is selecting of the total quantity of each item for a group of orders. Batches are then resorted into the quantities defined in each order. It allows to reduce travel to pick the total quantities of orders, especially when a large quantity of a stock unit needs to be collected. In this work we focus on discrete picking, however, batching methods are mentioned extensively in the literature and their effect on the order picking process would is an interesting research subject.

There are various order-picking system types that exist in warehouse systems today. The majority of warehouses employ people for order picking, the most common being the picker-to-parts systems. Another type of systems is part-to-picker and such systems include automated storage and retrieval systems (AS/RS).

The main objective for order picking systems is to minimise the order retrieval time (picking/travel time). Bartholdi, Hackman, and Technology. [5] refer to travel time as a "waste. It costs labour hours but does not add value".

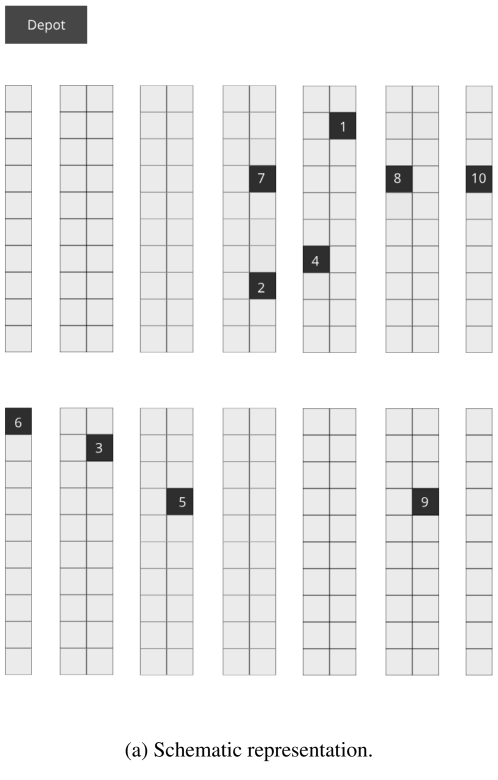
Minimising travel time means optimising the total travel distance of the order picker in a picking tour. Such problem in a warehouse is known as an *Order Picking Problem*, and it is a specific subclass of the Traveling Salesman Problem (TSP) – Steiner Traveling Salesman Problem [4], [6], [24]. This problem can be shortly formulated as following:

Given n products to pick in a rectangular warehouse what is the shortest tour to collect *all these products? [25]*

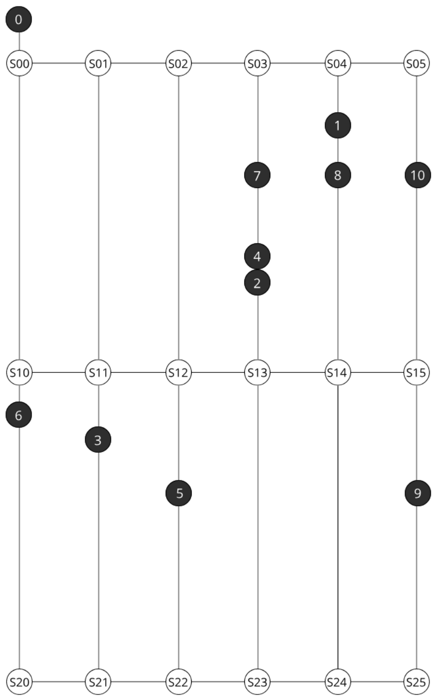
The *traveling salesman problem* got its name from the following situation. Starting in his hometown, a salesman must visit each city exactly once before returning home. The distance between each pair of cities is known to him and he needs to decide the order in which he would visit the cities so that the overall distance traveled is as short as possible.

The condition of the traveling salesman is close to that of an order picker in a warehouse. The order picker begins at the depot (home city), where he receives a pick list, travels to all pick locations (cities), and returns to the depot. A warehouse example with picking locations (black squares) and a corresponding graph representation are given in Figure 10.

Traveling salesman problem is an NP-hard problem (as shown by Karp [26]) in combinatorial optimization, important in theoretical computer science and operations research.



(a) Schematic representation.



(b) Graph representation.

Figure 10. Illustration of the warehouse order picking case in schematic form (a) and its graph representation (b).

2.2 Combinatorial Optimization Problems and NP-hard Problems

Our warehouse routing problems are classified as combinatorial optimization problems. Combinatorial optimization problems can be defined as those in which an optimal solution must be found from a finite number of possible solutions. Combinatorial optimization is a very active field of applied optimization research that combines techniques from combinatorics, linear and nonlinear programming, and algorithm and data structure theory to solve problems over discrete structures Geng [27].

Combinatorial optimisation problems can be distinguished as one of the two variants (Hoos and Stützle [28]):

- The *search variant*: given a problem instance, find a solution with minimal (or maximal, respectively) objective function value;
- The *evaluation variant*: given a problem instance, find the optimal objective function value (i.e., the solution quality of an optimal solution).

Recent advances in combinatorial optimization, including cutting-plane methods, branch and cut, branch and bound, local search and meta-heuristics together with advances in computer technology, have made it possible to apply combinatorial optimization to a wide range of practical problems. Many combinatorial optimization problems, however, are computationally unsolvable. Therefore, a practical approach for solving such problems is to employ heuristic (approximation) algorithms that can find near optimal solutions within a reasonable amount of computation time.

NP is a class of computational decision problems for which any given yes-solution can be verified as a solution in polynomial time by a deterministic Turing machine. A decision problem H is **NP**-hard when for every problem L in **NP**, there is a polynomial-time many-one reduction from L to H.

Traveling salesman problem was proven to belong to the class of NP-hard problems by Papadimitriou [29] and Itai et al. in 1984. We will describe the formulations of TSP and Steiner TSP in the following section.

2.3 Traveling Salesman Problem

The result of solving a standard TSP where the distance between two vertices is given by the shortest path in the warehouse will correspond to finding the optimal tour in the warehouse. Specifically, we define the problem through a complete graph, where the vertices represent orders (including the depot). "Every vertex has to be visited exactly once, by minimizing the travel distance. The distance between two vertices is given by d_{ij}" as defined by Pansart, Catusse, and Cambazard [25], [30]. The *conventional formulation* of the traveling salesman problem as defined by Dantzig, Fulkerson, and Johnson [31] is the following:

We define a variable x'_{ij} for each pair of products: $\forall i, j \in R$

$$x'_{ij} = \begin{cases} 1, \text{ if the tour uses the arc } (ij) \\ 0, \text{ otherwise} \end{cases} \tag{1}$$

$$\begin{array}{lll} \min & \sum_{i,j \in R} d_{ij} x'_{ij} & (1) \\ \text{subject to} & \sum_{j \in R} x'_{ij} = 1 & \forall i \in R (2) \\ & \sum_{j \in R} x'_{ji} = 1 & \forall i \in R (3) \\ & x'(S : \bar{S}) \geq 1 & \forall S \subset R : 2 \leq |S| \leq \frac{|R|}{2} (4) \\ & x'_{ij} \in \mathbb{N} \quad \forall i, j \in R (5) & \end{array}$$

Constraints (2) and (3) impose that the order picker comes exactly once at each product and leaves them exactly once. However, these two constraints are not enough to guarantee that the model's result has only one path. Constraints (4) defines that "for any partition into two subsets S and S, the order picker transits from S to its complementary at least once" [30].

Based on whether or not $d_{ij} = d_{ji}$ (i.e., if the cost of going from A to B is the same as going from B to A), the TSP can be divided into two general types: the symmetric TSP (STSP) and the asymmetric TSP (ATSP). In this work we're only concerned with the symmetric TSP. Also all of our TSP discussion in this work is about the Metric TSP, which means it satisfies the triangle inequality.

$$c_{ij} \leq c_{ik} + c_{kj}, \forall (i, j) \in A, \forall k \in V, k \neq i, j \tag{2}$$

2.4 Steiner Traveling Salesman Problem

The order picking problem can be formulated as a special case (Steiner) of the Traveling Salesman Problem (TSP). The Steiner special case of the traveling salesman problem was proposed by Cornuéjols, Fonlupt, and Naddef [24]. The principle is that the graph includes some vertices that are *required* to be visited and vertices of Steiner (*Steiner vertices*) that may or may not be visited. In addition, the graph is not complete and the edges can be travelled multiple times as well as vertices can be visited once or more.

The problem can be represented by a graph $G = (N, A)$ with node set $N = S \cup R$ and edge set A. Node subset $R = \{1, \ldots, n\} + \{0\}$ contains all n order items that need to be picked up in the warehouse, as well as the depot which is represented by node 0. Node subset $S = \{n+1, \ldots, n+p\}$ contains the Steiner nodes indicating all p "crossing nodes" between two or more aisles, where p equals the product of the number of cross aisles and the number of pick aisles in the warehouse. Edge set $A \subseteq R \times R$ represents all travel possibilities between order items and the depot.

This concept is illustrated by Figure 11 with 6 pick aisles and 2 cross aisles ($p = 12$) where 6 items and the depot location need to be visited ($n = 6$). Node subset $R = \{0, 1, \ldots, 6\}$ contains all nodes that need to be visited at least one, since they represent either an picking location or the depot. The Steiner nodes in $S = \{S00, S01, \ldots, S14, S15\}$ do not necessarily have to be part of an order picker's route. As a results of the special (rectangular) structure of the warehouse, they might, however, be visited while visiting nodes in R.

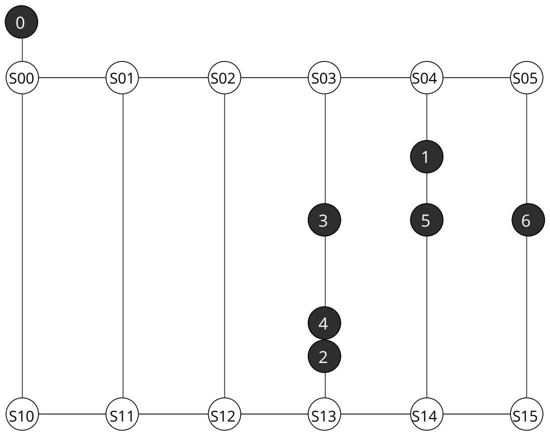


Figure 11. Warehouse: graph representation, small example.

3 State of the Art

The order picking literature has primarily focused on four major study streams: structure design, storage assignment, batching, and routing. In the following, two research streams are discussed: storage assignmnent and routing since they have the most significance for our work. For more details on the rest of the topics we refer to: De Koster et al. (2007), Gu et al. (2007), and Shah and Khanzode (2017).

The literature on warehouse storage policies is somewhat limited. examined the performance of an automated warehouse with random and volume-based storage. Schwarz, Graves, and Hausman [32] investigated the efficiency of an integrated warehouse with random and volume-based storage. Furthermore, Gibson and Sharp [33] discovered that finding large volume objects near the p/d point increases picking productivity significantly. They may not, however, provide details about how they are implementing volume-based storage. Jarvis and McDowell [23] state that the optimal storage strategy is to place the most frequently picked items in the aisle nearest the p/d point and the next most frequently picked items in the next aisle. Their study was constrained in that it assumed the aisles only enabled one-way travel and were only capable of transversal routing. In a number of working environments, this paper compares two volume-based storage policies to random storage. Routing seeks the fastest path through the warehouse to retrieve all things needed in an order.

Routing aims at finding the shortest tour through the warehouse for retrieving all items requested in an order. Different authors have proposed several general types of routing algorithms, specifically optimal routing policies, simple heuristics, and meta-heuristics [30].

The implementation of optimal routing algorithms and the comparison of routing heuristics are topics covered in the routing policy literature. Ratliff and Rosenthal [34] and Goetschalckx and Ashayeri [3] have developed optimal algorithms for routing pickers in a rectangular warehouse. Furthermore, De Koster and Van der Poort [35] compare optimal and the S-shape heuristic in a decentralized warehouse with no fixed p/d point. This work, on the other hand, focuses on heuristics in the more common manual warehouse. Hall [36] examined routing heuristics in a manual warehouse. In addition, Hall studied the effect of warehouse shape on distance approximations for multiple routing heuristics in a random storage warehouse. Petersen and Schmenner [7] expanded on this by testing a new routing heuristic and investigating the effect of form in a fixed-capacity warehouse.

In terms of TSP heuristics, "Makris and Giakoumakis [37] used a tweaked *k*-interchange heuristic to boost the solution of a basic routing heuristic. Grosse et al. (2014) investigated traditional architecture B with small aisles and used the savings algorithm for routing order pickers, among other things. Clarke and Wright (1964) suggested the savings formula, which begins with a series of tours in which each object is chosen individually. It then assesses the travel distance that can be avoided by combining two existing tours into a single tour and combines the tours that result in the greatest travel distance savings" Masae, Glock, and Vichitkunakorn [38].

Some authors have suggested exact algorithms developed originally to solve the TSP for the order picker routing problem in multi-block warehouses. To find an order picking tour with the least amount of travel time in a narrow-aisle warehouse, Roodbergen and Koster [6] used a branch-and-bound approach in their TSP formulation. "The branch-and-bound algorithm's downside is its inconsistent run-time behavior, which makes it unsuitable for practical implementations" [25]. Theys, Bräysy, Dullaert, *et al.* [10] applied the exact Concorde TSP algorithm to a traditional warehouse with two blocks, assuming that a picking tour begins and ends at a single depot in the front cross aisle, which may be in the middle (central depot) or at some other location (decentral depot). The exact concorde TSP algorithm was created to solve the symmetric TSP using a branch-and-cut approach. To find the shortest path for a given pick-list, the same concorde TSP solver was used.

The last work by Theys et al. researched the following questions: are the findings for routing in single-block or conveyor-equipped warehouses also valid in conventional multi-parallel-aisle warehouses, or does the adoption of problem-specific features make dedicated multiple-block order picking heuristics more efficient? How do the performances of TSP-based heuristic solution strategies correspond to those of dedicated heuristics (other than S-shape) suggested in the literature, and can they be developed further? The latter question has inspired this thesis to investigate further on the usage of the TSP heuristics in the warehouse order picking problem.

4 Implementation

In this section, we will explain how the implementation was carried out and what its key components are. We will start with preprocessing stage and then go over each of the implemented algorithms (Nearest Neighbor, 2-opt, Lin-Kernighan) and TSP Solvers (Google OR-Tools, TSP Concorde) that were integrated in the final java application. The implementation is available through author's GitLab repository (see Appendix B).

4.1 Preprocessing

Preprocessing stage was implemented in Python. The main objective of the preprocessing stage is to construct a valid input for the TSP heuristics based on the given warehouse layout and order information.

This information is provided as a set of parameters containing the following information about the warehouse layout: number of blocks, number of aisles, number of storage locations per sub-aisle (rows), type of the storage policy, as well as some optional parameters as shown in Listing 2. We refer to these parameters as warehouse parameters in the next sections. Number of orders (pick locations) must also be provided.

The primary way to run the python application is using `generate.py` with `PREFIX` argument which defines the prefix for the generated files. Other optional arguments are also available, see Listing 3.

A simple command line interface is implemented using the *Click Library* [39] that allows entering the command line arguments. It also provides a progress bar based on the total number of tests to be generated.

The warehouse and order parameters in this case are defined through global array variables as shown in Listing 1. The test cases are then generated by finding all the possible combinations of the provided parameter values.

```
AISLES = [4, 8, 12, 20]
BLOCKS = [1, 3, 5, 11]
ROWS = [50, 100, 500]
STORAGE_POLICY = ['RANDOM', 'ABC', 'ABCACROSS']
ORDERS = [5, 7, 20, 50]
ABC_CAPACITY = [(1, 1), (1, 3), (2, 4), (4, 8)]
ABC_ACROSS_CAPACITY = [(0.1, 0.2), (0.1, 0.2), (0.1, 0.2)]
SEEDS = list(range(10))
```

Listing 1. Warehouse parameters defined as global arrays in `generate.py`.

For each test case the `generate()` function first generates a Steiner graph by calling the `generate_steiner_graph()` function. The generated Steiner graph is then passed to the `process()` function that removes the Steiner nodes and produces the final graph that can be used by the TSP algorithms.

Finally, the generated graph is stored in two different formats using the `save_graph_file()` and `save_tsp_file()` functions.

The full flow of the function calls is described by the sequence diagram in Figure 12. Here, `graph_args` include all the parameters of the test case, and `graph_args` only consist of *blocks*, *aisles*, *rows*, and *orders* values.

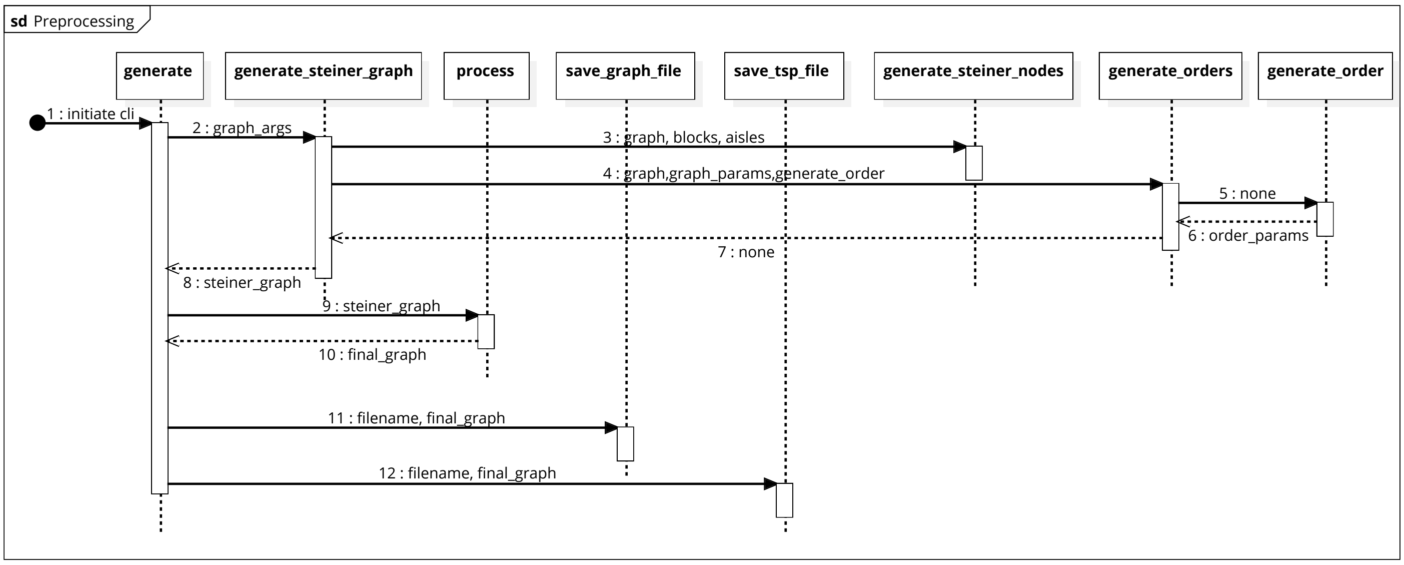


Figure 12. Sequence Diagram: Preprocessing.

```
process.py
python3 process.py --help
usage: process.py [-h] [--a-prob A_PROB] [--b-prob B_PROB] [--c-prob C_PROB]
                  [--a-capacity A_CAPACITY] [--b-capacity B_CAPACITY]
                  [--seed SEED]
                  blocks aisles rows orders storage_policy

To generate a simulation file define following parameters:
positional arguments:
  blocks                Number of blocks
  aisles                Number of aisles
  rows                  Number of product positions per sub-aisle
  orders                Number of orders
  storage_policy        Storage Policy.
                        Existing policies: RANDOM, ABC, ABCACROSS
optional arguments:
  -h, --help            show this help message and exit
  --a-prob A_PROB       Probability to place an order of type A
  --b-prob B_PROB       Probability to place an order of type B
  --c-prob C_PROB       Probability to place an order of type C
  --a-capacity A_CAPACITY
            Capacity for A (# of aisles for ABC, percentage for ABCACROSS)
  --b-capacity B_CAPACITY
            Capacity for B (# of aisles for ABC, percentage for ABCACROSS)
  --seed SEED           Random seed
```

Listing 2. Arguments for running process.py.

```
generate.py
./generate.py --help
Usage: generate.py [OPTIONS] PREFIX
Options:
  --max-tests INTEGER  maximum # of tests to be generated
  --help               Show this message and exit.
```

Listing 3. Arguments for running generate.py.

4.1.1 Steiner Graph Construction

The construction of the Steiner graph happens in the `generate_steiner_graph()` function. It starts by parsing the input arguments and creating an empty graph using the *NetworkX Library* [40].

As described in Section 2.4 the Steiner TSP is represented by a graph $G = (N, A)$ with two types of nodes: Steiner and order nodes. Each node is defined as Steiner or order nodes by setting the `steiner` node attribute to `True` or `False`.

Steiner nodes are nothing else but the intersection points between aisles and cross aisles which makes it easy to calculate their exact quantity. This calculation is performed in the `generate_steiner_nodes` function where all of the Steiner nodes are generated. The neighbouring Steiner nodes are then connected horizontally with edges of weight equal to 2. This weight value is selected for our simulation but can be adjusted when needed. A snippet of the described function can be seen in Listing 4. Steiner nodes are named using the format `S<block><aisle>` and follows the zero based naming convention. For example the Steiner node at the intersection of block 2 and aisle 3 has the name `S23`.

Afterwards, the depot is added as node `0` and connected to the `S00` Steiner node with an edge of weight equal to 0.

```python
def generate_steiner_nodes(g, blocks, aisles):
    # Generate steiner nodes
    for block in range(blocks + 1):
        for aisle in range(aisles):
            g.add_node('s%d%d' % (block, aisle), steiner=True)

    # Generate horizontal edges
    for block in range(blocks + 1):
        for aisle in range(aisles - 1):
            source = 's%d%d' % (block, aisle)
            target = 's%d%d' % (block, aisle + 1)
            weight = 2
            g.add_edge(source, target, weight=weight)
```

Listing 4. Snippet of the `generate_steiner_nodes` function.

4.1.1.1 Storage Policies

The storage policy defines the position of the products in the warehouse and influences the order locations. For that reason, for each of the storage policies we define a function that generates order nodes based on the specifics of that storage policy (see Figure 13). These functions all return a tuple with the following information defining location of an order: `(block, aisle, row, position)`. Here, `row` means vertical position of the order in the sub-aisle and `position` is either equal to 0 or 1, meaning it is either on the left or on the right side of the selected aisle respectively. The latter is based on the assumption that aisle width is negligible and the products facing each other on the same aisle have a distance of zero between them.

The simplest storage policy is the *Random* policy where the location of any product is random and, therefore, when generating orders we don't need to take any additional considerations. The function implementing the random storage policy can be seen in Listing 5.

```python
def generate_order_random_policy(blocks, aisles, rows):
    block = random.randint(0, blocks - 1)
    aisle = random.randint(0, aisles - 1)
    row = random.randint(1, rows)
    position = random.randint(0, 1)
    return block, aisle, row, position
```

Listing 5. Function generating an order based on the Random storage policy.

Other storage policies, namely Within-Aisle (`ABC`) and Across-Aisle (`ABCACROSS`) policies both belong to the class-based storage policy type and are implemented in a similar manner. Main difference, as may be seen from Figure 9 on the top left and right images, is in the direction that the warehouse storage is split into sections for the three product types: A, B and C.

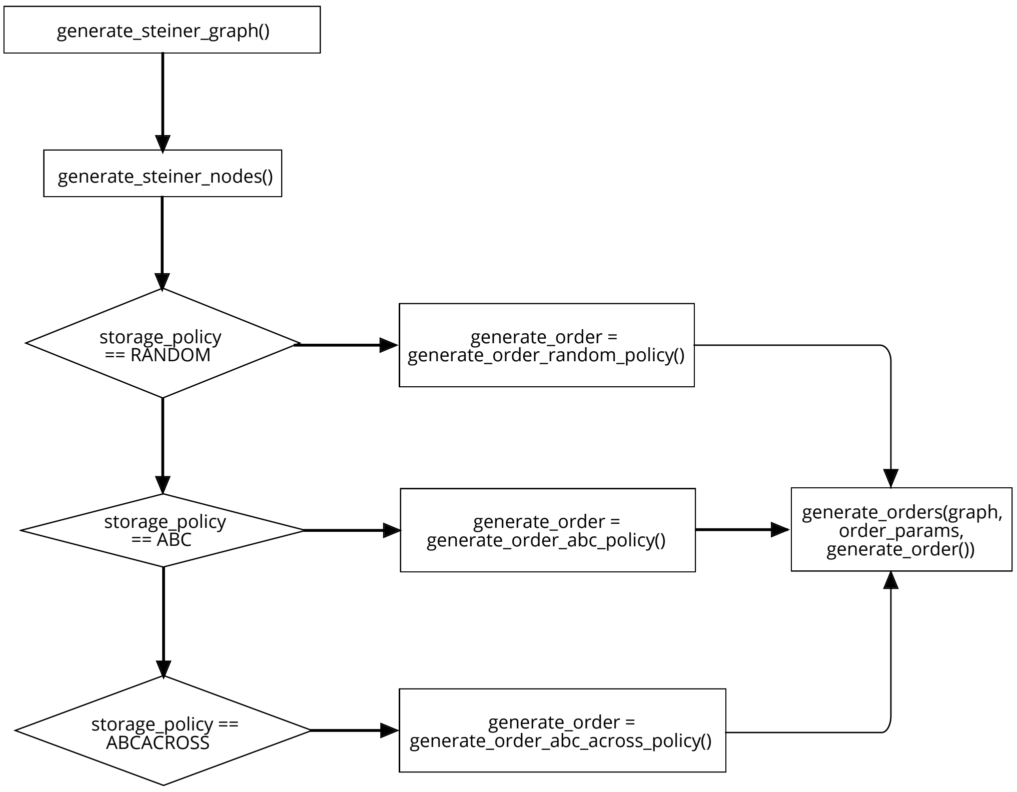

generate_steiner_graph()
generate_steiner_nodes()
storage_policy == RANDOM
generate_order = generate_order_random_policy()
storage_policy == ABC
generate_order = generate_order_abc_policy()
generate_orders(graph, order_params, generate_order())
storage_policy == ABCACROSS
generate_order = generate_order_abc_across_policy()

Figure 13. Flowchart: orders.py.

There are different aspects we need to consider when implementing these storage policies. First, we need to define the probability of each of the product types to occur among the orders. As was described in Section 2.1.2, type A items are the most frequently occurring, type C are the least frequently occurring and type B items are in between. In our implementation these values are predefined with probability for type A equal to 60%, for type B - 30%, and for type C - the remaining 10%. While these values are fixed for our simulations, they can be modified through optional parameters as was shown in Listing 2.

Other parameters that define the class-based storage policies are the storage capacities defined for each product type. We define these values differently for `ABC` storage policy and for `ABCACROSS` storage policy. When defining `a_capacity` and `b_capacity` values for `ABC` policy we define a specific number of aisles that each type of products will be assigned to. At the same time for the `ABCACROSS` policy these values are defined as percentage of the total number of rows. This decision is mainly related to the fact that the values selected for experiments are low for the number of aisles in accordance with common warehouse configurations as described by other papers [6] [21]. These values will be discussed in the next section.

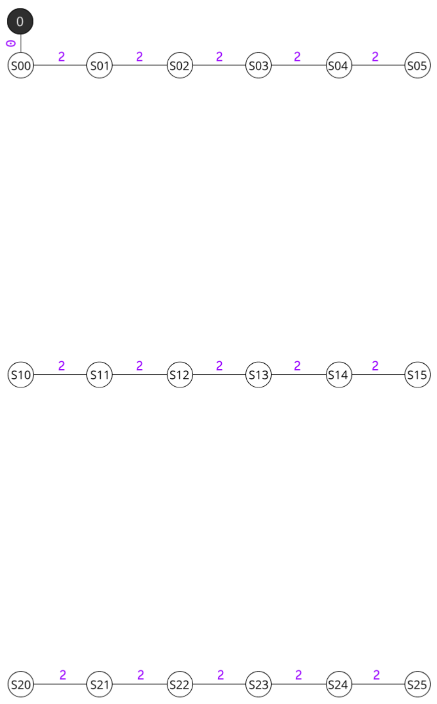
Based on the `policy` parameter appropriate function for order generation is determined from the three available as shown in Figure 13. It's worth noting, that our implementation allows trivial expansion on the available storage policy types by adding corresponding order generation function and another values in the `StoragePolicy` enumerator class.

Finally, when `generate_order()` function object is defined the `generate_orders()` function is executed, where for the defined number of orders the `generate_order()` function is called to define the potential position of each order. To keep track of the nodes' positions a `set` is used (`nodes_set`). If that position is not taken (doesn't exist in the `nodes_set`) it's added to the set, otherwise a new position is calculated until a unique one is found. All the generated orders are added to the `order_nodes` dictionary with a key based on the block, aisle, and position values.

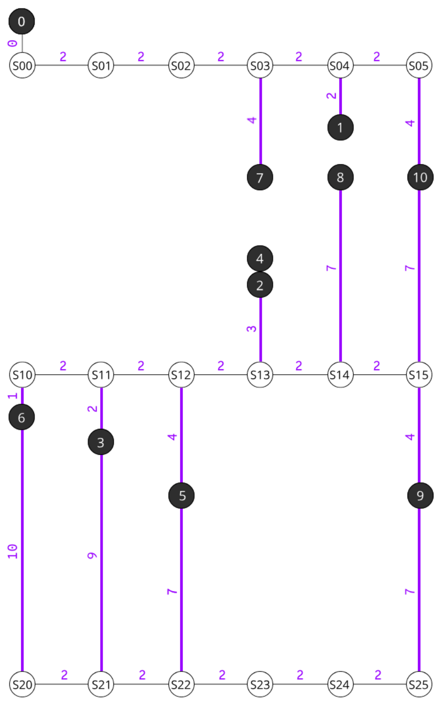
After generating all of the steiner and order nodes the next step is to create edges between these nodes. First, the Steiner nodes are connected to the order nodes that are placed closest to them. For each subaisle it means connecting the top steiner node with the first order node below it and connecting the bottom steiner node with the first order node above it. This will result in a situation illustrated in Figure 14b.

After that is done, the edges between the order nodes should be constructed as shown in Figure 14c. Finally, the vertical edges between the Steiner nodes must be created in the aisles where no orders were placed (see Figure 14d).

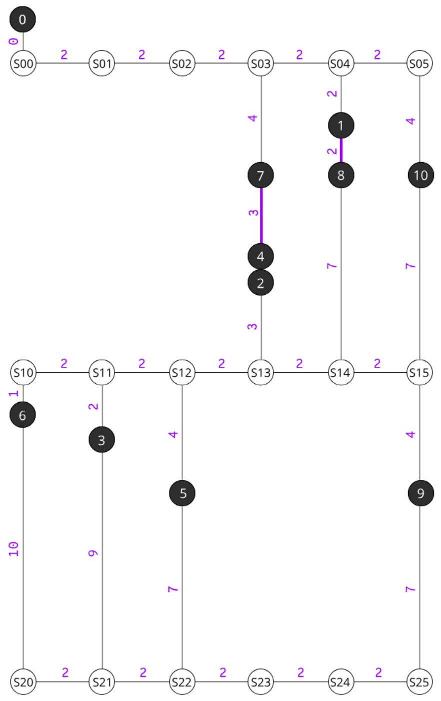
As a result we will receive a graph that contains Steiner nodes (S00. . . S<blocks><aisles - 1>) and order nodes (0. . . orders). This graph can be used as an instance of the Steiner TSP problem.



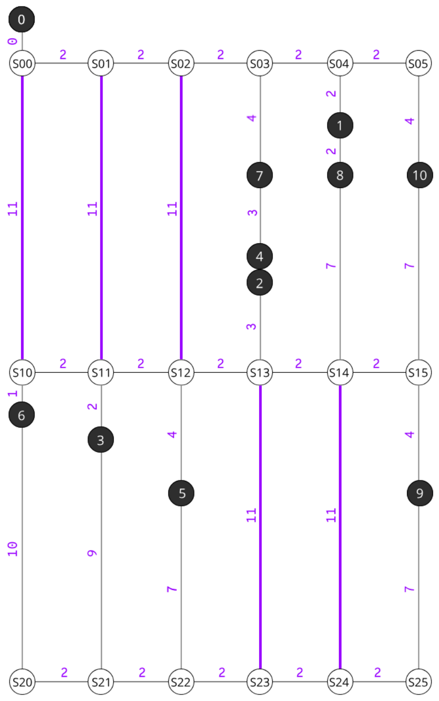
(a) Step 1: Generate depot, Steiner nodes and horizontal edge between Steiner nodes.



(b) Step 2: Create edges between Steiner nodes and order nodes closest to them.



(c) Step 3: Create edges between the order nodes.



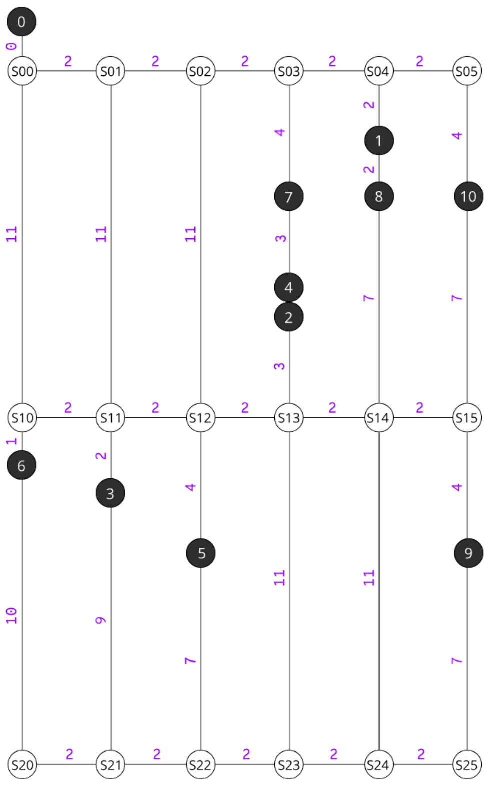
(d) Step 4: Connect the rest of the Steiner nodes vertically.

Figure 14. Constructing Steiner Graph for the order picking case illustrated in Figure 10a.

4.1.2 Final Distance Matrix Construction

In order to use TSP algorithms on our order routing problem we need to take one more step and "eliminate" the Steiner nodes from the constructed Steiner graph by finding the distances between order nodes. This step is implemented in the `process()` function. In order to achieve that result, the Dijkstra algorithm for finding the shortest path between the order nodes is used. This is done with the use of the *NetworkX Library* [40] for working with graphs and networks. Applying this algorithm to our Steiner graph results in a new graph where the shortest path between each order node is found.

Figure 15 illustrates the resulting Steiner graph that is passed to the `process()` function (15a) and the corresponding distance matrix that represents distances between order nodes in the final graph (15b) with the Steiner nodes removed.



(a) Final Steiner Graph.

	0	1	2	3	4	5	6	7	8	9	10
0	**0**	10	14	15	13	19	12	10	12	23	14
1	10	**0**	12	17	11	17	18	8	2	13	8
2	14	12	**0**	9	1	9	10	4	12	9	14
3	15	17	9	**0**	10	8	5	13	15	12	17
4	13	11	1	10	**0**	10	11	3	13	10	15
5	19	17	9	8	10	**0**	9	13	15	12	17
6	12	18	10	5	11	9	**0**	14	16	13	18
7	10	8	4	13	3	13	14	**0**	10	13	12
8	12	2	12	15	13	15	16	10	**0**	11	10
9	23	13	9	12	10	12	13	13	11	**0**	13
10	14	8	14	17	15	17	18	12	10	13	**0**

(b) Final Distance Matrix.

Figure 15. Steiner graph and the distance matrix constructed from this Steiner graph as the output of the `process.py` function.

4.1.3 Output Data Types

Finally, the resulting graph is saved in two formats:

- `.tsp` type is generated by the `save_tsp_file()` function as an input for the TSP Concorde solver and follows the format described in TSPLIB [TSPLIB95]. An example of such file is shown in Figure 16b;
- `.graph` files are used by the rest of the algorithms and follow the format shown in Figure 16a and are generated by the `save_graph_file()` function.

```
#Nodes
n - number of nodes
node_0
...
node_n
#Edges
m - number of edges
node_0 node_1 weight(0,1)
...
node_i node_j weight(i,j)
...
node_m-1 node_m weight(m-1,m)
```

(a) `.graph` file format.

```
NAME: filename
TYPE: TSP
DIMENSION: 7
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
 0  9  13 13 12 17 10
 9  0  10 16 9  16 17
 13 10 0  8  1  8  9
 13 16 8  0  9  6  3
 12 9  1  9  0  9  10
 17 16 8  6  9  0  7
 10 17 9  3  10 7  0
EOF
```

(b) `.tsp` file format (example).

Figure 16. The two file formats generated by the preprocessing stage representing the TSP graph.

4.2 TSP Algorithms and Solvers

To solve the TSP problem, local search algorithms are commonly used. They refine the current solution iteratively by looking for a stronger one in its pre-defined neighborhood. When there is no better solution in the given neighborhood, or when a certain number of iterations has been completed, the algorithm terminates.

Let's begin with describing the traditional λ-opt TSP operator. The theory behind these operators is that a TSP instance's solution tour can be improved by switching any of its edges if the new edges reduce the tour's length. In each step λ links of the current rout are replaced by λ links in such a way that a shorter tour is achieved.

It is evident that an optimum tour for a TSP with n cities must be n-optimal. However, the method's complexity increases with the scale of the operator: 2-opt is a $O(n^2)$, 3-opt is a $O(n^3)$, and so on, producing larger values of λ unusable.

We will use the following notation to describe the algorithms.

- $c(\cdot)$ the weight (cost) of an edge or a tour;
- $G[i]$ array represents the neighbours of i in the graph G;
- $T[i]$ represents the neighbours of i in the tour T, so its predecessor and successor;
- an edge is composed of a head and a tail: (t_i, t_j)

All of the implemented algorithms are classified as local search algorithms, Google OR-Tools are also implementing an effective heuristic, while Concorde TSP is based on the branch-and-cut algorithm (combination of cutting plane method and the branch-and-bound algorithm). Before describing each of the algorithms and solvers in more detail, the high level overview of the implementation is given in section 4.2.1.

It is also worth noting that the implementation of the TSP algorithms was performed in reference to articles by Maheo [41], [42]. These papers are often referred to when discussing the algorithms in the following sections.

4.2.1 Overview

The traveling salesman problem instance derived from the warehouse routing problem, as was described in the previous section, is solved using the integrated third-party TSP solvers and our Java implementation of the TSP heuristic algorithms. In order to use all these TSP solving options in the same platform an extensible framework was created.

The `abstract class TSP` implements the essential functionality that is common for all of the TSP algorithms and solvers. Each of them is implemented in a separate class that inherits from the TSP class. They all implement the `_optimize()` method that solves the TSP. The following TSP heuristic algorithms are implemented: Nearest-Neighbour, 2-opt, Lin-Kernighan.

For all the custom implemented TSP heuristics as well as for the Google OR-Tools integration the `_optimize()` method includes the calculations required for solving the problem based on the data provided by the `.graph` file. In the case of Concorde TSP, this method calls its command line executable and solves the TSP based on the distance matrix loaded in the `.tsp` file. The results of the Concorde executable are shown in the console and parsed using Regular Expressions.

The Application class is responsible for running the TSP algorithms and solvers for all the test files and saving the results into a `csv` file. All the `.graph` and `.tsp` files in the specified directory with a defined prefix are iterated over. Each file name is generated the way that it contains the test parameters and they are parsed into a `TestParams` data structure to be used by algorithms and saved into a csv file.

Additionally, a `GraphReader` class is used for the `.graph` files to parse the graph data into `nodes` and `edges` arrays that are used by the TSP algorithms and Google-OR solver.

An overview of the classes for the application can be seen in Figure 17. For the complete class diagram see Appendix A.

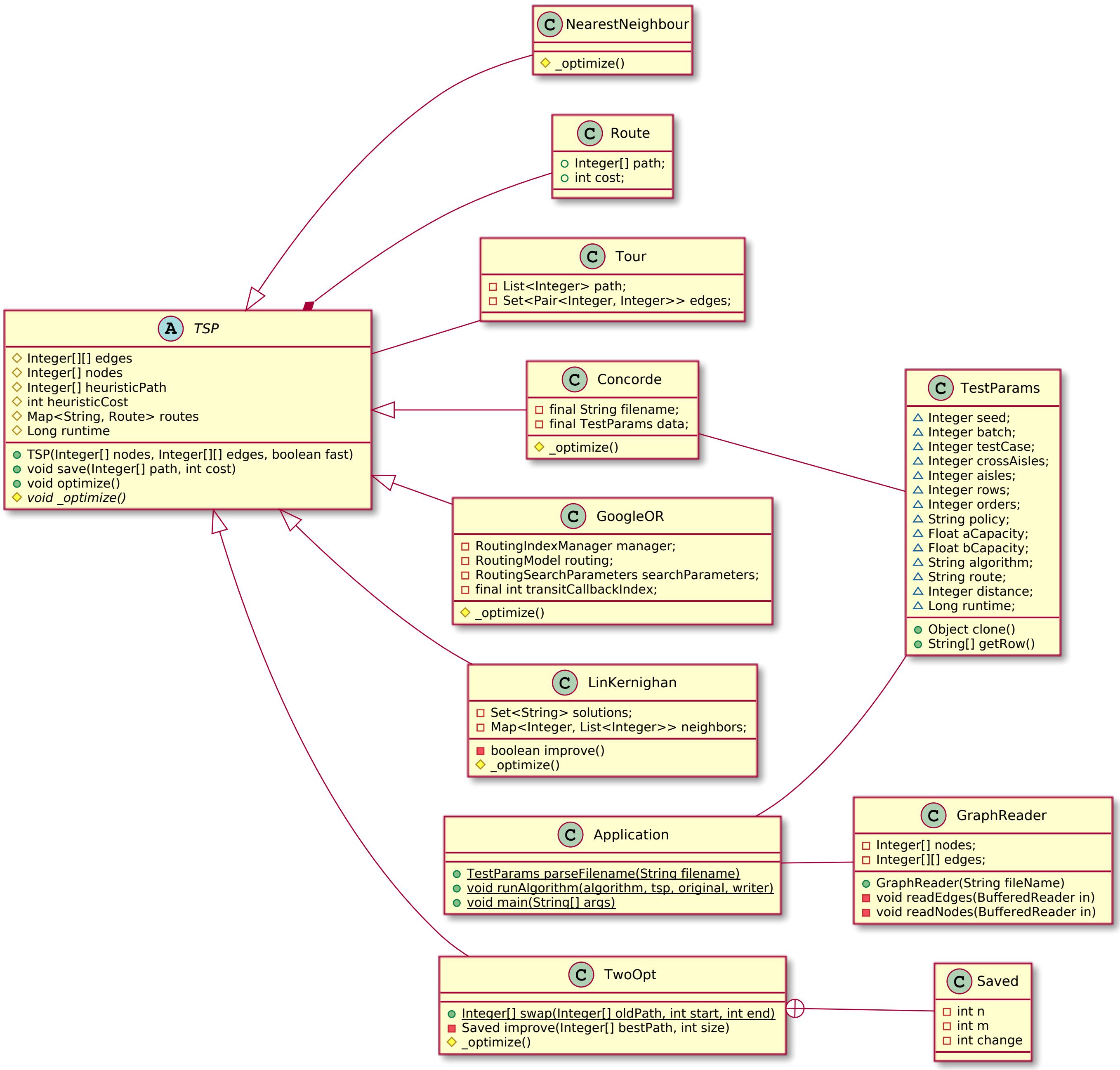


Figure 17. Class Diagram for Java Application.

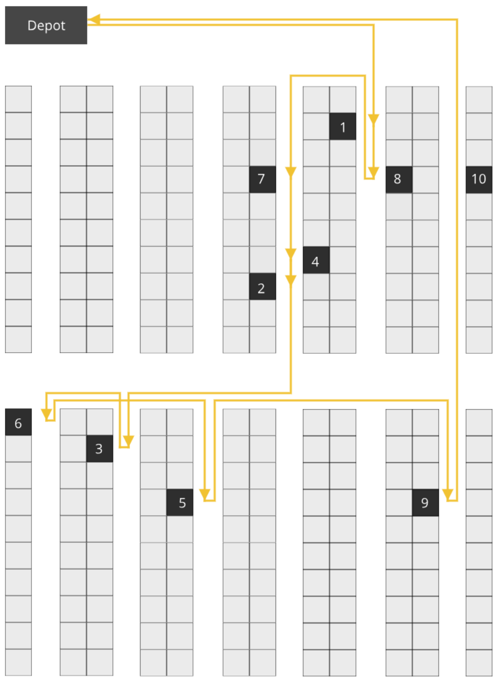
4.2.2 Nearest-Neighbour Algorithm

The basic operator of the λ-operators is the 1-opt operator. The algorithm will choose the nearest neighbor for each node before all nodes have been reached, at which point it will reconnect with the starting node. The pseudocode for the algorithm is given in Listing 6.

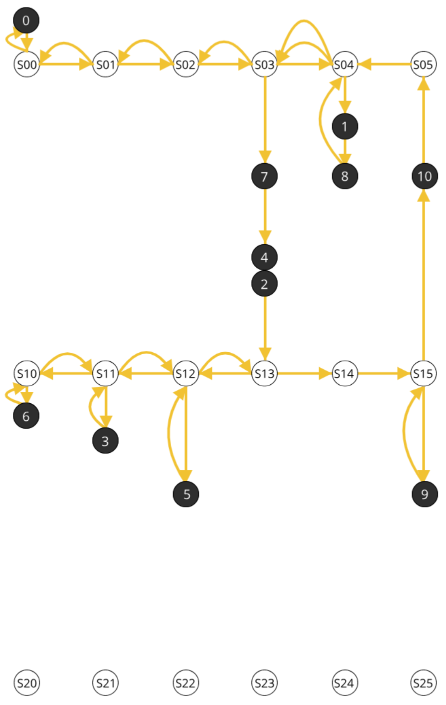
The Nearest Neighbour algorithm is inefficient since it does not consider the final relinking stage and may result in a local solution with a very long edge to return to the depot. Figure 18 illustrates the route constructed by this algorithm for the given warehouse.

```
node = 0
visited = set()
while len(visited) < len(nodes):
    tour.append(node)
    visited.add(node)
    # Find the closest, non-visited neighbour
    next = find_closest(G[i], visited)
    node = i
```

Listing 6. Nearest Neighbour pseudocode as described by Maheo [41].



(a) Schematic representation.



(b) Graph representation.

Figure 18. Path constructed by the Nearest Neighbour algorithm. Path cost: 88.

4.2.3 2-opt algorithm

The 2-opt algorithm is most likely the most fundamental and commonly used TSP local search heuristic. 2-Opt begins with an arbitrary initial tour and incrementally strengthens it by allowing consecutive adjustments that exchange two of the tour's edges with two other edges. Here, we want to choose two edges that, if swapped, will result in a shorter tour. The algorithm terminates in a local optimum in which no further improving step is possible.

A 2-opt move consists in finding a pair of nodes (i and j) for which changing their outgoing edge with a new one will reduce the cost of the tour. In other words, we replace: $(i, i+1)$ with (i, j) and $(j, j+1)$ with $(i+1, j+1)$. The gain offered by such move is calculated as the difference between the old edges and the new ones as shown in Equation (3). If the gain is positive, we have an improving move.

$$g = c(i, i+1) + c(j, j+1) - c(i, j) - c(i+1, j+1) \tag{3}$$

To complete the move, the tour is kept unchanged until node i, add its new neighbor (tail of the chosen edge), append the tour between $i+1$ and j in reverse order, and finish with the tail of j and the rest of the original tour. This is known as a "swap." An example of such swap is shown in Figure 19.

Figure 22 illustrates the route constructed by this algorithm for the given warehouse.

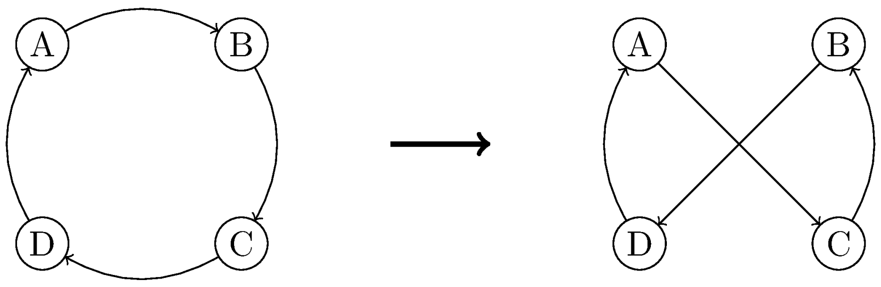
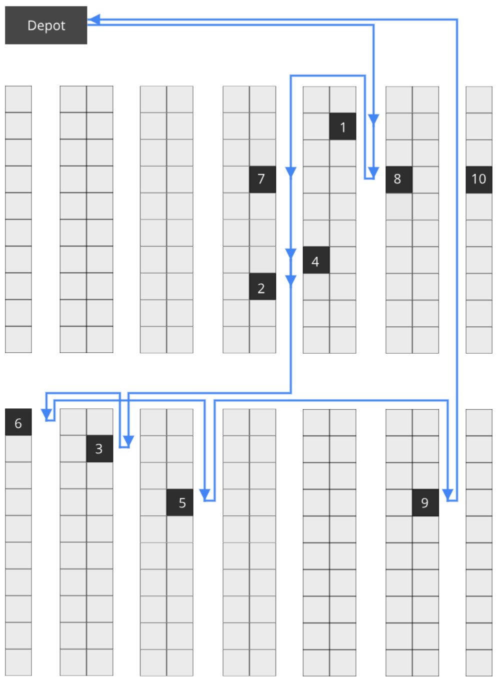


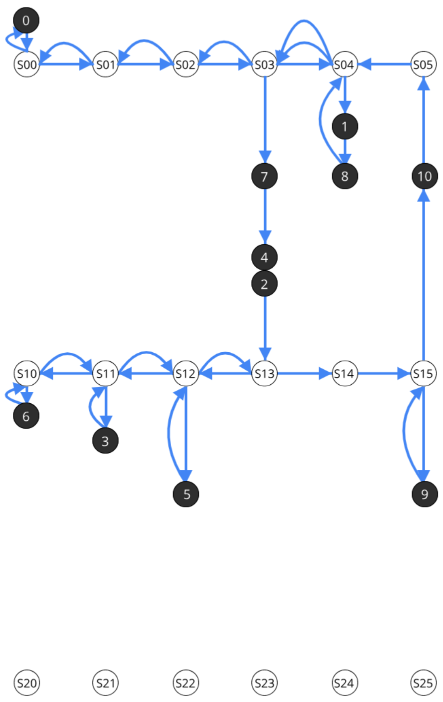
Figure 19. An example of a 2-opt move.

```
while improved:
    best = c(tour) # start with an initial tour
    size = len(tour)
    improved = False
    for i in tour[0:size-3]:
        #  i+2 because i+1 will be the tail of the edge
        for j in tour[i+2:size]:
            # Calculate gain: old edges - new edges
            gain = c(i, i+1) + c(j, j+1) - c(i, j) - c(i+1, j+1)
            if gain > 0:
                best -= gain
                # i is the last element in place
                tour = swap(tour, i + 1, j)
                improved = True
                break # return to while
```

Listing 7. 2-opt pseudocode as described by Maheo [41].



(a) Schematic representation.



(b) Graph representation.

Figure 20. Path constructed by the 2-opt algorithm. Path cost: 88.

4.2.4 Lin-Kernighan

A generalization of the simple principle that defines the 2-opt algorithm forms the basis for one the most effective [43] approximate algorithms for solving the symmetric TSP, the Lin-Kernighan algorithm.

As it was described in the beginning of this section, Lin-Kernighan algorithm is a variation of the λ-opt algorithm. The λ-opt algorithms is based on the consept of λ-opt:

"A tour is said to be λ-optimal if it is impossible to obtain a shorter tour by replacing any λ of its links by other set of λ links." [43]

Unfortunately, the number of operations to test all λ-exchanges increases rapidly as the number of cities increases. In a naive implementation the testing of a λ-exchange has a time complexity of $O(n^{\lambda})$. Additionally, the λ value must be defined in advance. The Lin-Kernighan heuristic answers the question: *which λ-opt to execute to improve the current tour?*

The algorithm keeps two sets of edges to represent the moves to execute: one with edges that will be omitted from the current tour and one with edges that will be added. Removing and inserting λ edges is equal to performing a λ-opt step.

Lin-Kernighan begins by choosing a node from which to begin the current tour (t_1); from there, it chooses either the predecessor or successor of the node on the tour (t_2), creating the first edge to eliminate. Then it chooses the first edge with a positive gain to have in the neighbours of t_2 that do not belong to the tour (t_3). It chooses either its predecessor or successor from t_3 (t_4). If relinking t_4 with t_1 results in a better tour, we restart the algorithm with the new tour; otherwise, if the gain remains positive, we search for another node outside of the tour (t_5) with a possible edge to add. This process continues. We may define four sections of the algorithm based on the description in Helsgaun's report [43]:

1. The main loop which will be use to restart the search (Listing 8).
2. The selection of the first two edges, that is selecting: t_1, t_2, and t_3.
3. The selection of the next edge to remove, which is called `chooseX()`, during which we may stop the search if we have an improved tour.
4. The selection of the next edge to add, `chooseY()`.

```
# tour is the current tour to improve
while improved:
improved = improve(tour)
```

Listing 8. Lin-Kernighan: Main Loop; as described by Maheo [41].

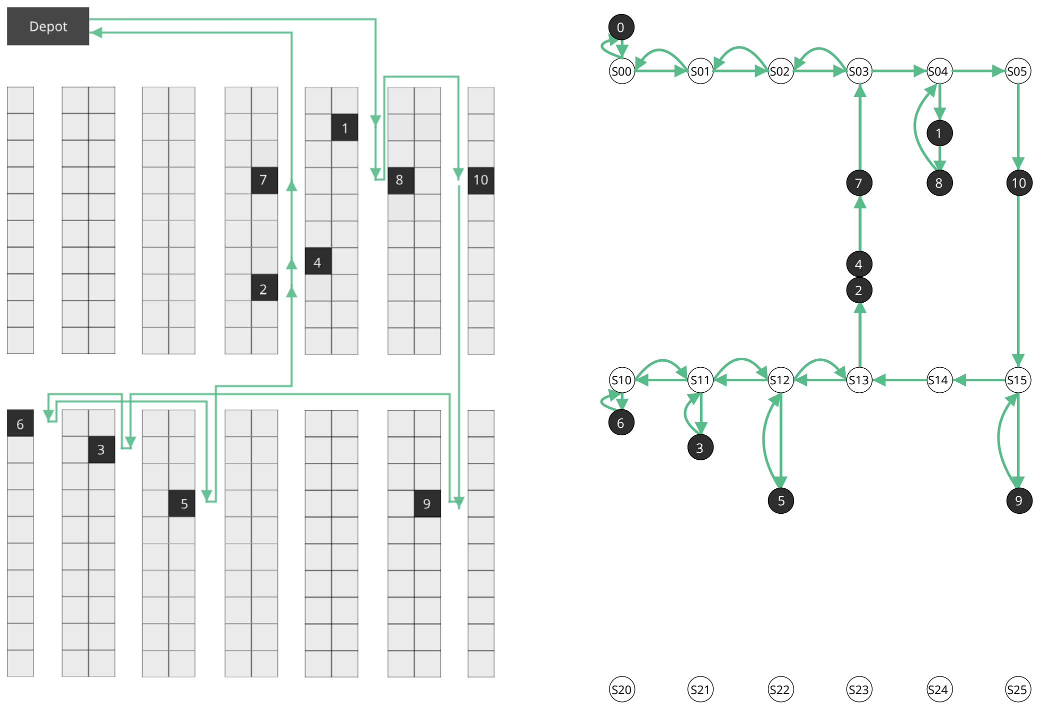
The move selection loop (Listing 9) is the core of the algorithm: it chooses the nodes to optimise from, the first edge to remove and the first to add, then it calls `chooseX()`.

```
# tour is the current tour to otimise
for t1 in tour:                  # Step 2
    for t2 in tour.around(t1):   # Step 3
        x1 = (t1, t2)
        X = set(x1)
        for t3 in neighbours[t2]:  # Step 4
            y1 = (t2, t3)
            Y = set(y1)
            gain = c(x1) - c(y1)
            if gain > 0:
                if self.chooseX(tour, t1, t3, gain, broken, joined): # Step 6
                    # Return to Step 2, that is the initial loop
                    return True
            # Else try the other options, note how we retain X and Y with the
            # right data. (Step 8-12)
# No improvement found
return False
```

Listing 9. Lin-Kernighan: Move Selection Loop; as described by Maheo [41].

The rest of the pseudocode along with the detailed description of the implementation can be found in Maheo [42] and won't be replicated in this work, since we refer to his implementation entirely.

The results of running the implemented Lin-Kernighan algortihm can be seen in 21.



(a) Schematic representation.

(b) Graph representation.

Figure 21. Path constructed by the Lin-Kernighan algorithm. Path cost: 84.

4.2.5 TSP Solvers: Google OR-Tools and Concorde TSP

The integration of the two TSP solvers was done in a similar way to the other algorithms in order to minimize the runtime overhead. Another goal was to completely integrate the solvers with the main application so that the same output format could be achieved for all the algorithms and solvers.

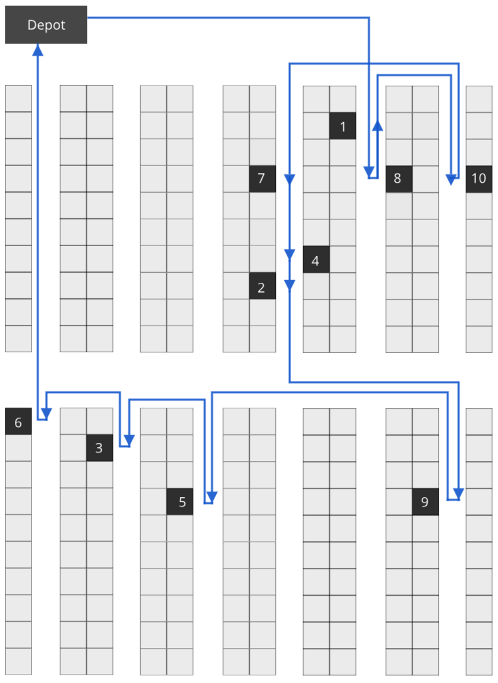
For Google OR-Tools we based our integration on the official example [44] provided by Google. The only change was that all of the tool's setup was placed in the constructor of the GoogleOR class and the actual TSP solver was called in the `optimize()` function. In this TSP solver the local search algorithm and first solution strategy can be selected from a list of available heuristics. For more details see *Google OR-Tools* [45].

Concorde TSP is slightly different from the rest of the solvers in several aspects. Firstly, because instead of calling a library we used the command line tool. Also, unlike the rest of the solvers, where the entire `_optimize()` method is timed to receive runtime results, for Concorde TSP the runtime values were provided by the tool itself with slight modifications to the Concorde source code.

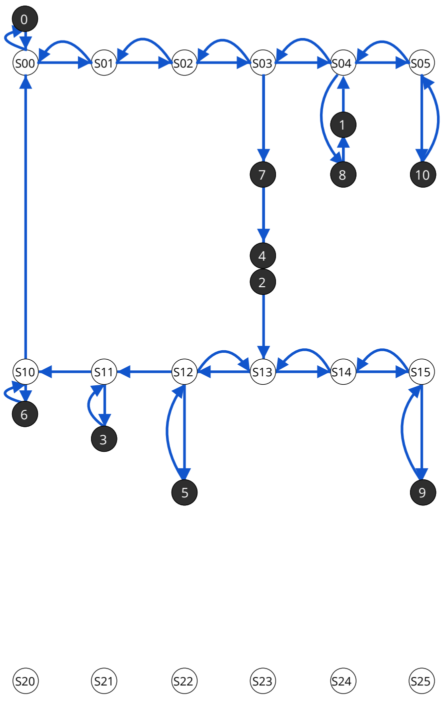
The modifications mainly affected the format of the runtime result, changing it from showing the runtim in seconds to nanoseconds. In order to achieve that, the timing function was changed to a more modern one that supported timing in nanoseconds.

To support all different test cases considered in our simulation, different Concorde TSP executables were used. This is due to the limitations of the Concorde software. Lin-Kernighan executable (`linkern`) only supports graphs that contain over 10 nodes. For that reason in tests where graphs did not contain enough nodes the `concorde` executable was used.

Finally, one regular expression is used to parse the output of each executable in order to store the runtime results.



(a) Schematic representation.



(b) Graph representation.

Figure 22. Path constructed by Google OR-Tools TSP. Path cost: 84.

4.3 Data Output

Output of the algorithms is a single `.csv` file that contains the following data

- test parameters:
  - random seed: seed
  - # of cross aisles: cross_aisles
  - # of aisles: aisle
  - # of rows: rows
  - # of orders: orders
  - storage policy: policy
  - # of aisles of type A (or %): a_capacity,
  - # of aisles of type B (or %): b_capacity
- simulation results:
  - which algorithm: algorithm
  - found route: route
  - route length: distance
  - run time: runtime

This data is generated for all of the test cases and processed using Google Sheets.

5 Results and Analysis

The experimental design consists of several factors: routing algorithms, storage policies, number of orders, and parameters defining the warehouse layout. The routing algorithms factor includes Nearest Neighbour, 2-opt, Lin-Kernighan (LK) algorithms, and TSP solvers: Google OR-Tools and Concorde TSP. The storage policy factor consists of random, within-aisle, and across-aisle storage policies. The parameters of the warehouse vary as shown in Table 3. The `a_capacity` and `b_capacity` for the ABC policy are defined manually for each of the values of the `aisle` parameter (Table 3b). For ABCACROSS these parameters are fixed and set to `a_capacity = 0.1` and `b_capacity = 0.2`.

Table 3. Warehouse parameters used in the simulation.

Cross-Aisles	Aisles	Rows	Orders	Storage Policy
1	4	50	5	RANDOM
3	8	100	7	ABC
5	12	500	20	ABCACROSS
11	20		50	

(a) Test Parameters.

Aisles	a_capacity	b_capacity
4	1	2
8	1	3
12	2	4
20	4	8

(b) `a_capacity` and `b_capacity`.

All possible combinations of these parameters are found and each combination defines a single test case. To gain insight into the performance of the different heuristics presented in Section 4 we set up a large simulation experiment based on these test instances.

Each of the algorithms was applied to the generated 576 test cases, so that in total 2880 test configurations were obtained, which were each replicated 10 times for statistical significance using different seed values for the random number generator. In all of the settings, the length between two consecutive pick aisles and the width of each cross aisle is equal to 1. For each of these situations, we receive the route found by each algorithm, travel distance, and runtime values (in nanoseconds) by the means of simulation.

Computational testing was done on a 2,7 GHz Dual-Core Intel Core i5 processor with 8 GB 1867 MHz RAM. Results were saved as a `csv` file as described in Section 4.3 and they are analyzed using Google Sheets online spreadsheets platform.

This section will discuss several aspects of the simulation results. First, we will address the general trends of each algorithm on all of the generated test cases, by comparing the route length calculated. Next, the impact on the runtime and route cost of varying a single parameter in a fixed warehouse configuration is analyzed and compared for different algorithms. Results of the simulation are shown in the form of tables, charts and plots.

It can be seen from Table 4 that Concorde TSP is the algorithm that finds the lowest cost routes in most of the test cases, specifically 98.26%. This fact is the reason that further in the analysis Concorde will be referred to as the baseline algorithm in terms of minimal route cost. On the other hand, we can also see that the algorithm that hasn't produced any minimal results and produced the worst route costs in over 75% of total test cases is the Greedy (Nearest-Neighbour) algorithm. This algorithm is also the one that has the highest deviation values, making it the least optimal among the tested algorithms.

Lin-Lernighan (LK) and Google OR-tools are both producing near baseline results, with 0.67% average deviation for LK and half that at 0.3% for Google OR. However, LK is not as consistent at achieving the optimal results (optimal refers to the best result among the algorithms), while Google OR-tools achieve them in almost half of the test instances.

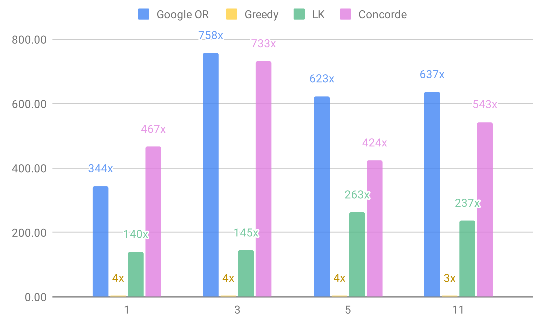
When evaluating these average results, it is worth noting that the 2-opt algorithm does not stand out as most or least optimal (when comparing to Concorde), generating average results with a deviation of about 7%. These values, however, are insufficient for us to draw any conclusions, so we will go through each algorithm in more depth later on.

Table 4. General Statistics.

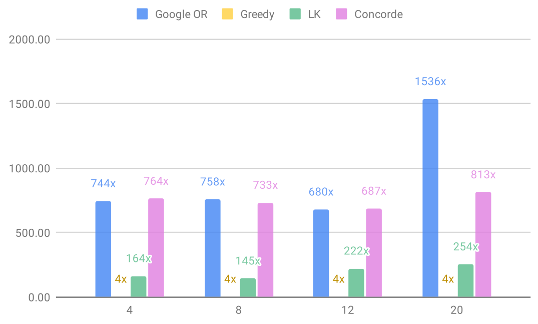
Algorithm	AVG deviations from baseline (%)	% of most baseline results	% of least baseline results
Greedy	10.92%	-	76.22%
2-opt	7.53%	0.35%	23.96%
LK	0.67%	9.55%	-
OR-tools	0.30%	49.65%	-
Concorde	0.00%	98.26%	-

We'd like to note that in order to evaluate the influence of the parameters on the performance and calculated route lengths we selected a base test configuration with *3 blocks, 8 aisles, 100 rows, 7 orders and Random policy* selected as the storage policy. We then changed a single parameter in this configuration to receive the results discussed further in this section.

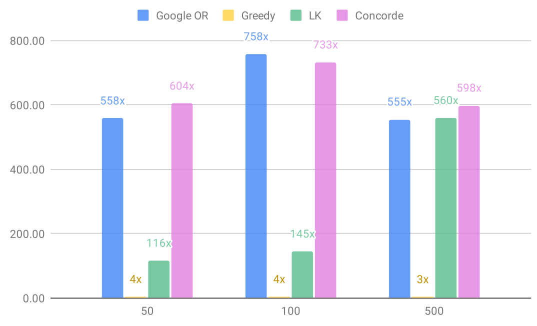
In terms of runtime 2-opt shows best results and for that reason it is used as a baseline for evaluating the rest of the algorithms. We may refer to it as "optimal" in the context of this work. For each algorithm Table 6 shows the performance decrease value of that algorithm when compared to the performance of 2-opt on the same test case and Table 7 shows the actual runtime values rounded to ms (10^{-3}s). These performance decreases are also visualised by charts Figure 23 for clearer view of peaks and trends.



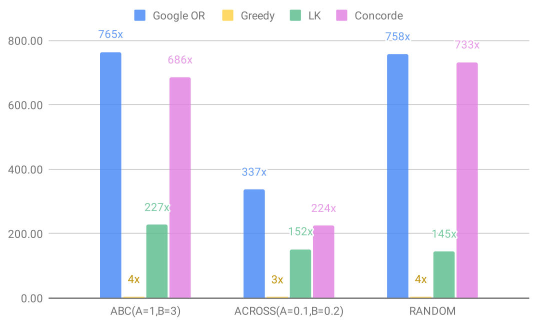
(a) Number of blocks.



(b) Number of aisles.



(c) Number of rows.



(d) Type of the storage policy.

Figure 23. Bar Charts illustrating runtime decrease of the TSP solvers and algorithms when compared to 2-opt with different changing parameters.

The first highlight of the runtime results is the consistently low degradation of the Greedy algorithm. This is opposed to the previously noted trend of this algorithm producing the longest routes, making it less of an interest for us. At the same time, Lin-Kernighan on average performs better than Google OR and Concorde except for the two cases where a larger number of orders is defined. On the same test cases when changing the number of orders from 20 to 50 we notice thatperformance degradation for all three of the other algorithms drops around 60%, implying that these algorithms appear to handle the growing number of orders more efficiently than LK and 2-opt. It also expands to the other values of `orders` parameter, meaning thatperformance decrease consistently reduces with the increase of the number of orders.

These results can be seen more clearly from Table 5 which shows that Greedy, Google and Concorde scale in a more stable manner when increasing the number of orders than 2-opt. LK shows over 17.4 times increase in runtime when orders number is changed from 7 to 20, compared to only 1.8 time for Google OR and 2.1 times for Concorde. We relate this large difference in the runtime degradation to the fact that Lin-Kernighan algorithm was implemented in an inefficient manner and should be further optimized in terms of memory usage and data structure choice. This Algorithm is more complex than Nearest-Neighbour and 2-opt algorithms, as was described in Section 4.2.4 and therefore requires a more careful approach in implementation to achieve better performance.

Table 5. Performance changes with the increase in the number of orders.

Orders change	2-opt	Google OR	Greedy	LK	Concorde
from 5 to 7	1.18x	0.42x	0.90x	3.05x	1.19x
from 7 to 20	7.75x	3.85x	1.83x	17.4x	2.18x
from 20 to 50	13.67x	4.51x	3.12x	20.7x	5.45x

Another notable fact from this table and from the original runtime results is that when the storage policy is changed from random to any of the two class-based policies, the runtimes increase for all of the algorithms except for Concorde, drops by 23% when the storage policy is set to ABCACROSS instead of Random.

Table 6. Runtime decreases compared to 2-opt. In bold: smallest, in italic: largest degradation values.

Cross Aisles	Aisles	Rows	Orders	Storage Polic	Google OR	Greedy	LK	Concorde
1	8	100	7	RANDOM	344.3x	**4.2x**	140.1x	*466.7x*
3	8	100	7	RANDOM	*758.0x*	**4.0x**	144.7x	732.5x
5	8	100	7	RANDOM	*623.4x*	**4.1x**	262.8x	424.4x
11	8	100	7	RANDOM	*636.6x*	**3.4x**	236.9x	543.4x
3	**4**	100	7	RANDOM	744.3x	**3.9x**	163.8x	*763.5x*
3	**8**	100	7	RANDOM	*758.0x*	**4.0x**	144.7x	732.5x
3	**12**	100	7	RANDOM	679.9x	**3.8x**	222.1x	*686.6x*
3	**20**	100	7	RANDOM	*1535.8x*	**4.3x**	254.5x	813.2x
3	8	**50**	7	RANDOM	558.5x	**4.0x**	116.2x	*604.4x*
3	8	**100**	7	RANDOM	*758.0x*	**4.0x**	144.7x	732.5x
3	8	**500**	7	RANDOM	555.1x	**3.4x**	559.7x	*598.3x*
3	8	100	**5**	RANDOM	*2132.5x*	**5.2x**	24.9x	727.9x
3	8	100	**7**	RANDOM	*758.0x*	**4.0x**	133.6x	732.5x
3	8	100	**20**	RANDOM	*376.4x*	**0.9x**	300.6x	206.1x
3	8	100	**50**	RANDOM	124.1x	**0.2x**	*454.5x*	82.1x
3	8	100	7	**ABC**	*765.1x*	**4.0x**	227.4x	685.8x
3	8	100	7	**ACROSS**	*337.1x*	**2.8x**	152.3x	224.4x
3	8	100	7	**RANDOM**	*758.0x*	**4.0x**	144.7x	732.5x

Table 7. Runtime results in ms (10^{-3} s). In bold: smallest, in red italic: largest runtime values.

Cross Aisles	Aisles	Rows	Orders	Storage Policy	2-opt	Google OR	Greedy	LK	Concorde
1	8	100	7	RANDOM	**0.034**	11.580	0.140	4.713	*15.694*
3	8	100	7	RANDOM	**0.024**	*18.152*	0.095	3.465	17.542
5	8	100	7	RANDOM	**0.032**	*20.062*	0.132	8.458	13.657
11	8	100	7	RANDOM	**0.025**	*16.164*	0.087	6.016	13.799
3	4	100	7	RANDOM	**0.026**	18.981	0.101	4.178	*19.471*
3	8	100	7	RANDOM	**0.024**	*18.152*	0.095	3.465	17.542
3	12	100	7	RANDOM	**0.022**	15.220	0.086	4.971	*15.370*
3	20	100	7	RANDOM	**0.025**	*38.139*	0.106	6.319	20.193
3	8	50	7	RANDOM	**0.023**	13.073	0.094	2.720	*14.148*
3	8	100	7	RANDOM	**0.024**	*18.152*	0.095	3.465	17.542
3	8	500	7	RANDOM	**0.027**	14.950	0.091	15.073	*16.114*
3	8	100	5	RANDOM	**0.020**	*43.102*	0.106	0.503	14.712
3	8	100	7	RANDOM	**0.024**	*18.152*	0.095	3.200	17.542
3	8	100	20	RANDOM	0.186	*69.874*	**0.174**	55.811	38.260
3	8	100	50	RANDOM	2.538	314.880	**0.544**	*1153.59*	208.352
3	8	100	7	ABC	**0.026**	*20.219*	0.105	6.010	18.125
3	8	100	7	ACROSS	**0.060**	*20.319*	0.169	9.179	13.529
3	8	100	7	RANDOM	**0.024**	*18.152*	0.095	3.465	17.542

The overall deviation trends in Table 9 correspond to the results discussed earlier based on the general deviation trends (Table 4 with Google OR results deviating the least from baseline and LK results slightly worse. The results shown in this table and in Table 8 will be analyzed in more detail with respect to the influence of changing of each of the warehouse parameters.

Starting at the bottom of Table 9 and from Figure 24 it can be seen that change in storage policy affects the four algorithms differently. While 2-opt and Greedy algorithms show more than twice improvement in results when the policy is changed from Random to any of the two class-based policies, LK and Google OR show slight decrease in precision. Specifically, Nearest-Neighbour algorithm goes down from 14% deviation when Random policy is set to 7.87% with ABCACROSS and even further down with ABC at 3.29%.

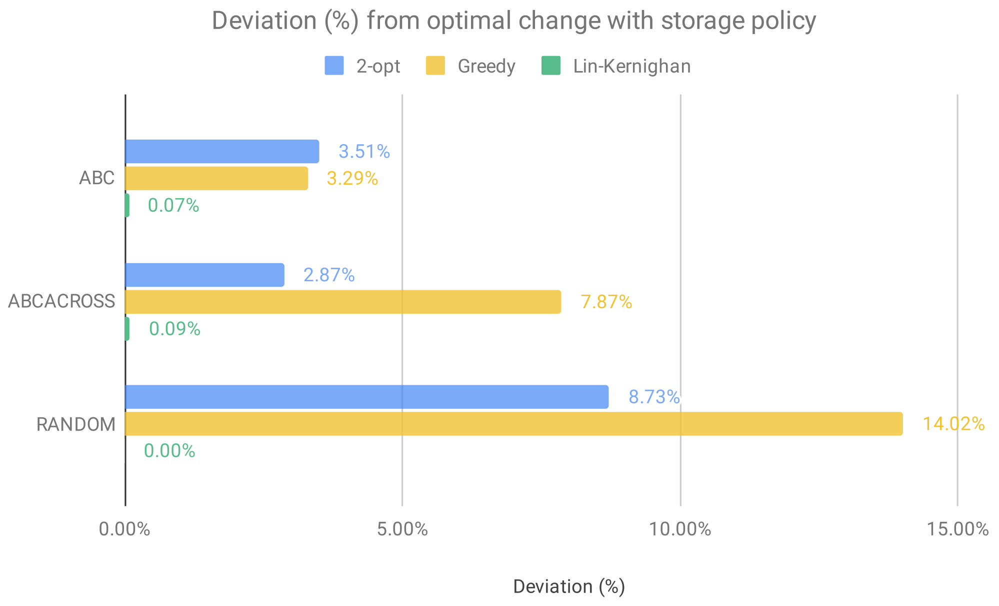


Figure 24. Deviation changes of the TSP solvers and algorithms with the change of storage policy type.

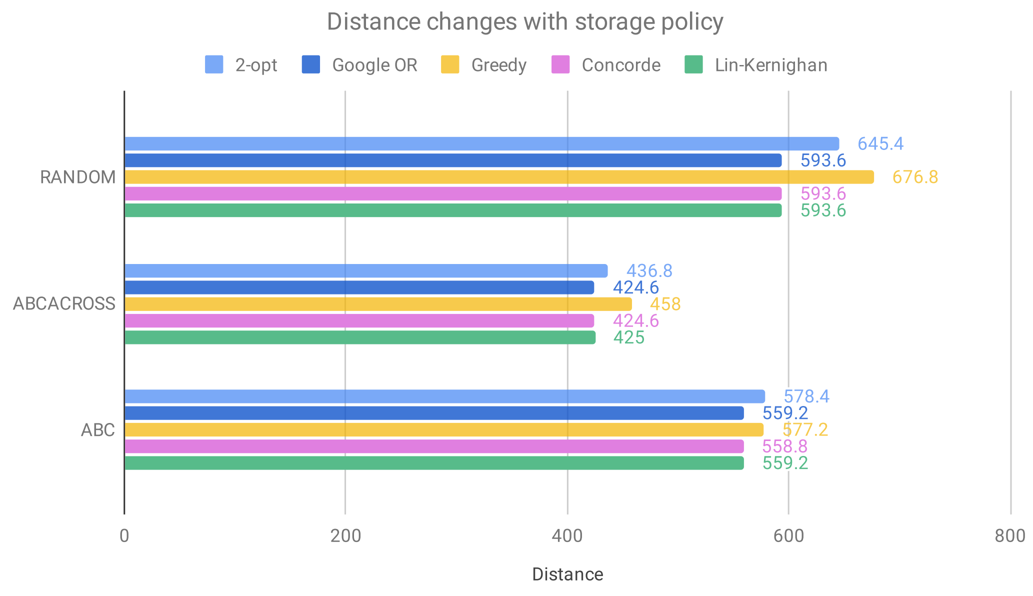


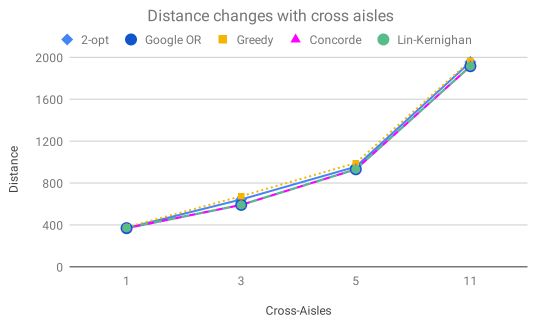
Figure 25. Distance changes of the TSP solvers and algorithms with the change of storage policy type.

Google OR is not affected by the change in neither cross-aisles, aisles nor rows, while other algorithms are significantly more sensitive to these changes but don't follow specific trends. However, when analyzing the influence of the number of orders we can see that the general trend is that with the increase of that number rises the deviation, because of the fact that the number of orders (nodes) is the main parameter adding complexity to the traveling salesman problem.

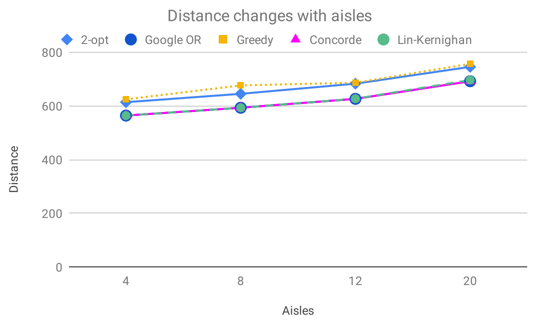
Interestingly, LK shows better result than Google OR on the largest tested number of orders (50). Here the difference is at almost 1% while in most other cases this value never exceeds 0.1 except for two more cases: when the number of blocks is only 1 and when the number of rows is equal to 50. This leads us to a conclusion that LK generally finds better routes in larger warehouse cases. For that reason, we consider optimizing the implementation of LK algorithm in order to improve the performance results, making it the best choice of algorithm in the context of this work.

At the same time the results of the Greedy algorithm are consistent with its overall bad trends: the average deviation remains at around 14% while the worst deviation can be noticed in the case of largest order number at almost 20%.

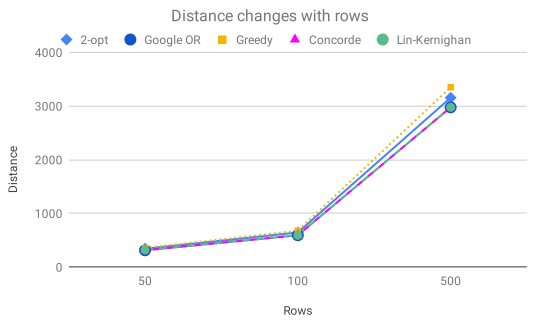
Overall, each of the algorithms in the current implementation has advantages and disadvantages.



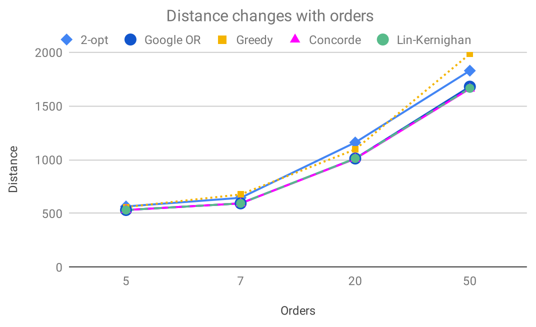
(a) Number of blocks.



(b) Number of aisles.



(c) Number of rows.

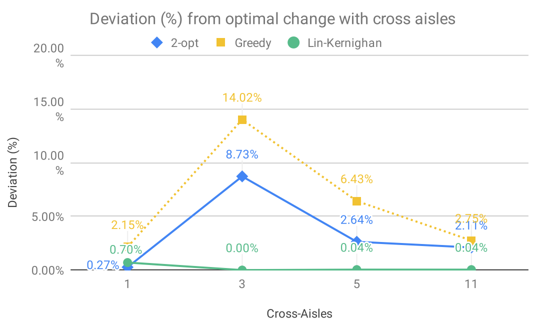


(d) Number of orders.

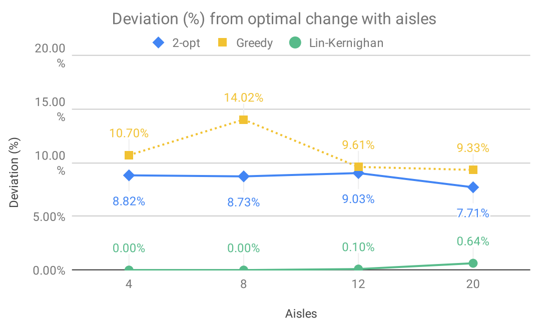
Figure 26. Plots illustrating distance changes of the TSP solvers and algorithms with different changing parameters.

Table 8. Route Length (Distance) values of each algorithm on the selected test cases. In bold: smallest, in italic: largest distance values.

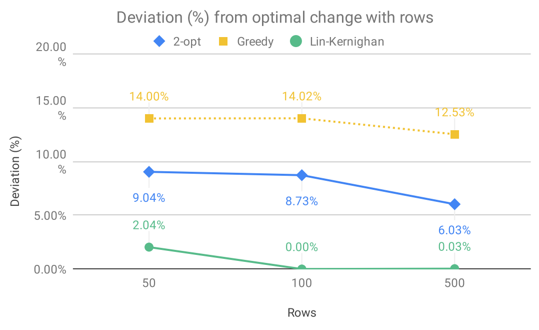
Cross Aisles	Aisles	Rows	Orders	Storage Policy	2-opt	Google OR	Greedy	LK	Concorde	**Min route**	*Max route*
1	8	100	7	RANDOM	372.4	**371.4**	*379.4*	374	**371.4**	**371.4**	*379.4*
3	8	100	7	RANDOM	645.4	**593.6**	*676.8*	**593.6**	**593.6**	**593.6**	*676.8*
5	8	100	7	RANDOM	958	**933.4**	*993.4*	933.8	**933.4**	**933.4**	*993.4*
11	8	100	7	RANDOM	1957.8	**1917.4**	*1970.2*	1918.2	**1917.4**	**1917.4**	*1970.2*
3	**4**	100	7	RANDOM	614.2	**564.4**	*624.8*	**564.4**	**564.4**	**564.4**	*624.8*
3	**8**	100	7	RANDOM	645.4	**593.6**	*676.8*	**593.6**	**593.6**	**593.6**	*676.8*
3	**12**	100	7	RANDOM	683.2	**626.6**	*686.8*	627.2	**626.6**	**626.6**	*686.8*
3	**20**	100	7	RANDOM	745.8	**692.4**	*757*	696.8	**692.4**	**692.4**	*757*
3	8	**50**	7	RANDOM	342.6	**314.2**	*358.2*	320.6	**314.2**	**314.2**	*358.2*
3	8	**100**	7	RANDOM	645.4	**593.6**	*676.8*	**593.6**	**593.6**	**593.6**	*676.8*
3	8	**500**	7	RANDOM	3155.2	**2975.8**	*3348.8*	2976.8	**2975.8**	**2975.8**	*3348.8*
3	8	100	**5**	RANDOM	*565.2*	**531.8**	559.2	**531.8**	**531.8**	**531.8**	*565.2*
3	8	100	**7**	RANDOM	645.4	**593.6**	*676.8*	**593.6**	**593.6**	**593.6**	*676.8*
3	8	100	**20**	RANDOM	*1160.8*	1011.4	1096.4	1011.8	**1009**	**1009**	*1160.8*
3	8	100	**50**	RANDOM	1829.4	1680.8	*1985.6*	1665.8	**1662**	**1662**	*1985.6*
3	8	100	7	**ABC(1,3)**	*578.4*	559.2	577.2	559.2	**558.8**	**558.8**	*578.4*
3	8	100	7	**ACROSS(0.1,0.2)**	436.8	**424.6**	*458*	425	**424.6**	**424.6**	*458*
3	8	100	7	**RANDOM**	645.4	**593.6**	*676.8*	**593.6**	**593.6**	**593.6**	*676.8*



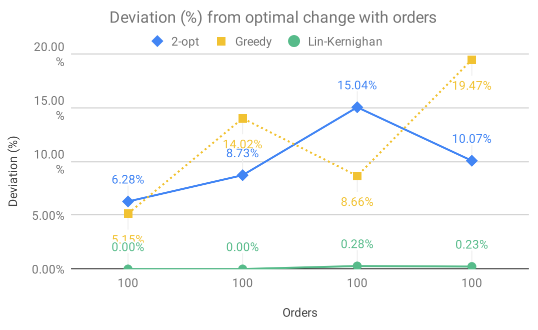
(a) Number of blocks.



(b) Number of aisles.



(c) Number of rows.



(d) Number of orders.

Figure 27. Plots illustrating deviation of the TSP solvers and algorithms with different changing parameters.

Table 9. Route Length Deviation values of each algorithm on the selected test cases. In bold: smallest, in italic: largest deviation values.

Cross Aisles	Aisles	Rows	Orders	Storage Policy	2-opt	Google OR	Greedy	LK	**Min deviation %**	*Max deviation %*
1	8	100	7	RANDOM	**0.27%**	0.00%	*2.15%*	0.70%	0.27%	*2.15%*
3	8	100	7	RANDOM	8.73%	**0.00%**	*14.02%*	**0.00%**	0.00%	*14.02%*
5	8	100	7	RANDOM	2.64%	0.00%	*6.43%*	**0.04%**	0.04%	*6.43%*
11	8	100	7	RANDOM	2.11%	0.00%	*2.75%*	**0.04%**	0.04%	*2.75%*
3	**4**	100	7	RANDOM	8.82%	**0.00%**	*10.70%*	**0.00%**	0.00%	*10.70%*
3	**8**	100	7	RANDOM	8.73%	**0.00%**	*14.02%*	**0.00%**	0.00%	*14.02%*
3	**12**	100	7	RANDOM	9.03%	0.00%	*9.61%*	**0.10%**	0.10%	*9.61%*
3	**20**	100	7	RANDOM	7.71%	0.00%	*9.33%*	**0.64%**	0.64%	*9.33%*
3	8	**50**	7	RANDOM	9.04%	0.00%	*14.00%*	**2.04%**	2.04%	*14.00%*
3	8	**100**	7	RANDOM	8.73%	**0.00%**	*14.02%*	**0.00%**	0.00%	*14.02%*
3	8	**500**	7	RANDOM	6.03%	0.00%	*12.53%*	**0.03%**	0.03%	*12.53%*
3	8	100	**5**	RANDOM	*6.28%*	**0.00%**	5.15%	**0.00%**	0.00%	*6.28%*
3	8	100	**7**	RANDOM	8.73%	**0.00%**	*14.02%*	**0.00%**	0.00%	*14.02%*
3	8	100	**20**	RANDOM	*15.04%*	0.24%	8.66%	**0.28%**	0.28%	*15.04%*
3	8	100	**50**	RANDOM	10.07%	1.13%	*19.47%*	**0.23%**	0.23%	*19.47%*
3	8	100	7	**ABC(1,3)**	*3.51%*	**0.07%**	3.29%	**0.07%**	0.07%	*3.51%*
3	8	100	7	**ACROSS(0.1,0.2)**	2.87%	0.00%	*7.87%*	**0.09%**	0.09%	*7.87%*
3	8	100	7	**RANDOM**	8.73%	**0.00%**	*14.02%*	**0.00%**	0.00%	*14.02%*

5.1 Case Study: Kuehne + Nagel Warehouse

The following warehouse layout characteristics were provided by Kuehne + Nagel:

- Warehouse dimensions: 197200 sf (340 feet by 560 feet)
- Parallel-aisle warehouse for the Picking operation
- Number of aisles: 23 (2 are end aisles ’one-sided’, 21 ’full’ aisles with shelving on both sides)
- Number of blocks: 3; blocks may vary in length
- Number of locations per subaisle: 500; so each end aisle has 500 and each full aisle has 1000 pick locations
- Average order list size: Ecomm @ 1.1 units/order; Store orders @ 7 units/order on Average
- Storage Policy: Random
- Routing: Use robot technology to bring order to pick location; picker uses serpentine path to locate robots in their zone and picks robots needs as they cross path.

This gives us the following configuration:

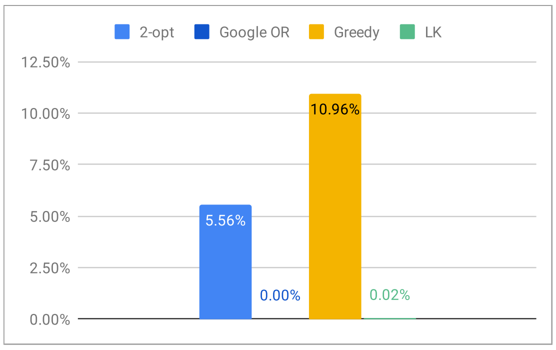
```
#For Kuehne+Nagel
BLOCKS = [3]
AISLES = [23]
ROWS = [500]
STORAGE_POLICY = ['RANDOM']
ORDERS = [7]
ABC_CAPACITY = [(3, 5)]
ABC_ACROSS_CAPACITY = [(0.1, 0.2)]
SEEDS = list(range(10))
```

Listing 10. Global array values set in `generate.py` to generate the test values for the Kuehne + Nagel case.

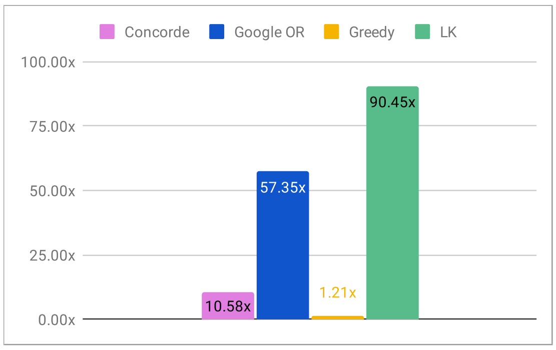
For this specific configuration we summarized the results in the following form: the average route length and average runtime in microseconds are depicted in Table 10. From this table we can derive that 2-opt shows the best performance, similarly to the results achieved in the main simulation. Based on this fact Figure 28b represents the decrease of the other four algorithms when compared to the 2-opt algorithm’s runtime. The Figure 28a shows the deviation of the route costs found by the four algorithms besides Concorde (that remains the baseline of the evaluation).

We can see controversial results from these two figures. While Greedy shows the lowest performance decrease from the 2-opt algorithm it also shows the highest deviation from the baseline at around 11%, which makes it inappropriate for this case. On the contrary, LK shows near baseline results with deviation of only 0.02% but the performance of this algorithm is the worst of all the rest.

This analysis leads us to comparing the remaining two algorithms: 2-opt and Concorde. The advantage of the Concorde solver is that it achieves minimal results in most of the cases, including the Kuehne + Nagel case. However, it is over 10 times slower than 2-opt. On the other side, 2-opt is the fastest of all the tested algorithms but it produces routes with over 5% increase in cost.



(a) Deviation from baseline route length.



(b) Performance performance decrease compared to 2-opt.

Table 10. Run results for the Kuehne + Nagel case.

	2-opt	Concorde	Google OR	Greedy	LK
Route Length (AVG)	3378.8	**3200.8**	**3200.8**	3551.6	3201.6
Runtime (mcs, AVG)	**1.552705**	16.431	89.045872	1.876476	140.447075

However, for both of these TSP solving options, the disadvantages are almost negligible. In case of the performance decrease of the Concorde, solver when looking at the actual numbers, we can see that the difference in runtime is only 15 microseconds, which is insignificant for a human user or even some machines.

On other hand, the 5% increase in route length as calculated by the 2-opt algorithm is, in our case, only at around 160 units, which in a large warehouse may or may not be an important decision point since the total distances are considerably larger. It would, influence the time travel time and if a slow machine or a human picker are performing the task this value might be more significant.

Another important point we'd like to make is that in the current layout, this warehouse allows blocks to be of different lengths and the order picking is automated with robot systems that may start and end their routes anywhere in the warehouse. These two aspects of the warehouse are not yet considered in the current implementation of the algorithms, but are part of our future work.

6 Summary

Selecting a suitable algorithm for solving the warehouse routing problem strongly depends on the requirements and parameters of the given warehouse. We have derived the strong connection between the performance and the results of each of the tested algorithms and the warehouse parameters, especially with the number of orders and the type of the storage policy defined in the warehouse. At the same time, some algorithms have shown better results on test cases that could characterize smaller warehouse types, while the others scaled better and showed significantly better performance when the complexity of the underlying traveling salesman problem increased with the increase of the warehouse size and number of orders.

Another important aspect that needs to be considered is the complexity of the implemented TSP algorithms and solvers. While Google OR-tools was simpler in our integration process it may cause issues for mobile platforms with certain types of processors, that are often used in the warehouse. On the other hand Concorde TSP, that produced best route options and showed stable performance on all of the tested warehouse configurations, may also be difficult in integration, but more suitable for large warehouse that accommodate appropriate devices for such integration.

Both Lin-Kernighan and 2-opt are commonly used for solving TSP problems and are easy to integrate. In the case of Lin-Kernighan, the algorithm has produced results closest to the baseline, showing worse performance results in terms of runtime. We are convinced that this problem is resolvable if more time was spent on optimizing the implementation of this algorithm.

Overall, we can see from the results of this work that using TSP algorithms to solve warehouse order picking problem is a viable solution that deserves further investigation. While the performance of the preprocessing stage wasn't considered in the evaluation, we believe that in its current state it serves its purpose and can be further modified for future use.

References

[1] G. Cormier and P. Eng, "A brief survey of operations research models for warehouse design and operation", *CORS-SCRO Bulletin*, vol. 31, no. 3, pp. 15–20, 1997.

[2] J. Gu, M. Goetschalckx, and L. F. McGinnis, "Research on warehouse operation: A comprehensive review", *European journal of operational research*, vol. 177, no. 1, pp. 1–21, 2007.

[3] M. Goetschalckx and J. Ashayeri, "Classification and design of order picking", *Logistics World*, 1989.

[4] R. De Koster, T. Le-Duc, and K. J. Roodbergen, "Design and control of warehouse order picking: A literature review", *European journal of operational research*, vol. 182, no. 2, pp. 481–501, 2007.

[5] J. Bartholdi, S. Hackman, and G. I. of Technology., *Warehouse and Distribution Science*. Supply Chain, Logistics Institute, School of Industrial, and Systems Engineering, Georgia Institute of Technology, 2008. [Online]. Available: `https://books.google.se/books?id=aNFFnQAACAAJ`.

[6] K. J. Roodbergen and R. Koster, "Routing methods for warehouses with multiple cross aisles", *International Journal of Production Research*, vol. 39, no. 9, pp. 1865–1883, 2001.

[7] C. G. Petersen and R. W. Schmenner, "An evaluation of routing and volume-based storage policies in an order picking operation", *Decision Sciences*, vol. 30, no. 2, pp. 481–501, 1999.

[8] C. G. Petersen and G. Aase, "A comparison of picking, storage, and routing policies in manual order picking", *International Journal of Production Economics*, vol. 92, no. 1, pp. 11–19, 2004.

[9] T. Le-Anh and M. De Koster, "A review of design and control of automated guided vehicle systems", *European Journal of Operational Research*, vol. 171, no. 1, pp. 1–23, 2006.

[10] C. Theys, O. Bräysy, W. Dullaert, and B. Raa, "Using a tsp heuristic for routing order pickers in warehouses", *European Journal of Operational Research*, vol. 200, no. 3, pp. 755–763, 2010.

[11] Eurostat, "Warehousing and transport support services statistics - nace rev. 2", 2015.

[12] J. D. Smith, *The warehouse management handbook*. Tompkins press, 1998.

[13] K. B. Ackerman, *Practical handbook of warehousing*. Springer Science & Business Media, 2012.

[14] K. J. Roodbergen, I. F. Vis, and G. D. Taylor Jr, "Simultaneous determination of warehouse layout and control policies", *International Journal of Production Research*, vol. 53, no. 11, pp. 3306–3326, 2015.

[15] S. T. Hackman, E. H. Frazelle, P. M. Griffin, S. O. Griffin, and D. A. Vlasta, "Benchmarking warehousing and distribution operations: An input-output approach", *Journal of Productivity Analysis*, vol. 16, no. 1, pp. 79–100, 2001.

[16] L. M. Pohl, R. D. Meller, and K. R. Gue, "Optimizing fishbone aisles for dual-command operations in a warehouse", *Naval Research Logistics (NRL)*, vol. 56, no. 5, pp. 389–403, 2009.

[17] J. A. Tompkins, J. A. White, Y. A. Bozer, and J. M. A. Tanchoco, *Facilities planning*. John Wiley & Sons, 2010.

[18] K. Choe, “Aisle-based order pick systems with batching, zoning, and sorting.”, 1992.

[19] W. H. Hausman, L. B. Schwarz, and S. C. Graves, “Optimal storage assignment in automatic warehousing systems”, *Management science*, vol. 22, no. 6, pp. 629–638, 1976.

[20] M. De Koster, E. S. Van der Poort, and M. Wolters, “Efficient orderbatching methods in warehouses”, *International Journal of Production Research*, vol. 37, no. 7, pp. 1479–1504, 1999.

[21] K. J. Roodbergen, “Storage assignment for order picking in multiple-block warehouses”, in *Warehousing in the global supply chain*, Springer, 2012, pp. 139–155.

[22] *Erasmus logistica*, `https://www.erim.eur.nl/material-handling-forum/research-education/tools`, Accessed: 2021-05-10.

[23] J. M. Jarvis and E. D. McDowell, “Optimal product layout in an order picking warehouse”, *IIE transactions*, vol. 23, no. 1, pp. 93–102, 1991.

[24] G. Cornuéjols, J. Fonlupt, and D. Naddef, “The traveling salesman problem on a graph and some related integer polyhedra”, *Mathematical programming*, vol. 33, no. 1, pp. 1–27, 1985.

[25] L. Pansart, N. Catusse, and H. Cambazard, “Exact algorithms for the order picking problem”, *Computers & Operations Research*, vol. 100, pp. 117–127, 2018.

[26] R. M. Karp, “Reducibility among combinatorial problems”, in *Complexity of computer computations*, Springer, 1972, pp. 85–103.

[27] Y. Geng, “On warehouse routing problems”, Ph.D. dissertation, 2005.

[28] H. H. Hoos and T. Stützle, *Stochastic local search: Foundations and applications*. Elsevier, 2004.

[29] C. H. Papadimitriou, “On the complexity of unique solutions”, *J. ACM*, vol. 31, no. 2, pp. 392–400, 1984-03, ISSN: 0004-5411. DOI: `10.1145/62.322435`. [Online]. Available: `https://doi.org/10.1145/62.322435`.

[30] L. Pansart, N. Catusse, and H. Cambazard, “Exact algorithms for the picking problem: Thesis”, 2016.

[31] G. Dantzig, R. Fulkerson, and S. Johnson, “Solution of a large-scale traveling-salesman problem”, *Journal of the operations research society of America*, vol. 2, no. 4, pp. 393–410, 1954.

[32] L. B. Schwarz, S. C. Graves, and W. H. Hausman, “Scheduling policies for automatic warehousing systems: Simulation results”, *AIIE transactions*, vol. 10, no. 3, pp. 260–270, 1978.

[33] D. R. Gibson and G. P. Sharp, “Order batching procedures”, *European journal of operational research*, vol. 58, no. 1, pp. 57–67, 1992.

[34] H. D. Ratliff and A. S. Rosenthal, “Order-picking in a rectangular warehouse: A solvable case of the traveling salesman problem”, *Operations research*, vol. 31, no. 3, pp. 507–521, 1983.

[35] R. De Koster and E. Van der Poort, “Routing orderpickers in a warehouse: A comparison between optimal and heuristic solutions”, *IIE transactions*, vol. 30, no. 5, pp. 469–480, 1998.

[36] R. W. Hall, “Distance approximations for routing manual pickers in a warehouse”, *IIE transactions*, vol. 25, no. 4, pp. 76–87, 1993.

[37] P. A. Makris and I. G. Giakoumakis, “K-interchange heuristic as an optimization procedure for material handling applications”, *Applied Mathematical Modelling*, vol. 27, no. 5, pp. 345–358, 2003.

[38] M. Masae, C. H. Glock, and P. Vichitkunakorn, “Optimal order picker routing in the chevron warehouse”, *IISE Transactions*, vol. 52, no. 6, pp. 665–687, 2020.

[39] *Click library*, `https://palletsprojects.com/p/click/`, Accessed: 2021-05-10.

[40] *Networkx library*, `https://networkx.org/`, Accessed: 2021-05-10.

[41] A. Maheo, *Local tsp heuristics in python*, `https://arthur.maheo.net/python-local-tsp-heuristics/`, Accessed: 2021-05-10.

[42] ——, *Implementing lin-kernighan in python*, `https://arthur.maheo.net/implementing-lin-kernighan-in-python/`, Accessed: 2021-05-10.

[43] K. Helsgaun, “An effective implementation of the lin–kernighan traveling salesman heuristic”, *European Journal of Operational Research*, vol. 126, no. 1, pp. 106–130, 2000.

[44] *Google or-tools: Tsp example*, `https://developers.google.com/optimization/routing/tsp#program1`, Accessed: 2021-05-10.

[45] *Google or-tools*, `https://developers.google.com/optimization`, Accessed: 2021-05-10.

Appendix 1 – Non-exclusive Licence

Appendix A – Class Diagram of the Java Application

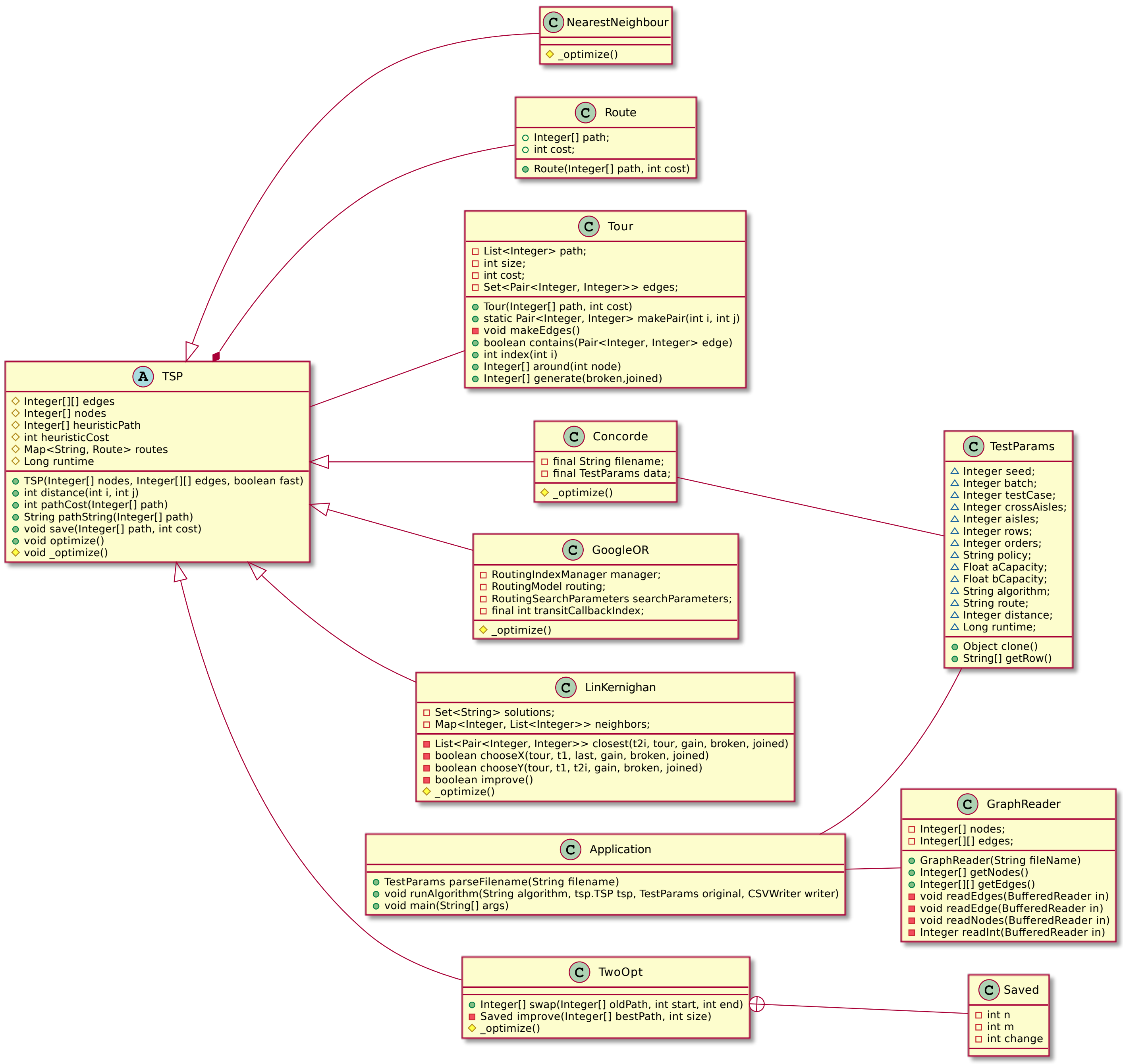

C NearestNeighbour
_optimize()
C Route
Integer[] path;
int cost;
Route(Integer[] path, int cost)
C Tour
List<Integer> path;
int size;
int cost;
Set<Pair<Integer, Integer>> edges;
Tour(Integer[] path, int cost)
static Pair<Integer, Integer> makePair(int i, int j)
void makeEdges()
boolean contains(Pair<Integer, Integer> edge)
int index(int i)
Integer[] around(int node)
Integer[] generate(broken,joined)
A TSP
Integer[][] edges
Integer[] nodes
Integer[] heuristicPath
int heuristicCost
Map<String, Route> routes
Long runtime
TSP(Integer[] nodes, Integer[][] edges, boolean fast)
int distance(int i, int j)
int pathCost(Integer[] path)
String pathString(Integer[] path)
void save(Integer[] path, int cost)
void optimize()
void _optimize()
C Concorde
final String filename;
final TestParams data;
_optimize()
C TestParams
Integer seed;
Integer batch;
Integer testCase;
Integer crossAisles;
Integer aisles;
Integer rows;
Integer orders;
String policy;
Float aCapacity;
Float bCapacity;
String algorithm;
String route;
Integer distance;
Long runtime;
Object clone()
String[] getRow()
C GoogleOR
RoutingIndexManager manager;
RoutingModel routing;
RoutingSearchParameters searchParameters;
final int transitCallbackIndex;
_optimize()
C LinKernighan
Set<String> solutions;
Map<Integer, List<Integer>> neighbors;
List<Pair<Integer, Integer>> closest(t2i, tour, gain, broken, joined)
boolean chooseX(tour, t1, last, gain, broken, joined)
boolean chooseY(tour, t1, t2i, gain, broken, joined)
boolean improve()
_optimize()
C GraphReader
Integer[] nodes;
Integer[][] edges;
GraphReader(String fileName)
Integer[] getNodes()
Integer[][] getEdges()
void readEdges(BufferedReader in)
void readEdge(BufferedReader in)
void readNodes(BufferedReader in)
Integer readInt(BufferedReader in)
C Application
TestParams parseFilename(String filename)
void runAlgorithm(String algorithm, tsp.TSP tsp, TestParams original, CSVWriter writer)
void main(String[] args)
C TwoOpt
Integer[] swap(Integer[] oldPath, int start, int end)
Saved improve(Integer[] bestPath, int size)
_optimize()
C Saved
int n
int m
int change

Figure 1. Complete Class Diagram for the Java application.

Appendix B – Implementation Source Code

GitLab Repository: https://gitlab.cs.ttu.ee/ammana/master-thesis-ttu

The structure of the repository is as following:

- `algo-maven`: contains the implementation of the TSP algorithms
- `preprocessing`: includes the entire preprocessing code
- `data`: contains data generated by preprocessing