CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Walk less, pick more: choosing optimal batches of orders in a warehouse

Master's thesis in Computer Science and Engineering

CHRISTIAN PERSSON
MARTIN SIGVARDSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

MASTER'S THESIS 2020

Walk less, pick more: choosing optimal batches of orders in a warehouse

CHRISTIAN PERSSON
MARTIN SIGVARDSSON

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Walk less, pick more: choosing optimal batches of orders in a warehouse
CHRISTIAN PERSSON
MARTIN SIGVARDSSON

Supervisor: Peter Damaschke, Computer Science and Engineering
Advisor: Johan Härdmark, Ongoing Warehouse
Examiner: Carl-Johan Seger, Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2020

Walk less, pick more: choosing optimal batches of orders in a warehouse
CHRISTIAN PERSSON
MARTIN SIGVARDSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Most warehouses employ a picker-to-parts strategy, where humans (termed *pickers*) traverse the warehouse to collect items. It is common for pickers to collect items for several orders at once. Such a set of orders is called a *batch*. Two optimization problems arise from this strategy. The picker routing problem refers to finding routes through the warehouse to minimize the distance traveled. The batching problem refers to selecting a combination of orders that minimizes the distance traveled. These problems are the focus of this thesis.

To solve the optimization problems in the context of a real-world warehouse, a graph model representing a warehouse was created. In addition, a model was created for representing orders and batches as sets of nodes in such a graph. Additionally, a collection of algorithms was designed to solve the optimization problems.

The models and the algorithms were implemented in code in the form of a library for the C# programming language. The library is accompanied by a suite of tests to help verify the correctness of the implementations. Furthermore, a suite of benchmarks was created based on real-world warehouse data supplied by the company Ongoing Warehouse. These benchmarks were used to evaluate the models and algorithms in terms of quality and runtime. Based on the evaluation, a recommendation was presented to Ongoing Warehouse of algorithms to use for integration into their warehouse management system.

Keywords: warehouse, optimization, picker routing, order batching, travelling salesperson problem, benchmarks, C#.

Acknowledgements

We would like to thank Peter Damaschke for support, guidance, and rapid feedback in the role of our supervisor. We would also like to thank Johan Härdmark, our advisor at Ongoing Warehouse, for teaching us details about warehouse operations as well as guiding us in collecting data for the benchmarks. Additionally, we want to thank Carl-Johan Seger for being our examiner. Finally, we want to thank Ongoing Warehouse for letting us use their offices as well as for the delicious shrimp sandwiches.

Christian Persson, Martin Sigvardsson, Gothenburg, July 2020

Contents

List of Figures

List of Tables

1 Introduction

Warehouses are a vital part of logistics throughout the world today. One of the most common styles of warehouse operations is the so-called picker-to-parts which essentially means that items are stored in shelves and a picker needs to walk around the shelves and isles collecting the items that are part of an order. This gives rise to the problem of picker routing. Picker routing is the problem of deciding in what order to visit each shelf and what paths to choose for traveling between the shelves while collecting all the items for an order.

To further improve the rate at which items can be picked, most warehouses equip their pickers with trolleys allowing them to collect more than one order at a time when traveling through the warehouse. This gives rise to the problem of order batching i.e., what orders to be combined to create a batch that can be picked as efficiently as possible.

Modern warehouses generally use a warehouse management system which is essentially a software application used to help manage operations on a day to day basis, keeping track of for example what items are in stock and what orders that need to be picked. Ongoing Warehouse is a company that develops a warehouse management system and as such has access to their customers' data, allowing hypothetical experiments to be performed using real data.

By using Ongoing Warehouse's insights this thesis evaluates different algorithms for the routing and batching problems with the use of real-world warehouse data.

1.1 Notation and conventions

Some mathematical notation and concepts are used repeatedly throughout this report. For consistency and clarity, the following conventions are used.

- **Graphs.** Unless specified otherwise, graphs are considered to be complete, weighted, and undirected. Graphs are defined as $G = (V, E)$ where V is the set of nodes and E is the set of edges. Nodes are generally referred to using the letter u, v, and w. The most common exceptions are the start and end nodes in the optimization problems, where s refers to the start node and t refers to the end node. An edge between nodes v and u is referred to as $e_{v,u}$. The weight of the edge $e_{v,u}$ is referred to as $w(\{v, u\})$ and is a positive integer unless specified otherwise. Note that since graphs are generally undirected, it holds that $e_{v,u} = e_{u,v}$ and $w(\{v, u\}) = w(\{u, v\})$.

- **Orders and batches.** In the context of the batch optimization problem (described below in Section 1.2.3), the set $\mathcal{L}$ refers to the set of available orders. $\mathcal{L}$ is defined in the context of a graph $G = (V, E)$. Each order in $\mathcal{L}$ is a subset of V. A batch $\mathcal{B}$ is a set of orders, and therefore $\mathcal{B} \subseteq \mathcal{L}$. The set $\mathcal{L}_N$ refers to the set of subsets of $\mathcal{L}$ with size N. In other words, $\mathcal{L}_N$ is the set of batches with size N.

1.2 Problem statement

The thesis treats two optimization problems: the *s-t shortest Hamiltonian path problem* and the *batch optimization problem.* The following sections first give an informal description of them and the connection to warehouse operations, as well as formal definitions of them.

1.2.1 Informal descriptions

In warehouse management terms, an *order* is a collection of items that are to be picked and then packed together before being sent to a customer. As such, an order represents an indivisible unit of work for a picker—they have to pick all items in the order during a single tour of the warehouse. The pickers are assumed to start their tour at some fixed location, pick all items, and then deliver the picked items to some fixed location (which can be different from where they started).

An order can be represented as a set L of locations in the warehouse, at which the items of the order can be picked. The picker travels along some path, visiting each location in L exactly once to pick the required items. In this context, the *s-t* shortest Hamiltonian path problem (henceforth simply referred to as the *shortest Hamiltonian path problem*) can be described as the problem of finding a tour of minimum length that starts in s, visits all locations in L exactly once, and ends in t. The name relates to the concept of *Hamiltonian paths*, which are paths that visit each node in a graph exactly once.

Furthermore, it is common for pickers to use trolleys with N compartments when picking orders. Each compartment is assumed to have sufficient capacity to hold all items in a single order, allowing a picker to pick up to N orders simultaneously during a single tour of the warehouse. Such a set of orders is called a *batch.* In most cases, picking a batch of N orders during a single tour yields a shorter total distance traveled compared to doing N tours picking a single order at a time. In this context, the batch optimization problem can be described as finding a batch $\mathcal{B}$ of exactly N orders, where the orders are selected from a set $\mathcal{L}$ of available orders, such that the distance of a single tour picking all orders in $\mathcal{B}$ is minimal among all possible batches.

1.2.2 Shortest Hamiltonian path problem

The shortest Hamiltonian path problem is formally defined as follows. Given are the following:

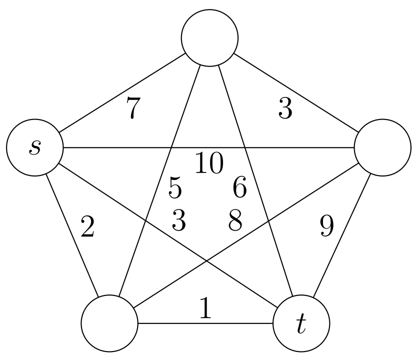
- A complete, undirected graph $G = (V, E)$, where V is the set of vertices, and E is the set of edges.

- Edge weights $w(\{v, u\})$ for all $v, u \in V$.
- Two vertices $s, t \in V$.

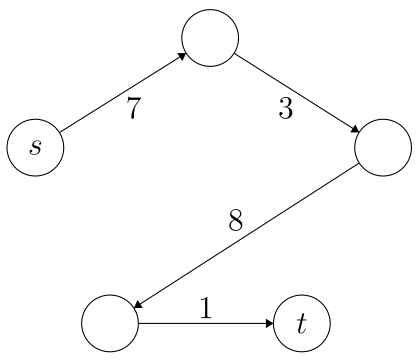
Let $V' = V \setminus \{s, t\}$ and let $\Pi(V')$ be the set of permutations of V'. The shortest Hamiltonian path problem is then a minimization problem defined as finding

$$\min_{(v_1,\dots,v_n)\in\Pi(V')} \left(w(\{s, v_1\}) + \left(\sum_{i=1}^{n-1} w(\{v_i, v_{i+1}\}) \right) + w(\{v_n, t\}) \right).$$

Figure 1.1 contains an example instance of the shortest Hamiltonian path problem on a graph with 5 nodes. Figure 1.1a contains the complete, weighted, and undirected graph, as well as the start and end nodes. Figure 1.1b contains the optimal solution to the example instance, showing the path through the graph with minimum length.



(a) The graph in the instance. s and t denote the start and end nodes, respectively.



(b) The optimal solution, with a length of 19.

Figure 1.1: Example instance of the shortest Hamiltonian path problem, along with its optimal solution.

1.2.3 Batch optimization problem

The batch optimization problem is formally defined as follows. Given are the following:

- A complete, undirected graph $G = (V, E)$, where V is the set of vertices, and E is the set of edges.
- Edge weights $w(\{v, u\})$ for all $v, u \in V$.
- Two vertices $s, t \in V$.
- A set $\mathcal{L}$ of subsets of V', where $V' = V \setminus \{s, t\}$.
- A positive integer N.

Let $\mathcal{L}_N$ be the set of subsets with size N of $\mathcal{L}$. More precisely, $\mathcal{L}_N$ is defined as

$$\mathcal{L}_N = \{\mathcal{B} \mid \mathcal{B} \subseteq \mathcal{L}, |\mathcal{B}| = N\}.$$

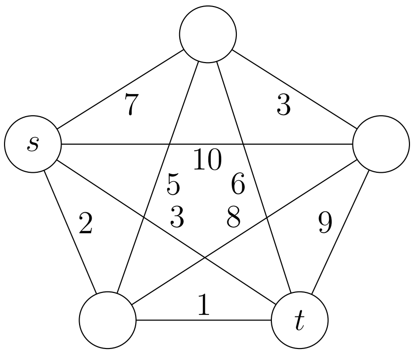
Define a function U that given a set of sets returns the union of the sets. More precisely, U is defined as

$$U(\mathcal{S}) = \bigcup_{S \in \mathcal{S}} S.$$

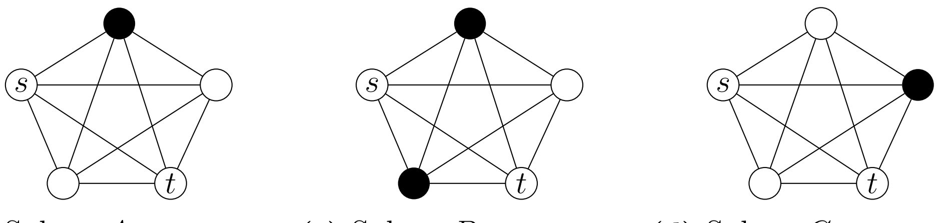
Finally, for any $Q \subseteq V$ such that $s, t \in Q$, define $G[Q]$ to be the induced subgraph of G by Q. Then define $D^*(Q)$ to be the minimum distance of a path starting in s, visiting each vertex in $G[Q]$ exactly once, and ending in t. Equivalently, $D^*(Q)$ is the length of an optimal solution to the instance of the shortest Hamiltonian path problem on the graph $G[Q]$, the weight function w, and the vertices s, t. The batch optimization problem is then a minimization problem defined as finding

$$\min_{\mathcal{B} \in \mathcal{L}_N} D^*(U(\mathcal{B}) \cup \{s, t\}).$$

Figure 1.2 contains an example instance of the batch optimization problem where $N = 2$. The subsets depicted in Figure 1.2b, Figure 1.2b, and Figure 1.2b together form the set $\mathcal{L} = \{A, B, C\}$.



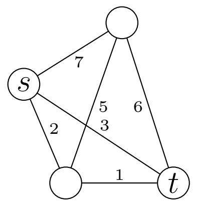
(a) The graph in the instance. s and t denote the start and end nodes, respectively.



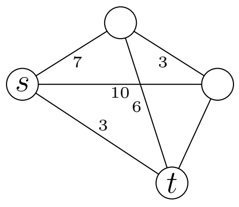
(b) Subset A. **(c)** Subset B. **(d)** Subset C.

Figure 1.2: Example instance of the batch optimization problem with $N = 2$. The shaded nodes in Figure 1.2b, Figure 1.2c, and Figure 1.2d mark the nodes that are included in the corresponding subsets.

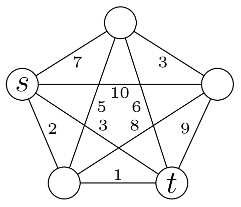
The set $\mathcal{L}_N = \mathcal{L}_2 = \{\{A, B\}, \{A, C\}, \{B, C\}\}$ is depicted in Figure 1.3, where each batch is combined with the start and end nodes, and a corresponding induced subgraph is created. The optimal solution is the batch $\{A, B\}$, depicted in Figure 1.3a.



(a) Subgraph induced by $\{A, B\}$. Minimum path length is 13. This is the optimal solution.



(b) Subgraph induced by $\{A, C\}$. Minimum path length is 19.



(c) Subgraph induced by $\{B, C\}$. Minimum path length is 19.

Figure 1.3: Subgraphs induced by each batch of size $N = 2$ combined with the start and end nodes.

1.3 Research questions

The work done in this thesis was guided by seeking answers to the following questions.

1. Can abstract models and algorithms from the existing literature on warehouse optimization, as well as combinatorial optimization, be modified, adapted, or combined to solve the optimization problems as defined in Section 1.2?

2. How well do such algorithms perform when applied to real-world warehouse data, when considering measurements such as running time, memory requirements, and approximation ratios? How does this compare to theoretical results such as average-case and worst-case scenarios?

3. Are such models and algorithms suitable to solve the batch optimization problem in a real-world warehouse? If not, what assumptions and delimitations need to be lifted for the models and algorithms to become suitable?

1.4 Contributions

The main contribution of the thesis is a flexible software library for solving the shortest Hamiltonian path problem and the batch optimization problem, written in the C# language. It has few dependencies to avoid unnecessary complications with version handling and complex configuration management. The library includes implementations of several different algorithms, to be used under a common interface.

Along with the library comes a set of problem instances with optimal solutions for the shortest Hamiltonian path problem and approximate solutions for the batch optimization problem. The library also includes a testing suite with property-based tests and regression tests, as well as a benchmarking suite implementing benchmarks for each of the aforementioned problem instances.

2

Theory

The purpose of this chapter is to give a more detailed description of two topics that are important for the rest of this report: the travelling salesperson problem, and induced subgraphs.

The travelling salesperson problem is important due to being closely related to the shortest Hamiltonian path problem. Section 2.1 gives a description of the travelling salesperson problem, how it is related to the shortest Hamiltonian path problem (including a reduction from the shortest Hamiltonian path problem to the travelling salesperson problem), and a description of some important types of special cases.

Induced subgraphs are important due to being a part of the definition of the batch optimization problem. Section 2.2 gives a description of subgraphs as well as how induced subgraphs are created from a set of nodes or a set of edges.

2.1 The travelling salesperson problem

The *travelling salesperson problem* is a classical problem in combinatorial optimization. In informal terms, it can be described as follows:

> Given a set of cities and the distances between them, find the shortest route that starts in some city, visits all cities exactly once, and returns to the starting city.

A tour of a graph that visits all nodes exactly once and returns to the starting node is called a *Hamiltonian cycle*. This hints at the close relation between the travelling salesperson problem and the shortest Hamiltonian path problem: in the former a Hamiltonian cycle is sought; in the latter a Hamiltonian path where the first and last nodes are distinct is sought. As such, one can view the travelling salesperson problem and the shortest Hamiltonian path problem as variations of one another.

The travelling salesperson problem (and therefore, also the shortest Hamiltonian path problem) has many practical applications outside warehouses. For example, finding a route for a drill machine that should drill a set of holes in an electronic circuit, moving the drill as little as possible. Another example appears in astronomy, where a telescope should be moved between a set of sources, minimizing the amount of movement.

2.1.1 Reducing the shortest Hamiltonian path problem to the travelling salesperson problem

It is possible to reduce an instance of the shortest Hamiltonian path problem to an instance of the travelling salesperson problem in polynomial time. The reduction is as follows. Let $G = (V, E)$ be the graph in the shortest Hamiltonian path problem instance, and let s and t be the start and end nodes, respectively. An instance of the travelling salesperson problem can be constructed by augmenting G with an extra node v_0 and constructing a new graph $G' = (V', E')$, where V' is defined as

$$V' = V \cup \{v_0\}$$

and E' is defined as

$$E' = E \cup \{e_{v0,v} \mid v \in V\}.$$

The weights of the new graph are updated so that

$$w(\{v_0, v\}) = \begin{cases} 0 & \text{if } v = s \\ 0 & \text{if } v = t \\ \infty & \text{otherwise} \end{cases}$$

for all nodes $v \in V$. A travelling salesperson problem instance can now be defined for G' using v_0 as the starting node.

Once a solution to the travelling salesperson problem instance has been found, it can be transformed into a solution of the shortest Hamiltonian path problem. An optimal solution O'_{OPT} to the travelling salesperson problem instance will be of the form

$$O'_{\text{OPT}} = (v_0, s, \ldots, t, v_0).$$

No other node than s or t can appear as second to first or second to last node of an optimal solution, since the weight from v_0 to any other weight is ∞, and therefore the solution would not be optimal. An optimal solution O_{OPT} to the shortest Hamiltonian path problem instance can now be found by removing v_0 from the beginning and the end of O'_{OPT}. The length of O_{OPT} will be equal to O'_{OPT}, since

$$w(\{v_0, s\}) = w(\{t, v_0\}) = 0.$$

Note that it is possible that O'_{OPT} instead has the form

$$O'_{\text{OPT}} = (v_0, t, \ldots, s, v_0)$$

in which case one can simply reverse O'_{OPT} and follow the same argument as above.

2.1.2 Classes of the travelling salesperson problem

The general case of the travelling salesperson problem places very little constraints on the input, other than that a solution should exist. (A solution does not exist if the graph has two or more components.) However, for many practical problems, certain special cases of the travelling salesperson problem are sufficient.

One such special case is the *symmetric travelling salesperson problem*, in which all distances are symmetric: the distance from city i to city j is equal to the distance from city j to city i, for all i and j. This case is commonly used for practical applications of the travelling salesperson problem.

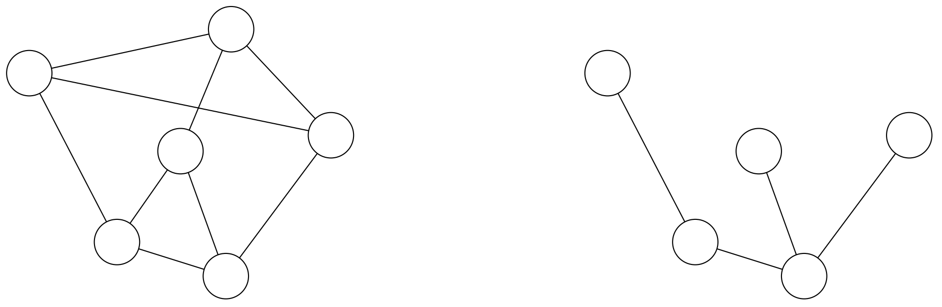
Another special case is the *metric travelling salesperson problem*, in which all distances fulfill the triangle inequality. The triangle inequality states that going directly from city i to city j is never longer than going from city i to city k and then from city k to city j. More formally, the triangle inequality states that

$$w(\{v,u\}) \leq w(\{v,w\}) + w(\{w,u\})$$

for any $v, u, w \in V$.

2.2 Induced subgraphs

From a graph $G = (V, E)$, a *subgraph* can be created. A subgraph is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. Obviously, it must hold that E' only contains edges between nodes in V'. Figure 2.1 contains an example of a graph G and a subgraph G' created from it.



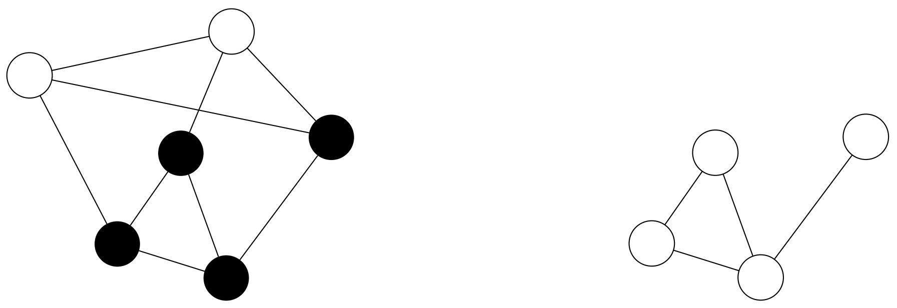
(a) The original graph. **(b)** The subgraph.

Figure 2.1: An example of an original graph and a subgraph created from it.

There exists a certain kind of subgraph, called an *induced subgraph*. An induced subgraph is created from an original graph $G = (V, E)$ and a set of nodes $V' \subseteq V$. The set of edges E' of the induced subgraph is defined as all edges in E that connect two nodes in V'. More formally, E' is defined as

$$E' = \{e_{v,u} \mid e_{v,u} \in E \land v, u \in V'\}.$$

The notation $G[V']$ denotes an induced subgraph of G by the set of nodes V'. Figure 2.2 contains an example of an original graph and an induced subgraph created from it.



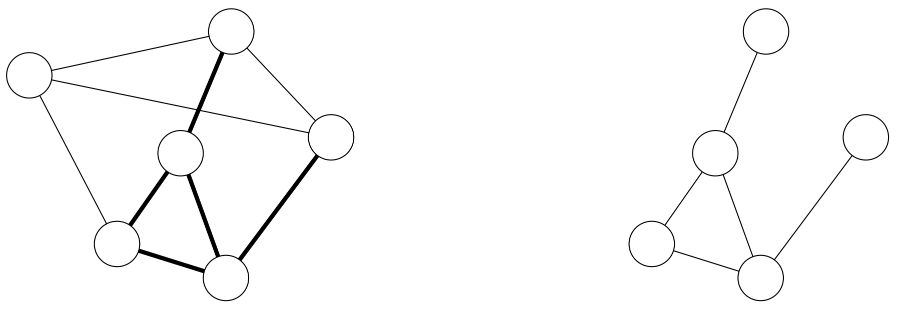
(a) The original graph. The shaded nodes are in the set of nodes V' to induce by.

(b) The induced subgraph.

Figure 2.2: An example of an original graph and an induced subgraph created from it using a set of nodes V'.

It is also possible to induce a subgraph by a set of edges, instead of by a set of nodes. The principle is similar: the set of nodes in the induced subgraph is then defined to be all nodes that are an endpoint to at least one edge in the set of edges. More formally: given a graph $G = (V, E)$ and a set of edges E', an edge-induced subgraph is a graph $G' = (V', E')$ where

$$V' = \{v \mid \exists e_{u,w} \in E' : (v = u \lor v = w)\}.$$



(a) The original graph. The thick edges are in the set of edges E' to induce by.

(b) The induced subgraph.

Figure 2.3: An example of an original graph and an induced subgraph created from it using a set of edges E'.

3
Method

To approach the shortest Hamiltonian path problem and the batch optimization problem in the context of warehouses, a set of models and algorithms were created. To evaluate them, sets of instances for the two optimization problems were created. This chapter gives theoretical descriptions of the models, the algorithms, and the instances.

Section 3.1 describes the abstract models used to represent warehouses and orders to be picked within them, as well as how those models related to the definition of the batch optimization problem. Section 3.2 describes the algorithms that were considered for implementation. Finally, Section 3.3 describes the instances of the shortest Hamiltonian path problem and the batch optimization problem that were used to evaluate the implemented algorithms.

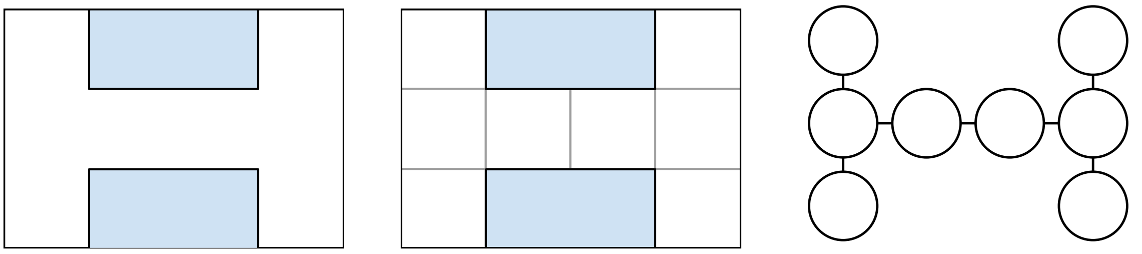
3.1 Abstract model of warehouse

In order to consolidate the abstract description of the batch optimization problem with concrete data from a warehouse, an abstract model of a warehouse and its orders is needed. A warehouse is represented as an undirected, unweighted graph, describing the layout of the warehouse. Orders are represented as subsets of nodes within that graph. This section gives a more detailed description of these models, along with a description of a method for converting the warehouse graph into a complete, weighted, undirected graph suitable for use in a batch optimization problem instance.

3.1.1 Graph representation

The initial representation of a warehouse as a graph is based on the simple intuition of looking down at the shelf layout from above. A simple example can be seen in Figure 3.1a, the white space represent aisles and the blue space represent shelves.

Onto this view a simple 2D grid is then added, this representation can be seen in Figure 3.1b. Every white square in the constructed view of the warehouse then corresponds to a node in a graph and any parts covered by shelves or walls are considered blocked spaces limiting the connection between the nodes. As all nodes are considered connected to each of the neighboring nodes sharing a full side this translates into the common graph representation that can be seen in Figure 3.1c.


(a) A model of a small warehouse as seen from above.

(b) The model from Figure 3.1a, with an added grid.

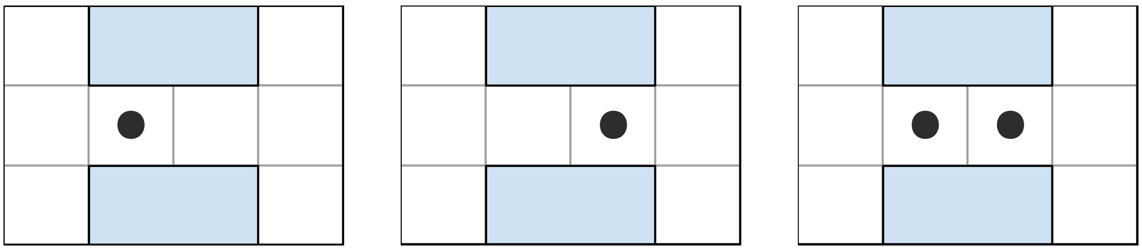
(c) The resulting graph representation of the model from Figure 3.1a.

Figure 3.1: The three steps taken in order to model a warehouse as a graph.

3.1.2 Orders

An order in a real warehouse consists of one or more *order lines*, where each order line represents a specific item, the ordered quantity of that item and one or more locations where the items are placed (if the full quantity is not matched at a single location there can be multiple). The warehouse locations usually follow a naming convention intended to make them easy to find, for example “A-01-04” corresponds to aisle A, shelf 1 and the fourth compartment from the bottom. In order to use the orders together with the aforementioned 2D warehouse representation the height dimension is disregarded while the rest of the name is translated into the corresponding row and column mapping to the node from where the compartment would be accessed. An example of this can be seen in Figure 3.2 where Figure 3.2a and Figure 3.2b display two different orders, each containing one location with different placements.

Since an order is represented by the set of nodes corresponding to the locations that need to be visited, the model discards the ordered quantity of each item. As such, the model makes a simplification to more easily consolidate the abstract batch optimization problem with a real-world warehouse. As a consequence, a batch is then a set of orders where the union of all included orders correspond to the locations necessary to visit in order to collect all the items in the batch. This is displayed in Figure 3.2c, where the union of order A and B corresponds to the combination of their separate locations. With this representation, the orders correspond to that of the formal definition of the batch optimization problem found in Section 1.2.3.


(a) Order A.

(b) Order B.

(c) Union of A and B.

Figure 3.2: Three different combinations of orders presented in the grid model from Figure 3.1b.

3.1.3 Creating a complete graph

In order to bridge the existing gap between the warehouse model with the theoretical problem descriptions mentioned in Section 1.2, the graph representation need more updates. Currently the representation consists of the 2D graph with unit weight edges presented in Section 3.1.1, while the theoretical problems require complete graphs. If the graph is connected (which is a reasonable assumption for a warehouse) such an adaptation can be performed with the help of Dijkstra's famous algorithm [10].

Running the algorithm with each node in the graph as input will sequently produce the shortest path between the input node and every other node: by adding an edge with the calculated weight between the source node and each of the targets subsequently a complete graph is produced. An example of the first step can be seen in Figure 3.3a, where the distance from the top left node to every other node in the graph has been calculated, Figure 3.3b then illustrates the addition of edges with corresponding weight to the graph representation. The resulting complete graph is then a representation of the warehouse where every edge weight corresponds to the shortest path between two nodes, and as such it follows the triangle inequality explained in Section 2.1.2. And matches the complete graph in the formal description of the batch optimization problem that is found in Section 1.2.3

Proposition 3.1. *The weights of the complete graph fulfill the triangle inequality.*

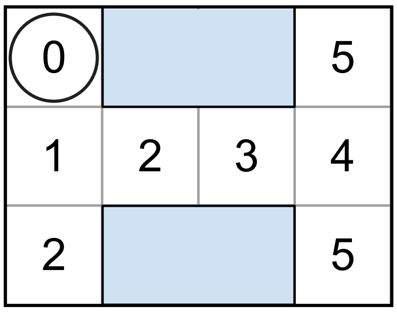
Proof. Define $P_{v,u}$ to be a shortest path between any pair of distinct nodes v and u. Furthermore, let $d(P_{v,u})$ denote the length of $P_{v,u}$. Now, assume that for some choice of nodes v, u, w the triangle inequality does not hold, i.e., there exists shortest paths $P_{v,w}, P_{v,u}, P_{u,w}$ such that

$$d(P_{v,w}) > d(P_{v,u}) + d(P_{u,w}).$$

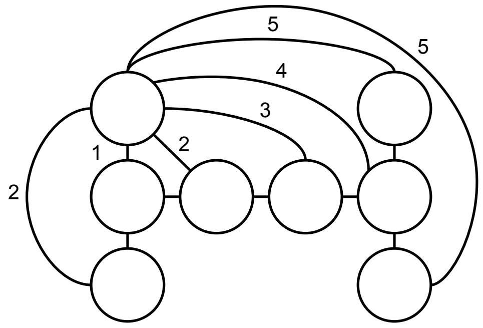
It would then be possible to create a path $P'_{v,w}$ by concatenating $P_{v,u}$ and $P_{u,w}$. Clearly, it holds that

$$d(P'_{v,w}) = d(P_{v,u}) + d(P_{u,w}).$$

Furthermore, $d(P_{v,w}) > d(P'_{v,w})$. This is a contradiction, since $P_{v,w}$ was defined to be a shortest path between v and w, and therefore no path exists that is shorter than $P_{v,w}$. Thus, the assumption is false, and it holds that the triangle inequality holds for any choice of nodes v, u, w. $\square$



(a) Calculated distances from the first node to every other node.



(b) The calculated edges in Figure 3.3a added to the graph representation displayed in Figure 3.1c.

Figure 3.3: Example of one calculation of paths of minimal distance, and the resulting edges in what will become a complete graph.

3.2 Considered algorithms

This section treats algorithms for the shortest Hamiltonian path problem and the batch optimization problem that were considered for implementation and evaluation. The algorithms are treated with a description of their origins, some details about how they work, as well as what their performance characteristics are in terms of approximation ratio, time complexity, and space complexity.

3.2.1 Algorithms for the shortest Hamiltonian path problem

Since the shortest Hamiltonian path problem is so closely related to the travelling salesperson problem, there exists many algorithms for the travelling salesperson problem that can be adapted for the shortest Hamiltonian path problem. This section presents a selection of such algorithms, with a brief description of each algorithm along with their time and space complexities.

3.2.1.1 Held-Karp

The Held-Karp algorithm [15] is a dynamic programming algorithm that solves the symmetric travelling salesperson problem optimally. The algorithm is an early application of the dynamic programming approach, and was discovered independently by Bellman [1] as well as Held and Karp in 1962. It is simple and arguably the most well-known algorithm for solving the symmetric travelling salesperson problem optimally.

The algorithm is based on the following property: if P is a path of minimum distance between two nodes in a graph then every subpath of P is also of minimum distance. This property can be used to solve a subproblem of the travelling salesperson problem: finding a path of minimum distance that visits a subset of the nodes in the graph exactly once, and ending in a specific node. Solutions to such subproblems can then be used in a recursive fashion to find a solution for the original problem.

In the authors' description of the algorithm the nodes of a graph are represented by numbers $\{1, \ldots, N\}$. The node denoted by 1 is then considered to be the start node of the tour. However, the authors do not prescribe a specific method for creating such a numbering; it is just used for convenient notation. A solution is then a permutation $(1, i_2, \ldots, i_N)$ where $i_2, \ldots, i_N \in \{2, \ldots, N\}$ and $i_j \neq i_k$ for $j \neq k$.

The core of the algorithm is a function called C. It is defined for all $S \subseteq \{2, \ldots, N\}$ and all $l \in S$, and denotes the minimum cost of starting in node 1, visiting each node in S, and ending in node l. Formally, it is defined as follows:

$$C(S, l) = \begin{cases} w(\{1, l\}) & \text{for } |S| = 1 \\ \min_{m \in S \setminus \{l\}} [C(S \setminus \{l\}, m) + w(\{m, l\})] & \text{for } |S| > 1 \end{cases} \tag{3.1}$$

The algorithm to find a path of minimum distance is a two-phase computation. In the first phase, the values of the function C are calculated recursively for all values of S and l. Finally, the minimum path cost is calculated as

$$C^* = \min_{l \in \{2, \ldots, N\}} [C(\{2, \ldots, N\}, l) + w(\{l, 1\})]. \tag{3.2}$$

In the second phase, a path of minimum cost is calculated. First, the value of l that gives a minimum value of C^* in (3.2) is determined. That gives the last node in the tour, i_N. Knowing i_N, it is possible to calculate i_{N-1} in a similar fashion based on the definitions in (3.1), and repeat that calculation recursively until the full tour has been calculated.

The algorithm is proven by its authors to have a time complexity of $\mathcal{O}(2^N N^2)$ and a space complexity of $\mathcal{O}(2^N N)$. As such, the execution time and the required space grow exponentially, and it is infeasible to use for even moderately large values of N. However, it still has value as a simple, deterministic algorithm to find an optimal solution.

Since the algorithm solves the travelling salesperson problem, it needs to be adapted slightly to solve the shortest Hamiltonian path problem. For the shortest Hamiltonian path problem, it is assumed that the tour starts in node 1, visits each node, and ends in node N. The C function is defined as in (3.1), however, the S argument is now defined to be $S \subseteq \{2, \ldots, N-1\}$. The minimum path cost is then calculated as

$$C^* = \min_{l \in \{2, \ldots, N-1\}} [C(\{2, \ldots, N-1\}, l) + w(\{l, N\})].$$

The adapted version of the algorithm can easily be seen to retain the complexities of the original algorithm.

3.2.1.2 Held-Karp successive approximation

Since the optimal Held-Karp algorithm (described in Section 3.2.1.1) has exponential complexities in both time and space, Held and Karp recognized that it was infeasible to use for larger problem instances. Therefore, in the same paper [15], Held and Karp also describe an approximation algorithm. Furthermore, Held and Karp describe a computer program that implements the approximation algorithm, and found that in many cases it was able to find an optimum tour (or what is believed to be an optimum tour).

The main idea of the algorithm is to from a given tour P try to construct a tour P' that is shorter than P. If successful, then P' can be the basis for a new application of the algorithm: from P', try to construct a tour P'' that is shorter than P'. Using this successive approach, one can hope to eventually construct a tour that is close to an optimal tour.

The approximation algorithm is carried out by optimally solving the travelling salesperson problem on an instance of smaller size, and using the solution for the smaller instance to construct a solution for the original problem. A tour P is partitioned into subpaths, where the number of subpaths is given by an additional parameter q. Then, an instance of the travelling salesperson problem consisting of q nodes is defined where the weight between one subpath $(i_j, \ldots, i_k)$ and another subpath $(i_l, \ldots, i_m)$ is given by $w(\{i_k, i_l\})$. Using the optimal algorithm described in Section 3.2.1.1, a solution to the q-node travelling salesperson problem instance is acquired, and the subpaths are then joined to form a complete tour of the original graph.

Partitioning the path P into q segments gives rise to the problem of how to choose a partition that leads to a better solution. Held and Karp describe two types of partitions that have, in their experience, proven to be useful: a *local partition* and a *global partition*. In the local partition, $q - 1$ subpaths consist of a single node each, and the remaining subpath consists of the rest of the nodes. In the global partition, all q subpaths are of nearly equal size. Held and Karp recommend that, when employing the successive approach, alternate phases of local and global partitions tend to be desirable [15].

The complexities of the algorithm depends largely on the choice of q. The optimal algorithm used to solve the q-node has a time complexity of $\mathcal{O}(2^q q^2)$ and a space complexity of $\mathcal{O}(2^q q)$. Doing k iterations of the successive approach then yields a time complexity of $\mathcal{O}(k \cdot 2^q q^2)$ and a space complexity of $\mathcal{O}(2^q q)$.

3.2.1.3 Nearest neighbor

The nearest neighbor algorithm is heuristic algorithm for the travelling salesperson problem. There appears to be no clear origin for the algorithm, although it has been described in a paper by Bellmore and Nemhauser from 1968 [2]; as such, the origin of the algorithm can be said to be “folklore”.

The algorithm is based on a simple greedy rule: from the current node select the closest node that has not yet been visited Application of this rule is repeated until all nodes in the graph have been visited. Algorithm 3.1 contains a full description of the nearest neighbor algorithm.

There is no bound on the approximation ratio achieved by the nearest neighbor algorithm for the general travelling salesperson problem. In fact, it is possible to specifically craft instances of the travelling salesperson problem where the nearest neighbor algorithm yields the worst route [14]. However, while there is no

Algorithm 3.1 The nearest neighbor algorithm for the travelling salesperson problem on a graph $G = (V, E)$.

```
1: Let $Q$ be the set of unvisited nodes. $Q$ is initialized to $V \setminus \{s\}$.
2: Let $T$ be the sequence of nodes in the tour. $T$ is initialized to $[s]$.
3: Let $v$ be the current node. Initialize $v$ to $s$.
4: while there are nodes in $Q$ do
5:     Find $u \in Q$ such that $w(\{v, u\})$ is minimized.
6:     Remove $u$ from $Q$.
7:     Append $u$ to $T$.
8:     Set $v$ to $u$.
9: end while
10: Return $T$.
```

theoretical bound of the approximation ratio, the nearest neighbor algorithm may produce good results in practice. Furthemore, for instances of the metric travelling salesperson problem it can be shown that the worst-case approximation ratio is $\mathcal{O}(\log |V|)$ [27]. This makes the nearest neighbor algorithm interesting for the purposes of this thesis, where the graphs will have weights fulfilling the triangle inequality (see Section 3.1.3).

The time complexity of the algorithm is $\mathcal{O}(|V|^2)$. There is one iteration of the loop in step 4 for each node in V. In each loop, finding the closest node in step 5 takes $\mathcal{O}(|V|)$ time. Therefore, the total complexity of the algorithm is $\mathcal{O}(|V|^2)$.

The space complexity of the algorithm is $\mathcal{O}(|V|)$ since both Q and T require $\mathcal{O}(|V|)$ space each, and finding the minimum element in step 5 can be done in $\mathcal{O}(1)$ space. All other operations can clearly be done in $\mathcal{O}(1)$ space, yielding a final space complexity of $\mathcal{O}(|V|)$.

Adapating the nearest neighbor algorithm for the shortest Hamiltonian path problem is straightforward. The greedy rule is unmodified, but the set Q is initialized to also not contain the end node v_t. Finally, there is an extra operation after the loop that adds the end node to the tour. Algorithm 3.2 contains a full description of the adapted nearest neighbor algorithm for the shortest Hamiltonian path problem. The adapted algorithm can easily be seen to retain the time and space complexities of the original algorithm.

3.2.1.4 Tree doubling

The tree doubling algorithm is a simple approximation algorithm for the metric travelling salesperson problem (see Section 2.1) that achieves an approximation ratio of 2 [3]. The algorithm does not appear to have a single, clear origin; it appears to be "folklore" [3; 29, pp. 31–32].

The algorithm is described in Algorithm 3.3. An Eulerian tour is a tour of a graph that visits each edge exactly once. Not all graphs have Eulerian tours; therefore, a graph where at least one Eulerian tour exists is called an Eulerian graph. A multigraph is a graph where there may exist several edges between a pair of nodes.

In order to prove the approximation ratio, one first needs to prove the following lemma.

Algorithm 3.2 The nearest neighbor algorithm for the shortest Hamiltonian path problem on a graph $G = (V, E)$.

```
1: Let $Q$ be the set of unvisited nodes. $Q$ is initialized to $V \setminus \{s, t\}$.
2: Let $T$ be the sequence of nodes in the tour. $T$ is initialized to $[s]$.
3: Let $v$ be the current node. Initialize $v$ to $s$.
4: while there are nodes in $Q$ do
5:     Find $u \in Q$ such that $w(\{v, u\})$ is minimized.
6:     Remove $u$ from $Q$.
7:     Append $u$ to $T$.
8:     Set $v$ to $u$.
9: end while
10: Append $t$ to $T$.
11: Return $T$.
```

Algorithm 3.3 Tree doubling algorithm for the travelling salesperson problem on a graph G, as described by Bläser [3].

```
1: Compute a minimum spanning tree $T^*$ of $G$.
2: Duplicate each edge of $T^*$ and obtain an Eulerian multigraph $T'$.
3: Compute an Eulerian tour of $T'$. Whenever a node is visited in the Eulerian
   tour that was already visited, this node is skipped and one proceeds with the
   next unvisited node along the Eulerian tour. Return the resulting Hamiltonian
   tour $H$.
```

Lemma 3.1. *Let G be an undirected, weighted graph, and let P^* be a tour of minimum length that visits each node in G exactly once (i.e., an optimal solution to the travelling salesperson problem). Define $cost(E)$ to be the sum of the weights of the edge set E. Finally, let T^* be a minimum spanning tree of G. Then $cost(T^*) \leq cost(P^*)$.*

Proof. Obtain T by removing an arbitrary edge from P^*. Then T is a spanning tree, and $cost(T) \leq cost(P^*)$. Since T^* is a minimum spanning tree, it holds that $cost(T^*) \leq cost(T)$. Then $cost(T^*) \leq cost(T) \leq cost(P^*)$. □

Using Lemma 3.1, it is possible to prove the approximation ratio of Algorithm 3.3.

Theorem 3.1. *Let G be an undirected, weighted graph, and let P^* be a tour of minimum length that visits each node in G exactly once (i.e., an optimal solution to the travelling salesperson problem). Let H be a tour output by Algorithm 3.3. Then $cost(H) \leq 2 \cdot cost(P^*)$.*

Proof. Let T^* be the minimum spanning tree computed in step 1. By Lemma 3.1, $cost(T^*) \leq cost(P^*)$. Since T' is obtained by doubling every edge in T^*, it follows that $cost(T') = 2 \cdot cost(T^*) \leq 2 \cdot cost(P^*)$. Let H be the Hamiltonian tour output computed in step 3. When skipping already visited nodes, a path $(i_j, i_{j+1}, \ldots, i_{k-1}, i_k)$ is replaced by an edge $\{i_j, i_k\}$. The triangle inequality ensures that $w(\{i_j, i_k\}) \leq cost(i_j, i_{j+1}, \ldots, i_{k-1}, i_k)$. It then follows that $cost(H) \leq cost(T')$ and therefore $cost(H) \leq 2 \cdot cost(P^*)$. □

The proofs of both Lemma 3.1 and Theorem 3.1 are adaptations of the proofs given by Vazirani [29].

The complexities of Algorithm 3.3 are dominated by the complexities of the algorithm creating the minimum spanning tree in step 1. For example, if using Kruskal's algorithm [20] and using Mergesort to sort the edges by weight as a pre-computation step, step 1 has a time complexity of $\mathcal{O}(|E| \log |E|)$ and a space complexity of $\mathcal{O}(|E|)$. The other steps can trivially be seen to have a time complexity of $\mathcal{O}(|E|)$ and space complexity of $\mathcal{O}(|E|)$, where $|E|$ is the number of edges in the graph.

The tree doubling algorithm can easily be adapted to instead solve the shortest Hamiltonian path problem with fixed start and end points. The adapted algorithm proceeds as in Algorithm 3.4.

Algorithm 3.4 Tree doubling algorithm for the shortest Hamiltonian path problem on a graph G with start point s and end point t.

```
1: Compute a minimum spanning tree $T^*$ of $G$.
2: Duplicate each edge of $T^*$ and obtain an Eulerian multigraph $T'$.
3: Compute an Eulerian tour of $T'$ starting in $s$. Whenever a node is visited in the
   Eulerian tour that was already visited, this node is skipped and one proceeds
   with the next unvisited node along the Eulerian tour. The only exception is $t$,
   which is always skipped, and finally appended to the end of the tour. Return
   the resulting Hamiltonian path $H$.
```

The approximation ratio of Algorithm 3.4 is 2, and the proof of the approximation ratio is similar to the proof for the original algorithm.

Theorem 3.2. *Let G be an undirected, weighted graph, and let P^* be a path of minimum length that starts in node s, visits each node in G exactly once, and ends in node t (i.e., an optimal solution to the shortest Hamiltonian path problem). Let H be a path output by Algorithm 3.4. Then $cost(H) \leq 2 \cdot cost(P^*)$.*

Proof. Let T^* be the minimum spanning tree computed in step 1. Since P^* is a Hamiltonian path, it follows that P^* is a spanning tree. Therefore, $cost(T^*) \leq cost(P^*)$. The remainder of the proof is analogous to the proof of Theorem 3.1, without the use of Lemma 3.1. $\square$

3.2.1.5 Christofides

Christofides's algorithm is an approximation algorithm for the metric travelling salesperson problem. It was described in a technical report in 1976. The approximation ratio achieved by Christofides's algorithm is 3/2 [5], which remains the currently best known approximation ratio for the metric travelling salesperson problem.

It is similar to the tree doubling algorithm described in Section 3.2.1.4. However, instead of doubling every edge in order to obtain an Eulerian multigraph, a minimum weight perfect matching is computed for nodes with an odd degree in the minimum spanning tree. The matching is then merged with the minimum spanning tree, creating an Eulerian multigraph. Then, an Eulerian tour is calculated on the resulting graph, and the practice of skipping already visited nodes is employed in order to obtain a Hamiltonian tour [3].

The complexities of Christofides's algorithm are dominated by the step of finding a minimum weight perfect matching. There exist algorithms for finding such a matching whose time complexity is $\mathcal{O}(|V|^3)$ [12; 18], making the time complexity of Christofides's algorithm equivalent.

For the shortest Hamiltonian path problem, where the start and end points are fixed and different from one another, the Christofides algorithm can be adapted. The adapted algorithm achieves an approximation ratio of 5/3 [17], and retains the time complexities of the original algorithm. For certain special cases, there exist algorithms that improve this bound slightly. For example, for the special case where each edge in the underlying graph has unit distance, there exists an algorithm that improves the bound to 1.586 [24].

3.2.2 Algorithms for the batch optimization problem

This section describes a set of *relative algorithms* for the batch optimization problem. The concept of relative algorithms is described in Section 3.2.2.1, and three such algorithms then follow in the subsequent sections.

The notation in this section is consistent with the notation in Section 1.2.3. Let $\mathcal{L}$ be the set of all orders, let $\mathcal{L}_N$ be the set of batches of size N, and let U be the union function. Then let $|\mathcal{B}_{\max}|$ be the size of the batch where the union of the included orders has maximum size. More precisely,

$$|\mathcal{B}_{\max}| = \max_{\mathcal{B} \in \mathcal{L}_N} |U(\mathcal{B})|.$$

3.2.2.1 Relative algorithms

The considered algorithms for the batch optimization problem are all what has been dubbed *relative algorithms*. They use an *evaluation algorithm* that evaluates (partial) solutions by assigning an integer number to the solution. The main algorithm then makes decisions based on that number. In other words, the main algorithm is "relative" to the evaluation algorithm; hence the name "relative algorithms".

For the relative batch optimization problem algorithms, a (partial) solution is a batch of orders. To evaluate a batch, a set of nodes is created by taking the union of the start node, the end node, and the nodes in all included orders. The set of nodes is then used to create an induced subgraph, which combined with the start and end nodes defines an instance of the shortest Hamiltonian path problem. Therefore, an evaluation algorithm for the relative batch optimization problem algorithms is an algorithm for the shortest Hamiltonian path problem.

The notation used in this report is that F denotes an evaluation algorithm, and $F(\mathcal{B})$ denotes the integer value assigned to the batch $\mathcal{B}$ by F. Note that $F(\mathcal{B})$ is defined in the context of a specific instance of the batch optimization problem, meaning a specific graph and start and end nodes. Furthermore, $\mathcal{O}_F^T(n)$ and $\mathcal{O}_F^S(n)$ denote the time and space complexities of the evaluation algorithm for an instance of size n, respectively.

3.2.2.2 Relative brute force algorithm

As with any combinatorial optimization problem, one can always do a brute force search for an optimal solution. In the case of the batch optimization problem, a relative brute force search algorithm will yield an optimal solution if combined with an optimal evaluation algorithm for the shortest Hamiltonian path problem. However, it may also be the case that a brute force search combined with an approximating evaluation algorithm will yield good (but not necessarily optimal) solutions.

The relative brute force algorithm is straightforward. Let $F(\mathcal{B})$ be the value returned by the evaluation algorithm for some batch $\mathcal{B}$. Then, find

$$\mathcal{B}^* = \arg \min_{\mathcal{B} \in \mathcal{L}_N} F(\mathcal{B}) \tag{3.3}$$

by complete enumeration of all $\mathcal{B} \in \mathcal{L}_N$.

The quality of the solution returned by the brute force search depends, as mentioned above, on the choice of evaluation algorithm. More specifically, if the evaluation algorithm solves the shortest Hamiltonian path problem with an approximation ratio of α, then the brute force search solves the batch optimization problem with an approximation ratio of α. This is formalized as the following theorem.

Theorem 3.3. *Let $\mathcal{B}^*$ be the batch returned by the brute force search for some evaluation algorithm. Let $\mathcal{B}_{OPT}$ be an optimal solution to the batch optimization problem, meaning that $D^*(U(\mathcal{B}_{OPT})) \leq D^*(U(\mathcal{B}))$ for all $\mathcal{B} \in \mathcal{L}_N$. Finally, let α be the approximation ratio of the evaluation algorithm, meaning that $D^*(U(\mathcal{B})) \leq F(\mathcal{B}) \leq \alpha \cdot D^*(U(\mathcal{B}))$ for all $\mathcal{B} \in \mathcal{L}_N$. Then $D^*(U(\mathcal{B}^*)) \leq \alpha \cdot D^*(U(\mathcal{B}_{OPT}))$.*

Proof. By the definition of $\mathcal{B}^*$ in (3.3) it holds that

$$\forall \mathcal{B} \in \mathcal{L}_N : F(\mathcal{B}^*) \leq F(\mathcal{B})$$

and specifically

$$F(\mathcal{B}^*) \leq F(\mathcal{B}_{\text{OPT}}). \tag{3.4}$$

By definition of the approximation ratio it holds that

$$F(\mathcal{B}_{\text{OPT}}) \leq \alpha \cdot D^*(U(\mathcal{B}_{\text{OPT}}))$$

and in combination with (3.4) it then follows that

$$F(\mathcal{B}^*) \leq \alpha \cdot D^*(U(\mathcal{B}_{\text{OPT}})). \tag{3.5}$$

By definition of the approximation ratio, it also holds that

$$D^*(U(\mathcal{B}^*)) \leq F(\mathcal{B}^*)$$

which in combination with (3.5) yields

$$D^*(U(\mathcal{B}^*)) \leq \alpha \cdot D^*(U(\mathcal{B}_{\text{OPT}})).$$

□

The time complexity of the brute force search is

$$\mathcal{O}(|\mathcal{L}_N|) \cdot \mathcal{O}_F^T(|\mathcal{B}_{\max}|).$$

The algorithm invokes the evaluation algorithm $|\mathcal{L}_N|$ times, with each invocation having complexity $\mathcal{O}_F^T(|\mathcal{B}_{\max}|)$. Note that $|\mathcal{L}_N| = \binom{|\mathcal{L}|}{N}$, and as such, the brute force search has a time complexity that is worse than exponential.

The space complexity of the brute force search is $\mathcal{O}_F^S(|\mathcal{B}_{\max}|)$. The brute force search algorithm itself requires $\mathcal{O}(1)$ space: it only needs to keep track of the currently best known batch.

3.2.2.3 Relative greedy algorithm

One can construct a simple greedy algorithm that is also relative to an evaluation algorithm, similar to the brute force search described in Section 3.2.2.2. The greedy algorithm builds a batch by repeatedly adding the order that causes the least increase in the total distance travelled. The algorithm is described more precesily in Algorithm 3.5. It should be noted that $\mathcal{B}_0$ may be chosen before the execution of the algorithm. The simplest possible choice is to set $\mathcal{B}_0 = \emptyset$, but it may also be feasible to choose it according to some other heuristic, such as finding an optimal batch of size $k \ll N$.

Algorithm 3.5 A greedy approximation algorithm for the batch optimization problem.

```
1: $\mathcal{B} \leftarrow \mathcal{B}_0$                                  ▷ $\mathcal{B}_0$ is some initial solution.
2: while $|\mathcal{B}| < N$ do
3:     $L^* \leftarrow \arg\min_{L \in \mathcal{L} \setminus \mathcal{B}} F(\mathcal{B} \cup \{L\})$
4:     $\mathcal{B} \leftarrow \mathcal{B} \cup \{L^*\}$
5: end while
6: return $\mathcal{B}$
```

At the time of writing, it is unknown whether the algorithm can provide any approximation ratio guarantees; it is suspected that the algorithm can yield arbitrarily bad solutions.

The time complexity of the algorithm is

$$\mathcal{O}(N|\mathcal{L}|) \cdot \mathcal{O}_F^T(|\mathcal{B}_{\max}|).$$

Assume that $\mathcal{B}_0$ is set to $\emptyset$. In the first iteration of the loop, $|\mathcal{L}|$ orders are evaluated using the evaluation algorithm (from the minimization in step 3). In the second iterations, $|\mathcal{L}| - 1$ orders are evaluated. This scheme continues for N iterations, resulting in

$$\sum_{i=0}^{N-1} |\mathcal{L}| - i \leq N|\mathcal{L}|$$

invocations of the evaluation algorithm.

The space complexity is $\mathcal{O}(N) + \mathcal{O}_F^S(|\mathcal{B}_{\max}|)$. The algorithm needs to keep track of which orders are currently selected, hence the $\mathcal{O}(N)$ term. Each invocation of the evaluation algorithm can be made in sequence and the occupied space can be reused among invocations, hence the constant $\mathcal{O}_F^S(|\mathcal{B}_{\max}|)$ term.

3.2.2.4 Relative randomized algorithm

It is easy to imagine in a randomized approach to the batch optimization problem: from the set of available orders, select randomly a number of batches. Of the randomly selected batches, return the batch that yields the shortest path through the warehouse. Algorithm 3.6 describes such a randomized approach more formally. The number k is given as an input parameter to the algorithm.

Algorithm 3.6 A randomized algorithm for the batch optimization problem.

```
1: Let $\mathcal{L}$ be the set of available orders.
2: Let $N$ be the batch size.
3: Let $F$ be an evaluation algorithm.
4: Select uniformly at random $\mathcal{B}_1, \dots, \mathcal{B}_k \in \mathcal{L}_N$ such that $\mathcal{B}_i \neq \mathcal{B}_j$ for $i \neq j$.
5: Return $\mathcal{B}_i$ such that $F(\mathcal{B}_i) \leq F(\mathcal{B}_j)$ for all $j = 1, \dots, k$.
```

It is assumed that randomly selecting a single element in $\mathcal{L}$ has a time complexity of $\mathcal{O}(1)$. Randomly selecting a batch of size N then has a time complexity of $\mathcal{O}(N)$, and thus randomly selecting k such batches has a time complexity of $\mathcal{O}(kN)$. The evaluation algorithm will be invoked for each of the k batches, leading to a overall time complexity of $\mathcal{O}(k) \cdot \mathcal{O}_F^T(|\mathcal{B}_{\max}|)$ to evaluate all batches. The final time complexity is therefore

$$\mathcal{O}(kN) + \mathcal{O}(k) \cdot \mathcal{O}_F^T(|\mathcal{B}_{\max}|).$$

The space complexity of the algorithm is similar. It is again assumed that randomly selecting a single element in $\mathcal{L}$ has a space complexity of $\mathcal{O}(1)$. The space required to store all randomly selected batches is then $\mathcal{O}(kN)$. The evaluation algorithm will be invoked for each of the k batches, but the space required by one invocation can be re-used for the next invocation. Therefore, the total space complexity of all invocations of the evaluation algorithm is $\mathcal{O}_F^S(|\mathcal{B}_{\max}|)$, and the total space complexity of the algorithm is

$$\mathcal{O}(kN) + \mathcal{O}_F^S(|\mathcal{B}_{\max}|).$$

Clearly, there is no worst-case approximation ratio of the algorithm described in Algorithm 3.6: there is no guarantee that any randomly selected batch is better than the worst possible batch. The value of the algorithm lies instead in its relatively low time and space complexities. For the purposes of this thesis, a randomized algorithm is mostly interesting as a baseline comparison: how do the algorithms described in Section 3.2.2.2 and Section 3.2.2.3 perform when compared to a simple randomized approach?

3.3 Problem instances

To evaluate the considered algorithms, a set of problem instances were needed. One set of instances was created for the shortest Hamiltonian path problem, and another set was created for the batch optimization problem. This section describes in more detail how these sets were created.

3.3.1 Instances of the shortest Hamiltonian path problem

An instance of the shortest Hamiltonian path problem can be created on any complete, weighted, undirected graph by designated two nodes v_s, v_t where $v_s \neq v_t$ to act as the start and end nodes. This section describes a set of instances of the shortest Hamiltonian path problem that were chosen to be of different kinds and sizes, and which were used for testing and benchmarking the algorithms described in Section 3.2.1.

The set of instances was created based on two sources: an existing set of instances for the travelling salesperson problem called TSPLIB [26], and a set of randomly chosen induced subgraphs based on a graph describing a warehouse. Section 3.3.1.1 describes the instances from TSPLIB, and Section 3.3.1.2 describes the instances created from the induced subgraphs.

3.3.1.1 TSPLIB instances

There exists a collection of instances of the travelling salesperson problem (and problems that are related to the travelling salesperson problem) called *TSPLIB*. It was first described in a paper by Reinelt published in 1991 [26], and has since become well-known in the research community. TSPLIB has commonly been used to benchmark solvers for the travelling salesperson problem [7; 16; 11].

The problem instances contained in TSPLIB are diverse. Several of the instances—such as `brazil58`, `brd14051`, and `usa13509`—are based on cities in specific countries or other geographical regions. Other instances are based on randomly generated graphs (e.g., `dsj1000`) or points on an electronic circuit where holes must be drilled (e.g., `d493`).

An advantage of using a diverse set of problem instances is that it may surface subtle characteristics of a solver. A solver could for example have different running times for problem instances of approximately equal sizes. An example is the Concorde solver for the travelling salesperson problem: the TSPLIB instance `ali535` containing 535 nodes was solved in 9.50 seconds by Concorde, while the instance `si535` also containing 535 nodes was solved in 21.73 seconds [7]. Having different performance characteristics (such as running time) for different types of problem instances of similar size could be an indication that an algorithm works well for certain kinds of graph, but less so for other kinds.

Table 3.1 contains a subset of instances from the TSPLIB collection of instances. These instances were selected based on their size: the number of nodes should not be much more than 1000, and should be fairly evenly distributed among the instances. In all instances, the start and end nodes were set to be the first and second node, respectively, as determined by the order of the nodes in the corresponding data files containing the instances. The instances were reduced to instances of the shortest Hamiltonian path problem using the reduction described in Section 2.1.1.

Instance	Size ($\lvert V\rvert$)
`burma14`	14
`ulysses22`	22
`att48`	48
`pr76`	76
`bier127`	127
`d198`	198
`a280`	280
`gr431`	431
`u574`	574
`vm1084`	1 084

Table 3.1: List of TSPLIB instances that were used to test and benchmark algorithms for the shortest Hamiltonian path problem.

3.3.1.2 Induced instances

The algorithms for the shortest Hamiltonian path problem are for the purposes of this thesis intended to be run on graphs representing warehouses. However, it will rarely be the case that the algorithms are applied to a graph representing an entire warehouse; rather, the algorithms would be applied to induced subgraphs (see Section 2.2), only containing the locations that a picker would need to visit in order to collect the items in a batch. Therefore, it would be interesting to test and benchmark the algorithms on such induced subgraphs of different sizes.

Starting from a graph representing a full warehouse layout (see Figure 3.4), a set of induced subgraphs was created. The full warehouse graph contains 1423 nodes. Each induced subgraph was of a different size, where the sizes were chosen to have approximately the same distribution as the sizes of the TSPLIB instances described in Section 3.3.1.1. For each size, a subset of nodes were selected uniformly at random from the full set of nodes in the graph. Then, an induced subgraph was created based on the selected subset of nodes.

Table 3.2 contains a list of instances of the shortest Hamiltonian path problem that were created in the fashion described above. For each instance the start and end nodes were selected to be the first and second node, respectively, as determined by a fixed, arbitrary numbering of the nodes.

Instance	Size ($\lvert V \rvert$)
`induce10`	10
`induce15`	15
`induce25`	25
`induce50`	50
`induce75`	75
`induce100`	100
`induce150`	150
`induce200`	200
`induce300`	300
`induce400`	400
`induce600`	600
`induce800`	800

Table 3.2: List of instances created from randomly selected induced subgraphs.

3.3.2 Instances of the batch optimization problem

This section describes two sets of instances used for evaluation of the batch optimization problem algorithms. One set of instances was created from snapshots from the database belonging to one of Ongoing Warehouse's customers. These instances therefore contain real-world warehouse data. Another set of instances was derived from the aforementioned set of instances, by randomly selecting subsets of the sets of available orders. These instances therefore also contain real-world warehouse data, but are of a much smaller size.

3.3.2.1 Creating snapshots of databases

Ongoing Warehouse has the ability to retroactively create snapshots of their customers' database. More precisely, Ongoing Warehouse has the ability to specify a point of time in the past and make a copy of a database's state (e.g., its schema and data) at that point in time. This copy is what is referred to as a *snapshot* for the purposes of this report.

Snapshots can be created by an internal tool which lets the user specify a database and a timestamp given with a granularity of one minute, and then creates a copy of that database's state for the given timestamp. A snapshot can be created for points in time that are generally not more than 6 months in the past. The snapshots can be deployed as separate database instances, allowing Ongoing Warehouse to interact with the system at a specific point in time without affecting the current state of the system.

3.3.2.2 Using snapshots to create sets of available orders

The ability to create snapshots (see Section 3.3.2.1) was used to create sets of available orders. A set of available orders corresponds to the set $\mathcal{L}$ as described in

Section 1.2.3. The sets of available orders were created based on batches that already existed in the database, i.e., batches that had actually been picked by a picker in the warehouse. To determine the set of available orders at the time of creation for such a batch, a snapshot was created at a point in time one minute before the batch was created. It was also verified that no new orders arrived between the time of the snapshot and the creation time of the batch. All orders with the status of *pickable* were included in the set of available orders.

The status pickable means that the warehouse has a sufficient number of all item included in the order, meaning that the entire order can be picked. However, a single item could be placed in different locations in the warehouse. Therefore, an order must then be *allocated*, which means that each item in the order is allocated a specific location in the warehouse from where it should be picked. All pickable orders can be allocated, and all allocated orders are pickable.

After the snapshot had been created, all orders that were included in the set of available orders were allocated. The result was a set of sets of locations in the warehouse that a picker must visit to pick all items in an orders, which is then the set $\mathcal{L}$.

3.3.2.3 Description of warehouse

The data used for creating instances of the batch optimization problem from real world warehouse data was collected with the help of one of Ongoing Warehouse's customers. The customer is based in Norway and is a large retailer of kitchen supplies. The warehouse used for the model and collection of data has a total of around 4000 storage spaces. This results in a total of 866 potential item locations when disregarding height as in the model described in Section 3.1.

As can be seen in Figure 3.4, the shelf layout does not follow a strictly rectangular shape. The reason behind the model structure is due to taking the amount of shelf compartments into account rather than their physical length into account, due to the length information being unknown for the thesis. The actual shelf-placement and their correlation to the packing stations (displayed as the grey area in the bottom left corner) is an estimation corresponding to the actual warehouse layout, and is based on information from Ongoing Warehouse.

Figure 3.4 also contain two small dark grey squares (in the top and bottom left corner respectively), they correspond to the start and end points necessary in order to create an instance of the batch optimization problem as described in Section 1.2.3.

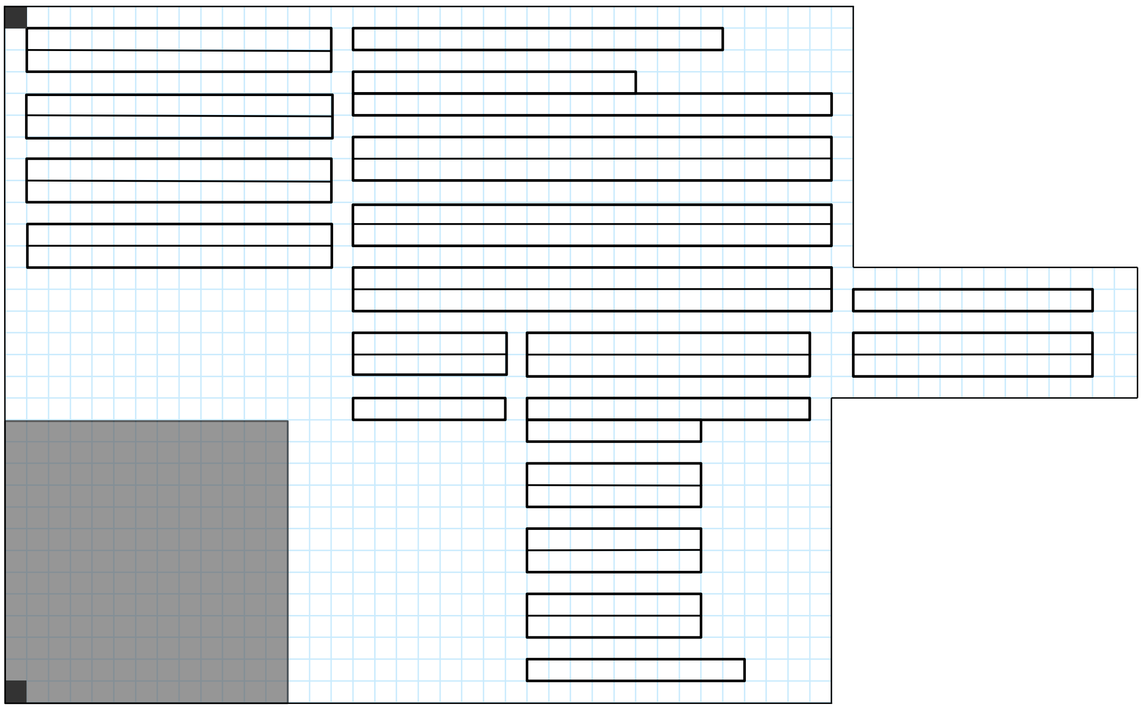


Figure 3.4: An approximate birdseye view map of the warehouse, modeled from the shelf layout.

3.3.2.4 Full instances from warehouse data

From the database of the customer described in Section 3.3.2.3 a total of six batches were selected. The batches were originally created on different dates during the first four months of 2020. No particular method was used to select these six batches; they were chosen mostly arbitrarily.

For each original batch, a set of available orders was created using the method described in Section 3.3.2.2. The sizes of the sets of available orders varied from 60 to more than 5000. As such, the sizes of the sets hint towards how the workload in a single warehouse can be very different at different points in time. Each set of available orders was combined with the graph of the warehouse (see Section 3.3.2.3) as well as the size of the original batch to create a corresponding instance of the batch optimization problem. These instances were named *full instances* due to containing the full sets of available orders. Table 3.3 lists the names assigned to the aforementioned instances, the sizes of the sets of available orders, as well as the size of the originally batches.

Instance	Size ($\lvert\mathcal{L}\rvert$)	Batch size (N)
`orders60`	60	16
`orders107`	107	15
`orders223`	223	16
`orders769`	769	16
`orders1531`	1 531	20
`orders5176`	5 176	16

Table 3.3: List of instances of the batch optimization problem, created from snapshots of a customer's database.

3.3.2.5 Random subinstances

It is easy to observe that for larger instances of the batch optimization problem, a brute force search is infeasible. For example, for the `orders5176` instance above, the number of possible batches is equal to

$$\binom{5176}{16} \approx 1.239 \times 10^{46}.$$

To put this number into perspective: suppose that there are 1 billion computers that are checking 1 billion combinations per second each. It would then take approximately 3.93×10^{20} (393 quintillion) years to check all combinations. As such, it is infeasible to acquire optimal solutions using the brute force algorithms for all instances in Table 3.3. The lack of optimal solutions makes it more difficult to evaluate how well the relative randomized and greedy algorithms approximate a solution.

In an attempt to tackle this problem, it was assumed that the relative randomized and greedy algorithms achieve similar approximation ratios for small as well as large instances. Therefore, a large number of small *subinstances* was created based on the instances in Table 3.3. A total of 100 subinstances were created for each instance. Each subinstance was created by selecting 20 orders from the full order set. For each subinstance, the batch size was 16.

For each subinstance, the relative brute force algorithm was applied together with an exact algorithm for the shortest Hamiltonian path problem. As such, an optimal solution could be found for each subinstance, which acted as the basis for calculating approximation ratios for the other relative algorithms.

4
Implementation

This chapter cover the details of everything that has been implemeneted throughout the thesis. The majority of the thesis has been implemented in the C# programming language, with the exception being the Concorde toolkit described in Section 4.6. The chapter starts with descriptions of the library containing classes for working with graphs, the shortest Hamiltonian path problem and the batch optimization problem. The library has been split into three namespaces: `Graph`, `Shpp` and `Bop`, which are described in Section 4.2, Section 4.3, and Section 4.4 respectively.

Following the description of the library namespaces is an introduction to the evaluation program that uses the library and has been used to solve instances of the problems throughout the thesis. This is followed up by a motivation and description of the aforementioned Concorde toolkit. The chapter is then summed up with the descriptions of the testing suite and the benchmarking suite.

For the remainder of this report, the terms *algorithm* and *solver* refer to different things. Algorithm refers to a theoretical algorithm, such as those described in Section 3.2. Solver refers to an implementation of an algorithm, such as the `HeldKarpSolver` described in Section 4.3.2.

4.1 Unimplemented algorithms

Two algorithms described in Section 3.2.1 were not implemented. They are the Held-Karp successive approximation algorithm and Christofides's algorithm. The former was not implemented simply due to time constraints. The latter was not implemented as it appeared very difficult to correctly implement an algorithm for the minimum weight perfect matching problem, which is a subproblem solved as part of Christofides's algorithm.

4.2 The `Graph` namespace

This section covers the interfaces and algorithms that can be found in the Graph namespace. It includes a description of the common interface `IUndirectedGraph` as well as descriptions of some concrete implementations of the interface, namely the `WeightArrayGraph`, `GridGraph` and `SubGraph` classes.

4.2.1 The `IUndirectedGraph` interface

The main graph abstraction considered in the library is an interface called `IUndirectedGraph`. As the name implies, it is an interface that represents undirected graphs. It generally represents weighted graphs, and an unweighted graph can be represented by having unit weights. Listing 4.1 contains the definition of the `IUndirectedGraph` interface.

Listing 4.1 The definition of the `IUndirectedGraph` interface.

```
public interface IUndirectedGraph<TNodeId>
  where TNodeId : IEquatable<TNodeId>
{
  // Returns a collection of node identifers for all nodes
  // in the graph.
  IEnumerable<TNodeId> GetNodes();
  // Returns a collection of all edges in the graph.
  IEnumerable<UndirectedEdge<TNodeId> GetEdges();
  // Returns a collection of edges adjacent to the given `node`.
  IEnumerable<UndirectedEdge<TNodeId> GetEdges(TNodeId node);
  // Returns the weight of the given `edge`.
  long GetWeight(UndirectedEdge<TNodeId> edge);
}
```

The `IUndirectedGraph` interface has a type parameter for a *node identifier* type. A node identifier is some value that is used to refer to a specific node in a graph. For example, a node identifier can be an integer, a string, or an ordered tuple of integers. Only types that implement the `IEquatable` interface (defined in the standard library) can be used as identifiers, meaning that values of that type can be compared for equality among themselves as well as provide a hash value for themselves. Constraining the node identifier type in this way is not strictly necessary for representing graphs, but it simplifies implementations of some algorithms on graphs. Furthermore, classes implementing the `IUndirectedGraph` interface must ensure that all node identifiers are unique within an instance of the class.

Accompanying the `IUndirectedGraph` interface is a class called `UndirectedEdge` that represents an edge between two nodes in the graph. The class is a simple class that contains the node identifiers associated with the two nodes connected by the edge. An instance of the `UndirectedEdge` class is created by passing the two node identifiers to the constructor. `UndirectedEdge` implements the `IEquatable` interface, and additionally ensures that two instances that were created with the same node identifiers are considered equal, regardless of in which order the node identifiers were passed to the constructor.

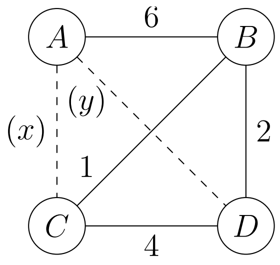
A limitation of the `IUndirectedGraph` interface is that it cannot represent multigraphs, i.e., graphs where there can be multiple edges between a pair of nodes.

4.2.2 The `WeightArrayGraph` class

The `WeightArrayGraph` class is an implementation of the `IUndirectedGraph` interface. The node identifiers of a `WeightArrayGraph` are integers $0, \ldots, |V| - 1$. A further semantic constraint of the `WeightArrayGraph` class is that it does not allow for self-edges. The reason for this constraint is that self-edges were not needed for the algorithms in which the `WeightArrayGraph` class was intended to be used.

As the name hints, the `WeightArrayGraph` class represents a graph using a *weight array*. The idea is similar to that of using a *weight matrix*, but the weights are stored in a one-dimensional array instead of a two-dimensional matrix. The reason for using a weight array is based on the fact that the graph is undirected. When using a weight matrix w to represent an undirected graph, it holds that $w_{i,j} = w_{j,i}$. Therefore, each weight is duplicated, leading to unnecessary memory usage. With a weight array, each weight is stored once.

The weight array stored in a `WeightArrayGraph` is interpreted according to a "lower row" scheme. The weights are arranged from beginning to end row-wise in a triangular shape to form the lower-right half of a weight matrix. Each row represents a node in the graph, as does each column. The element at row i, column j represents the weight of an edge from v_i to v_j. A `null` value encodes that there is no edge between v_i and v_j. As there are no self-edges in a `WeightArrayGraph`, there are no values on the diagonal, i.e., there are no elements at row i, column i for any value of i. Figure 4.1 contains an example graph along with its representation in a `WeightArrayGraph`.



(a) An example graph. The dashed edges are not present in the graph, but need to be represented in the weight array.

$$\begin{array}{c|cccc} & A & B & C & D \\ A & \cdot & 6 & (x) & (y) \\ B & 6 & \cdot & 1 & 2 \\ C & (x) & 1 & \cdot & 4 \\ D & (y) & 2 & 4 & \cdot \end{array}$$

(b) Weight matrix representation of the example graph. The gray elements are copies of the black elements and will not appear in the weight array.

$$[6, \underset{(x)}{\texttt{null}}, 1, \underset{(y)}{\texttt{null}}, 2, \quad 4]$$

(c) The weight array as it is stored in a `WeightArrayGraph`.

Figure 4.1: An example of how a graph is represented by a `WeightArrayGraph`.

As with all data structures, the `WeightArrayGraph` has a certain set of trade-offs with regards to computational complexity. To represent a graph with n nodes, the weight array needs $\frac{n(n-1)}{2}$ elements. Therefore, the space complexity of the `WeightArrayGraph` is $\mathcal{O}(|V|^2)$, regardless of the number of edges. As such, the `WeightArrayGraph` is ill-fitted for representing sparse graphs. However, the advantage of using a weight array is that it allows accessing specific edge weights in $\mathcal{O}(1)$

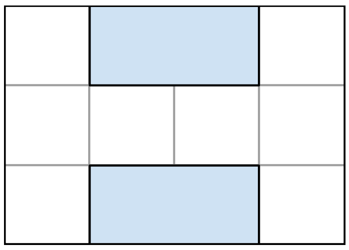
time. Furthermore, accessing all edges adjacent to a specific node takes $\mathcal{O}(|V|)$ time since the weight array has to be accessed $\mathcal{O}(|V|)$ times.

The `WeightArrayGraph` class exposes several methods in addition to those required by `IUndirectedGraph`. Two of those are the methods `InduceByNodes` and `InduceByEdges`. The methods are called on an existing instance of `WeightArrayGraph`, and are used to create induced subgraphs from it. The induced subgraphs are created from a set of nodes or a set of edges, respectively, as described in Section 2.2. For both methods, a new instance of `WeightArrayGraph` is returned along with a mapping of node identifiers from the original graph to the corresponding node identifiers in the new graph.

4.2.3 The `GridGraph` class

The `GridGraph` class is a special-purpose implementation of the `IGraph` interface. It is special-purpose in the sense that it cannot be used to represent arbitrary undirected, weighted graphs. Instead, it is focused on efficiently representing a specific type of graphs where the nodes are arranged in a 2D grid, and a node has edges with unit weights to its orthogonal neighbors. The node identifiers of a `GridGraph` are ordered pairs (r, c) of two integers. r and c corresponds to the row and column, respectively, of the identified node. Zero-based indexing is used for both r and c.

Internally, the `GridGraph` class uses a two-dimensional array (i.e., a matrix) of booleans to represent the graph. If `m` is the two-dimensional array of booleans and `m[i, j] == true`, then there is a node at row i, column j. Conversely, if `m[i, j] == false` then there is no node. Figure 4.2 contains an example of a simple warehouse layout and how it is represented internally in `GridGraph`.



(a) A grid representation of a simple warehouse layout.

1	0	0	1
1	1	1	1
1	0	0	1

(b) The two-dimensional array of booleans used to represent the warehouse in a `GridGraph`.

Figure 4.2: An example of a simple warehouse layout and its representation in a `GridGraph`.

The `GridGraph` class is intended to be a direct implementation of the graph model of a warehouse, as described in Section 3.1.1. As such, the `GridGraph` class is designed to be the "entrypoint" for solving the batch optimization problem in the context of a real-world warehouse. For example, after creating an instance of `GridGraph` that represents the warehouse, one can use the `ShortestPathCompleteGraph` (see Section 4.2.4) function to acquire a complete graph containing the shortest paths among all pairs of nodes in the warehouse.

Along with implementing the operations specified by `IUndirectedGraph`, the implementation of `GridGraph` also includes code to parse a textual representation of a 2D grid into a corresponding instance of `GridGraph`. The textual representation allows only two characters: a full stop (`.`) representing a node, and a hash (`#`) representing a wall. A text file consisting of n lines each with exactly m characters can be parsed into an instance of `GridGraph` with n rows and m columns. The first line in the file corresponds to row 0, the second line corresponds to row 1, and so forth. Likewise, the first character on each line corresponds to column 0, the second character corresponds to column 1, and so forth.

The advantage of the `GridGraph` graph is that it can efficiently be queried. Determining whether there is an edge between two nodes can be done in $\mathcal{O}(1)$ time: there are edges between all orthogonally adjacent nodes, so by comparing the node identifiers of the two nodes, which is a $\mathcal{O}(1)$ operation, one can determine whether there is an edge between them. Furthermore, accessing all edges adjacent to a specific node can also be done in $\mathcal{O}(1)$ time: one simply needs to check whether the orthogonally adjacent positions in the graph contain nodes, and if so, there is an edge to the adjacent node.

The space complexity of a `GridGraph` instance with R rows and C columns is $\mathcal{O}(RC)$. Note that the space complexity is independent of the number of nodes in the graph. As such, whether a complexity of $\mathcal{O}(RC)$ is good or not depends on the graph. For a graph with few nodes, i.e., $|V| \ll RC$, then the space complexity is bad since a lot of space is wasted to store the absence of nodes. Conversely, for a graph with many nodes, i.e., $|V| \approx RC$, the space complexity can be considered good since very little space is wasted to store the absence of nodes.

4.2.4 The `ShortestPathCompleteGraph` function

The `ShortestPathCompleteGraph` function is a static function (i.e., a function that is not tied to a specific instance of a class) in a class called `Algorithms`. The function takes as parameter an instance of `IUndirectedGraph` and calculates for each pair of nodes the length of a shortest path between those nodes. A necessary pre-condition is therefore that the input graph is connected, which is checked by the function before doing any calculations. `ShortestPathCompleteGraph` implements Dijkstra's algorithm [10] to calculate the shortest paths. The function returns two values: a `WeightArrayGraph` representing a complete graph where the edge weights are the aforementioned lengths; and a mapping from the original node identifiers to the node identifiers of the `WeightArrayGraph`.

The primary use case for `ShortestPathCompleteGraph` is to take a `GridGraph` describing the layout of a warehouse and return a complete graph that contains the shortest distances between any two points in the warehouse, as described in Section 3.1.3. The complete graph can then be used as the graph in an instance of the batch optimization problem.

4.2.5 The `IsConnected` function

Similarly to the `ShortestPathCompleteGraph` function, the `IsConnected` function is a static function in the class called `Algorithms`. The function takes as parameter an instance of `IUndirectedGraph` and returns a boolean denoting whether the graph is connected or not. The main use case of `IsConnected` is to verify that a graph that has been passed to `ShortestPathCompleteGraph` (see Section 4.2.4) fulfills the pre-condition of being connected.

The property of being connected is defined as there existing a path between any pair of nodes v, u. Since the context is undirected graphs, it then holds that if there exists a path from v to u, then there also exists a path from u to v.

The implementation of `IsConnected` is a simple breadth-first search starting from an arbitrary node in the input graph. When a node is visited for the first time, it is marked as visited and all its adjacent nodes are added to the queue of nodes to visit. When visiting a node that has already been marked as visited, it is skipped. `IsConnected` returns `true` if all nodes have been marked as visited when the breadth-first search terminates.

4.3 The `Shpp` namespace

The `Shpp` namespace contains interfaces and classes related to representing and solving instances of the shortest Hamiltonian path problem. This section contains descriptions of these interfaces and classes: what their purposes are, how they are used, and some details about their implementations.

4.3.1 The `IShppSolver` interface

The `IShppSolver` interface is used to represent algorithms that solve the shortest Hamiltonian path problem. Along with the `IShppSolver` interface are two classes, `ShppInstance` and `ShppSolution`, that are described in Section 4.3.1.1 and Section 4.3.1.2, respectively. Listing 4.2 contains the definition of the interface.

Listing 4.2 The definition of the `IShppSolver` interface.

```
public interface IShppSolver
{
  // Solves an instance of the shortest Hamiltonian path problem.
  ShppSolution Solve(ShppInstance instance);
}
```

4.3.1.1 The `ShppInstance` class

The `ShppInstance` class represents an instance of the shortest Hamiltonian path problem. It holds all information that defines an instance of the shortest Hamiltonian path problem: a graph together with node identifiers specifying the start and end nodes. It is a simple class in the sense that it only holds data and provides access to it, but does not provide any operations on the data.

`ShppInstance` requires that the graph is specified using a `WeightArrayGraph`, and therefore the start and end nodes are specified using integer node identifiers. In theory, any implementation of the `IUndirectedGraph` interface could have been used. The choice to specifically use `WeightArrayGraph` was made since instances of the shortest Hamiltonian path problem requires that the graph is complete, and `WeightArrayGraph` provides an efficient way to represent complete graphs.

4.3.1.2 The `ShppSolution` class

The `ShppSolution` class represents a solution to a instance of the shortest Hamiltonian path problem. Like `ShppInstance`, the class is only used to hold data and provide access to it, and does not provide any operations on the data. Currently, the `ShppSolution` class holds only an integer containing the path length of the solution. The reason for having a class instead of simply using integers directly is that in the future it should be possible to extend the `ShppSolution` class to also hold more information, such as the actual path found.

4.3.2 The `HeldKarpSolver` class

The `HeldKarpSolver` class is an implementation of the `IShppSolver` interface. It implements the adapted Held-Karp algorithm described in Section 3.2.1.1.

The implementation is a rather direct translation of the description of the algorithm. Internally, arrays of booleans are used to represent the subsets S, where a `true` value indicates that a node is included in the subset, and a `false` value indicates that the node is not included. The C function is represented using a dictionary `C`. The keys of `C` are the boolean arrays, corresponding to the S argument of the function. The values of `C` are integer arrays, where the l argument of the function is used as an index into the integer array. As such, the value of $C(S, l)$ can be accessed using `C[S][l]`, where `S` is a boolean array and `l` is an integer.

The `C` dictionary is built in a bottom-up fashion. All subsets are placed in a queue that is processed one element at a time. Initially, subsets containing only a single node (i.e., boolean arrays with a single `true` value) are added to the queue. Processing the queue is done in the following fashion. First, a subset `S` is removed from the head of the queue. If the key `S` has already been stored in `C`, then `S` is skipped. Otherwise, all values of `C[S][l]` are calculated in the fashion described in Section 3.2.1.1 using values that has been previously been stored in `C`. Finally, new subsets are created by adding one additional node to `S`. The new subsets are then added to the tail of the queue. After all elements in the queue have been processed, the length of the shortest path can be calculated.

4.3.3 The `NearestNeighborSolver` class

The `NearestNeighborSolver` class is another implementation of the `IShppSolver` interface. It implements the adapted nearest neighbor algorithm described in Section 3.2.1.3.

Internally, the implementation stores the path so far, along with a boolean array indicating which nodes have been visited. Furthermore, the node identifier of the current node is stored. Iterating in a loop, the implementation finds all edges adjacent to the current node, and order them in an ascending order according to their weights. The edges are then filtered so that only edges leading to unvisited nodes that are not the end node are considered. After this, the first edge is picked as the edge to travel along, as it is the edge with the lowest weight. The node that the edge leads to is added to the end of the path, is marked as visited, and set as the current node.

When the loop terminates, there are no more nodes to visit except the end node. The end node is therefore added to the end of the path, and the length of the path is returned as the solution.

4.3.4 The `TreeDoublingSolver` class

The `TreeDoublingSolver` class is another implementation of the `IShppSolver` interface. It implements the adapted tree doubling algorithm described in Section 3.2.1.4.

The implementation of `TreeDoublingSolver` follows the description of the algorithm rather directly. There are two important functions implemented in `TreeDoublingSolver`: `MinimumSpanningTree` and `ShortcutTour`. These functions are used together to eventually build the path whose length is returned.

The `MinimumSpanningTree` function builds a minimum spanning tree of the graph in instance. It returns a set of edges that corresponds to the minimum spanning tree. Internally, `MinimumSpanningTree` implements Prim's algorithm [25] for building a minimum spanning tree. A hash set is used to keep track of unvisited nodes. Another hash set is used to keep track of the edges included in the minimum spanning tree. As long as the set of unvisited nodes contains elements, an edge of minimum weight is found that connects the partially constructed minimum spanning tree with an unvisited node. That edge is then added to the minimum spanning tree and the node is marked as visited. The set of edges returned by `MinimumSpanningTree` is used to create an induced subgraph containing only those edges. The induced graph then contains all the nodes in the original graph but only the edges that comprise the minimum spanning tree.

The induced graph is passed to the `ShortcutTour` function, which constructs a Eulerian path that visits all nodes in the graph. The Eulerian path is constructed using a simple depth-first search. The search starts at the start node. During the search, a list of nodes contains the partially constructed path. When visiting a node, the list of nodes is searched for the node. If the node is found, it is skipped. Otherwise, the node is appended to the path and the node's children is searched.

When the Eulerian path is returned from `ShortcutTour`, the end node is moved to the end of the list, regardless of where in the path it was located earlier. The length of this new path is then returned as the solution.

4.3.5 The `CachingSolver` class

The `CachingSolver` class is an implementation of the `IShppSolver` interface. However, instead of directly implementing some algorithm for the shortest Hamiltonian path problem, it wraps another `IShppSolver` and provides a caching mechanism. The caching mechanism uses instances of the shortest Hamiltonian path problem as keys, and stores solutions to those instances as values. The intent is to solve instances of the shortest Hamiltonian path problem only once and be able to reuse the solutions later to save time.

When constructing a `CachingSolver`, another implementation of the `IShppSolver` interface must be supplied to the constructor. This other `IShppSolver` is referred to as the *inner solver*.

The caching mechanism is provided by using a dictionary where `ShppInstance` values are used as keys and `ShppSolution` values are used as the corresponding values. When the `Solve` method of a `CachingSolver` instance is called, the dictionary is first checked to see whether the instance given as the argument to `Solve` has already been stored in the dictionary. If so, the value is retrieved from the dictionary and immediately returned. Otherwise, the `CachingSolver` calls the `Solve` method of the inner solver. When the solution is returned from the inner solver, it is stored in the dictionary, and subsequently returned.

The purpose of the `CachingSolver` class is to be used in a context where there is a significant possibility that equivalent instances of the shortest Hamiltonian path problem needs to be solved multiple times. One such context is the context of solving the batch optimization problem. It is entirely possible that the algorithms described in Section 3.2.2 will evaluate different combinations of orders that result in equivalent sets of nodes to visit. If so, the corresponding instance of the shortest Hamiltonian path problem would be solved several times. In those situations, the `CachingSolver` can be used to reduce computation time.

4.4 The `Bop` namespace

This section covers the `Bop` namespace. It includes descriptions of the interfaces and classes within, used for representing and solving instances of the batch optimization problem: what their specific purposes are, how they are used, and some details about their implementations.

4.4.1 The `IBopSolver` interface

The `IBopSolver` interface is used to represent algorithms that solve the batch optimization problem. It is similar in design to the `IShppSolver` interface, in that it specifies a single method and is accompanied by the two classes `BopInstance` and `BopSolution` that are described in Section 4.4.1.1 and Section 4.4.1.2, respectively. Listing 4.3 contains the definition of the `IBopSolver` interface.

Listing 4.3 The definition of the `IBopSolver` interface.

```
public interface IBopSolver
{
  // Solves an instance of the batch optimization problem.
  BopSolution Solve(BopInstance instance);
}
```

4.4.1.1 The `BopInstance` class

The `BopInstance` class represents an instance of the batch optimization problem. Much like the `ShppInstance` holds all information that defines an instance of the shortest Hamiltonian path problem, `BopInstance` holds all information that defines an instance of the batch optimization problem: a graph, node identifiers specifying the start and end nodes, an integer representing the batch size N, and a list of orders. In the list of orders, each order is represented by a collection of node identifiers.

`BopInstance` mandates using a `WeightArrayGraph` to represent the graph, and therefore all node identifiers (start node, end node, orders) are integers. Like with the `ShppInstance` class, any implementation of `IUndirectedGraph` could have been used in theory.

Unlike `ShppInstance`, the implementation of `BopInstance` contains operations on the data it holds. `BopInstance` has a method called `Evaluate` which is used to evaluate a batch of orders in the context of a specific instance of the batch optimization problem. The `Evaluate` method takes a reference to an implementation of `IShppSolver` and a set of indices. The indices refer to orders in the list of orders held by the `BopInstance`. The `Evaluate` method uses the indices to retrieve the corresponding orders, and then creates a set of all nodes covered by those orders, along with the start and end nodes also held by the `BopInstance`. From this set of nodes, an induced subgraph is created from the graph held by the `BopInstance`. The induced subgraph is combined with the start and end nodes to create a `ShppInstance`. The `ShppInstance` is then used as argument to the `Solve` method of the referenced `IShppSolver`, and the solution is returned to the caller of `Evaluate`.

4.4.1.2 The `BopSolution` class

The `BopSolution` class represents a solution to an instance of the batch optimization problem. Like `BopInstance`, it is a simple class that only holds and provides access to its data.

A solution to an instance of the batch optimization problem is a batch of orders. Therefore, an instance of `BopSolution` should represent such a batch. Currently, however, the only data held by an instance of `BopSolution` is an integer. The integer is the length of a path that visits all nodes included in the batch. Therefore, it is currently not possible to access the batch that constitutes a solution, but it is possible to use the path length to evaluate the quality of the solution.

4.4.2 The `GreedySolver` class

The `GreedySolver` class is an implementation of the `IBopSolver` interface. It implements the relative greedy algorithm described in Section 3.2.2.3.

As the greedy algorithm is relative to some algorithm for solving the shortest Hamiltonian path problem, an instance of `GreedySolver` needs a reference to an implementation of `IShppSolver`. This is enforced by requiring an argument of type `IShppSolver` in the constructor for `GreedySolver`.

Internally, `GreedySolver` refers to orders using their indices into the order list held by the `BopInstance`. Two sets of indices are used for each invocation of the `Solve` method. One set of indices selected to be part of the solution, and one set of indices that are available to choose from. Initially, the set of selected indices is empty, and the set of available indices contains all indices. The main work of the algorithm is implemented in a loop. In each iteration of the loop, each of available indices are combined with the current set of selected indices to create a candidate set of selected indices. The candidate sets are then used as arguments to the `BopInstance`'s `Evaluate` method together with the `IShppSolver` given in the constructor to `GreedySolver`. The index that resulted in the candidate set with the lowest value returned from `Evaluate` is determined to be the "best index". The best index is then added to the set of selected indices, as well as removed from the set of available indices. Afterwards, the loop begins its next iteration. The loop terminates when the size of the set of selected indices is equal to the batch size of the `BopInstance`.

After the loop has terminated, the `Evaluate` method is called with the final set of selected indices as an argument. The integer value of the solution is used to create a `BopSolution` which is then returned.

4.4.3 The `BruteForceSolver` class

The `BruteForceSolver` class is an implementation of the `IBopSolver` interface. It implements the relative brute force algorithm described in Section 3.2.2.2.

Like with the `GreedySolver` class, the constructor of `BruteForceSolver` takes as parameter a reference to an implementation of `IShppSolver`.

Internally, `BruteForceSolver` uses a lazy iterator to build all possible batches of orders. The iterator is then traversed over, and each batch is evaluated using the `Evaluate` method together with the `IShppSolver` given in the constructor. The minimum value returned by `Evaluate` is then used for constructing the solution.

4.4.4 The `RandomSolver` class

The `RandomSolver` class is an implementation of the `IBopSolver` interface. It implements the relative randomized algorithm described in Section 3.2.2.4.

Like with the `GreedySolver` and `BruteForceSolver` classes, the constructor of `RandomSolver` takes as argument a reference to an implementation of `IShppSolver`.

The randomly selected batches are created based on a list of integers $0, \dots, |\mathcal{L}|-1$, acting as indices into the list of orders. The list is randomly shuffled, and the first

Listing 4.4 Example invocation of the evaluation program for an instance of the shortest Hamiltonian path problem.

```
BatchOptimizationEvaluation.exe \
  # Path to a JSON file containing an instance of the
  # shortest Hamiltonian path problem.
  --instance path/to/shpp_instance.json \
  # The shortest Hamiltonian path problem solver to use.
  # `nn` corresponds to the `NearestNeighborSolver`.
  --shpp-solver nn \
  # Path to where the solution should be written.
  --output path/to/output.txt
```

N elements of the shuffled list is used to create a batch of N orders. This process is repeated until k batches have been created.

When the batches have been created, a simple loop evaluates them using the `Evaluate` method. The minimal value returned from `Evaluate` is then used to create a `BopSolution` which is then returned from the `Solve` method.

4.5 Evaluation program

The library described in Section 4.2 through Section 4.4 provides implementations of various concepts but does not provide a convenient way to execute the implementations for arbitrary inputs. For this purpose, a simple evaluation program was created. The evaluation program is a command-line interface (CLI) that links with the library and uses it to solve instances of the shortest Hamiltonian path problem and the batch optimization problem. Command-line options given to the program are used to specify the problem instance, which solver(s) to use, whether to use caching, and where the output should be written.

The evaluation program uses JSON[1] to encode an instance. The path to a JSON file is used as a command-line option to specify which instance the evaluation program should use, regardless of whether it is a shortest Hamiltonian path problem instance or a batch optimization problem instance. The format of the JSON files closely (but not exactly) match the structure of the `ShppInstance` and `BopInstance` classes, respectively.

The output of the program is a single number that contains the path length of the solution returned (either `ShppSolution` or `BopSolution`). The path length is written to a file whose path is specified as a command-line option.

Listing 4.4 contains an example invocation of the evaluation program for an instance of the shortest Hamiltonian path problem. Similarly, Listing 4.5 contains an example invocation for an instance of the batch optimization problem.

[1] `https://www.json.org`

Listing 4.5 Example invocation of the evaluation program for an instance of the batch optimization problem.

```
BatchOptimizationEvaluation.exe \
  # Path to a JSON file containing an instance of the
  # batch optimization problem.
  --instance path/to/bop_instance.json \
  # The batch optimization problem solver to use.
  --bop-solver greedy \
  # The shortest Hamiltonian path problem solver to use.
  # `td` corresponds to the `TreeDoublingSolver`.
  --shpp-solver td \
  # Whether to enable caching by wrapping the shortest
  # Hamiltonian path solver in a `CachingSolver`.
  --cache true \
  # Path to where the solution should be written.
  --output path/to/output.txt
```

4.6 Concorde toolkit

Over the course of the thesis, a toolkit was built for more conveniently working with a travelling salesperson problem solver by the name of Concorde [8]. The toolkit started as a single script that constructed a travelling salesperson problem instance, used it as input to Concorde, and then parsed the length of the solution from Concorde's output. Eventually, more scripts were added, resulting in a collection of scripts to work with the travelling salesperson problem and the batch optimization problem. This collection of scripts is what is referred to as the *Concorde toolkit*.

The toolkit is implemented in the Go programming language[2] and (more or less) specific to the Linux operating system. It is therefore not directly compatible with the library described in Section 4.2 through Section 4.4. However, the toolkit uses JSON files formats similar to those described in Section 4.5.

4.6.1 Motivation

The `HeldKarpSolver` was implemented in an early stage of the thesis, since optimal solutions to the shortest Hamiltonian path problem were needed for calculating approximation ratios. The implementation of `HeldKarpSolver` turned out to be very slow: during experimentation, an instance with 22 nodes took 9 minutes to solve for `HeldKarpSolver`. It was deemed infeasible to improve the efficiency of `HeldKarpSolver` significantly enough for it to be used to acquire optimal solutions.

The inefficieny of `HeldKarpSolver` would be an even larger problem when solving the batch optimization problem, since the algorithms in Section 3.2.2 would all require solving a large number of instances of the shortest Hamiltonian path problem for a single instances of the batch optimization problem. As such, there existed a

[2]`https://golang.org/`

need for some other tool that could be used to optimally solve instances of the shortest Hamiltonian path problem. Furthermore, it was determined early in the thesis that there was a need for the random subinstances described in Section 3.3.2.5, and a tool for acquiring optimal solutions to them. The combination of these needs motivated the search for an efficient travelling salesperson problem solver and a set of tools for interacting with it, automating the task of invoking it and reading solutions from its output.

The search for an efficient travelling salesperson problem solver quickly yielded Concorde as a promising candidate. Concorde does expose an interface for interacting with it from software in the form of a callable library. Unfortunately, using ths interface requires significant experience with the C programming language; something the authors do not possess. However, invoking the Concorde exectuable does not require complicated command-line arguments, and its debugging output follows a fairly consistent format. Therefore, it was simple to write scripts for Concorde, and eventually it was decided that the scripts should be combined into a more comprehensive toolkit.

4.6.2 The Concorde solver

The Concorde solver is a solver for the travelling salesperson problem developed primarily by Cook at the University of Waterloo [8]. It is written in the ANSI C programming language. The most recent version was released in 2003 [9]. Even though the code is old, Concorde still runs reliably on modern computers and operating systems.

Concorde uses several algorithmic techniques to optimally solve the travelling salesperson problem, such as a branch-and-bound search. Furthermore, the code is heavily optimized to minimize execution time. The benchmarks for Concorde [7] shows that Concorde can solve instances of the travelling salesperson problem containing hundreds of nodes in a short amount of time.

The Concorde executable takes a file in the TSPLIB format [26] containing an instance of the *symmetric* travelling salesperson problem. It produces output in several ways: for example, an optimal tour in the form of an output file, as well as debugging output written to the Concorde process' standard output stream. The debugging output contains the length of the optimal tour.

Listing 4.6 contains an example of how the Concorde executable is invoked.

Listing 4.6 Example invocation of the Concorde executable.

```
./concorde \
  # Path to a file where the optimal tour should be written.
  -o path/to/solution.sol \
  # Delete temporary files on completion.
  -x \
  # Path to a file containing a travelling salesperson problem
  # instance in the TSPLIB format.
  path/to/instance.tsp
```

Listing 4.7 contains the output produced by the Concorde executable for an example invocation. The input file `a280.tsp` contains the `a280` instance from TSPLIB [26]. The length of an optimal solution can be found on line 23.

Listing 4.7 Example debugging output from an invocation of the Concorde executable. The file `a280.tsp` contains the `a280` instance from TSPLIB. The length of the optimal solution can be found on line 23.

```
1  ./concorde -o a280.sol -x a280.tsp
2  Host: hostname  Current process id: 1061383
3  Using random seed 1590855753
4  Problem Name: a280
5  drilling problem (Ludwig)
6  Problem Type: TSP
7  Number of Nodes: 280
8  Rounded Euclidean Norm (CC_EUCLIDEAN)
9  Set initial upperbound to 2579 (from tour)
10   LP Value  1: 2557.000000  (0.04 seconds)
11   LP Value  2: 2575.300000  (0.11 seconds)
12   LP Value  3: 2576.750000  (0.17 seconds)
13   LP Value  4: 2578.000000  (0.24 seconds)
14   LP Value  5: 2578.000000  (0.38 seconds)
15   LP Value  6: 2578.117450  (0.52 seconds)
16   LP Value  7: 2578.171429  (0.62 seconds)
17   LP Value  8: 2578.171429  (0.73 seconds)
18 New lower bound: 2578.171429
19 Final lower bound 2578.171429, upper bound 2579.000000
20 Exact lower bound: 2578.171429
21 DIFF: 0.000000
22 Final LP has 347 rows, 556 columns, 2202 nonzeros
23 Optimal Solution: 2579.00
24 Number of bbnodes: 1
25 Total Running Time: 0.84 (seconds)
```

4.6.3 Caching mechanism

To decrease execution times, a caching mechanism similar to that of `CachingSolver` was implemented in the Concorde toolkit. The purpose of the cache is to reduce the number of times Concorde is invoked, since each such invocation can incur a significant amount of execution time.

Concorde is a travelling salesperson problem solver, and therefore the keys of the cache were defined to be travelling salesperson problem instances. For ease of implementation, a travelling salesperson problem instance was considered to be a weight array (see Section 4.2.2) representing the graph for which a tour should be found. A constraint of Go's builtin dictionaries is that arrays cannot be used as keys. To work around this problem, the array was converted to a "stable" string representation, since strings can be used as dictionary keys. The string representation is stable in the sense that two arrays that are element-wise equal will result in equal string representations.

4.6.4 Examples of scripts in the toolkit

This section contains descriptions of three scripts from the toolkit. Each script is described in terms of how it should be invoked, what format the input should be, and some other technical details. An example invocation is included for each described script.

4.6.4.1 The `generate-random-index-sets` script

In the toolkit, *index sets* are frequently occurring. The definition of an index set is a set of *indices*, i.e., non-negative integers. An index set is not very useful on its own. Instead, it is intended to be used together with a data structure that can be accessed using indices, such as an array. In the context of the shortest Hamiltonian path problem and the batch optimization problem, an index set could, for example, be a set of node identifiers used for inducing a graph, or be a set of indices into a list of orders.

The toolkit contains a script called `generate-random-index-sets` that can be used to generate index sets. The purpose of the script is to allow the creation of many index sets of different sizes with a single invocation.

Listing 4.8 contains an example invocation of the `generate-random-index-sets` script. The output of the invocation in Listing 4.8 would yield a total of 18 index sets, distributed over 3 set sizes. These index sets could, for example, be used to create subinstances of a batch optimization problem instance by interpreting the indices as indices into a list of orders.

Listing 4.8 Example invocation of the `generate-random-index-sets` script in the Concorde toolkit. This invocation will generate a total of 18 sets: 6 with 10 elements, 6 with 15 elements, and 6 with 30 elements.

```
./toolkit generate-random-index-sets \
  # The lower bound for indices. Inclusive.
  --start 0 \
  # The upper bound for indices. Exclusive.
  --end 25 \
  # Generate sets that have 10 elements, 15 elements,
  # and 30 elements.
  --sizes 10,15,30 \
  # For each size specified above, generate 6 index
  # sets.
  --count 6 \
  # Path to a file where the output should be written
  # in JSON format.
  --output index_sets.json
```

4.6.4.2 The `solve-shpp` script

The `solve-shpp` script was one of the first scripts written for the Concorde toolkit. It takes an instance of the shortest Hamiltonian path problem and reduces it to an instance of the travelling salesperson problem as described in Section 2.1.1. The travelling salesperson problem instance is then used as input to the Concorde executable, and the length of an optimal solution is returned.

The shortest Hamiltonian path problem instance is given as a set of command-line options. One option defines the graph by specifying a path to a JSON file containing a weight matrix. Two options are then used to specify the node identifiers of the start and end nodes. The weight matrix is combined with the start and end nodes to reduce the shortest Hamiltonian path problem instance to an instance of the travelling salesperson problem, which is then formatted in the TSPLIB format and written to a temporary file. The Concorde executable is then called with the path of the temporary file as input, and the length of an optimal solution is parsed from the Concorde's debugging output (see Section 4.6.2). The length is then written to an output file.

Listing 4.9 contains an example invocation of the `solve-shpp` script.

Listing 4.9 Example invocation of the `solve-shpp` script.

```
./toolkit solve-shpp \
  # Uses the first node as start node.
  --start 0 \
  # Uses the second node as end node.
  --end 1 \
  # Specifies a path where the optimal length
  # should be written.
  --output path/to/solution.txt \
  # Specifies the path where the weight matrix
  # should be read from.
  path/to/matrix.json
```

4.6.4.3 The `solve-bop` script

The `solve-bop` script implements the brute force and greedy algorithms from Section 3.2.2.2 and Section 3.2.2.3, respectively. These implementations use Concorde as the underlying shortest Hamiltonian path problem solver. A batch optimization problem instance is specified using a set of command-line options. The output is the length of a shortest path that visits all nodes in the batch constructed by the batch optimization problem solver. In the case of the brute force algorithm, the solution is optimal, and therefore so is the path length.

The `solve-bop` takes the same command-line options as the `solve-shpp` script described in Section 4.6.4.2. Three additional command-line options are required for the `solve-bop` script: one to specify the list of orders, one to specify the batch size, and one to specify which batch optimization problem algorithm to use. The list of orders is specified using a path to a JSON file containing a list of orders, where each order is a list of node identifiers of the nodes that the order requires must be visited.

Listing 4.10 contains an example invocation of the `solve-bop` script.

Listing 4.10 Example invocation of the `solve-bop` script.

```
./toolkit solve-bop \
  # Specifies the path where the weight
  # matrix should be read from.
  --matrix path/to/matrix.json \
  # Uses the first node as start node.
  --start 0 \
  # Uses the second node as end node.
  --end 1 \
  # Specifies the path where the order list
  # should be read from.
  --order-list path/to/orders.json \
  # Uses 16 as the batch size.
  --batch-size 16 \
  # Uses the greedy algorithm for the batch
  # optimization problem.
  # Using "brute_force" instead causes the
  # brute force algorithm to be used.
  --solver greedy \
  # Specifies a path where the path length
  # should be written.
  --output path/to/solution.txt
```

4.7 Test suite

The library is equipped with a suite of tests for the different classes in the library. The test suite consists of mainly property-based tests but also contain a couple of regression tests. This section introduces property-based testing and the `FsCheck` library that has been used for the implementation in C#. It also includes a couple of examples of the tested properties. This is followed by an introduction to regression tests and how they are used in the library.

4.7.1 Property-based testing

Property-based testing of software is a style of automatic software testing. Traditional software testing often uses unit tests (essentially a pair of a specific input and the expected output) as opposed to property-based testing. In property-based testing test cases are generated randomly and makes sure that different properties are asserted for the output of every function call using the generated data. The main value proposition of using property-based testing is that by generating many random test cases it is likely to find corner cases that normal unit tests would not have found.

As an example of how property-based testing can be used, consider a function `reverse` that reverses a list. A property of list reversal is that if a list is reversed and then reversed again, the original list is returned. This is a property that is suitable for use in property-based testing: if `xs` is a list, then `reverse(reverse(xs))` should be equal to `xs`.

4.7.2 The `FsCheck` library

The `FsCheck` library is essentially a port of the QuickCheck [6] library to the .NET platform. Originally it was implemented in the F# language[3], however as F# provides interoperability with C# it is also usable for this thesis. Similar to QuickCheck `FsCheck` has support for custom test data generation, conditional properties, test case shrinking and more.

Consider again the `reverse` function from Section 4.7.1, and the property that reversing a list twice should return the original list. Listing 4.11 contains an example of how `FsCheck` can be used to test this property.

Listing 4.11 Example of a property-based test written using `FsCheck`. The test verifies that calling `reverse` twice on any array of integers returns the original array.

```
[TestMethod]
public void Reverse_Twice_ReturnsOriginal()
{
    // `arrays` generates non-null integer arrays of any length.
    var arrays = Arb.Default.Array<int>()
        .Filter(array => array != null);
    // Generates 100 arrays at random and verifies that
    // applying `reverse` twice yields an array that is
    // element-wise equal to the original array.
    Prop.ForAll(arrays, array =>
        reverse(reverse(array)).SequenceEqual(array))
        .QuickCheckThrowOnFailure();
}
```

4.7.3 Examples of tested properties

The test suite consists of a total of 46 test methods, out of which 42 are property-based. Following will be a brief description of a couple of tests in order to give a grasp of what they might look like. Listing 4.12 containing the `New_NonEmptyArray_Ok` method is a simple property-based test method. It makes sure that the `WeightArrayGraph` constructor sucessfully creates graphs when randomly generated data of the expected size is given as input.

[3] `https://fsharp.org/`

Listing 4.12 The `New_NonEmptyArray_Ok` test method.

```
[TestMethod]
public void New_NonEmptyArray_Ok()
{
    // WeightArrayGen generates random weight arrays with correct size
    var arb = WeightArrayGen().ToArbitrary();
    // Generats 100 tests using random data and makes sure that
    // the WeightArrayGraph constructor does not throw an error.
    Prop.ForAll(arb, weights => new WeightArrayGraph(weights))
        .QuickCheckThrowOnFailure();
}
```

Listing 4.13 contains the `IsConnected_AtLeastTwoComponents_IsFalse` method which is another property-based test method. It is similar to the `New_NonEmptyArray_Ok` in structure but differs on a few points. In this scenario a more complex generator is necessary in order to avoid mirroring the functionality of the tested method in the generation of test data. For this scenario the input graphs are constructed from two separately generated graphs that are combined into a single `WeightArrayGraph` instance while not adding any edges between the two. This guarantees that the graph contains at least two separate components. The test also differs from the `New_NonEmptyArray_Ok` test in that it tests the expected functionality of the `IsConnected` method rather than that of a constructor.

Listing 4.13 The `IsConnected_AtLeastTwoComponenets_IsFalse` test method.

```
[TestMethod]
public void IsConnected_AtLeastTwoComponents_IsFalse()
{
    // DisconnectedWeightArrayGraphGen randomly generates
    // disconnected graphs.
    var disconnectedGraphArb = DisconnectedWeightArrayGraphGen(
      WeightArrayGraphTest.WeightArrayGraphGen()).ToArbitrary();
    // Generats 100 tests using random data and makes sure that
    // the IsConnected method always returns false.
    Prop.ForAll(disconnectedGraphArb, disconnectedGraph =>
    !Algorithms.IsConnected(disconnectedGraph))
        .QuickCheckThrowOnFailure();
}
```

4.7.4 Regression tests

In addition to the property-based tests, a set of regression tests was created. The regression tests are in the style of traditional unit tests, where an input is paired with an expected output.

The property-based tests and the regression tests complement each other by having different purposes. The purpose of the property-based tests is to help verify the correctness of most classes in the library. The regression tests are only for the solver classes, and their purpose is different: to detect changes in the solver's behaviors that do not violate the correctness of the solvers.

For example, consider the `NearestNeighborSolver` class. If, at some point, there are two or more nodes which are of minimal distance from the current node, then there is a tie which must be broken in order to pick one of the nodes. The logic for breaking such a tie is left as an implementation detail, and does not affect the correctness of the `NearestNeighborSolver` class. However, the outcome of the tie may affect the subsequently constructed solution. The regression tests are designed to detect such changes the implementation details, where the solution is affected but not necessarily incorrect.

The set of regression tests is divided by solver. There is one set of regression tests for `NearestNeighborSolver`, one set of regression tests for `TreeDoublingSolver`, and so forth. Each set of tests is based on the sets of instances for the shortest Hamiltonian path problem and batch optimization problem described in Section 3.3.1 and Section 3.3.2.4, respectively. The instances were solved with the corresponding solver using the evaluation program described in Section 4.5. The instance and the solution are then used to create a pair of input and expected output, respectively. In the execution of a set of regression tests, the instance is solved with the corresponding solver and the output is compared to the expected output. If the output differs from the expected output, it is an indication that some behavior has changed.

As stated earlier, the regression tests are not designed to help verify the correctness of the solvers, but rather to help detect changes in their behavior. If a regression test fails, it is not necessarily an indication that the solvers contain bugs. The value proposition of the regression tests is to make a maintainer of the library *aware* that behavior has changed. The expected outputs of the regression tests can be updated at any time, if the change in behavior was intended, or are for some other reason acceptable. For example, the tiebreaking logic described above may be purposely changed, in which case the behavior may change as a result.

4.8 Benchmark suite

The library is accompanied by a benchmarking suite, used to measure the performance of the library. The performance is measured in terms of *quality* and *runtime*.

Quality corresponds to the path length returned by a solver. It is measured both as the path lengths themselves, as well as the approximation ratio achieved by comparing the path lengths with an exact solution. Section 4.8.1 describes how the quality benchmarks were implemented, as well as the exact set of benchmarks that were run.

Runtime corresponds to the execution time of a solver. It is measured as elapsed "wall-clock" time for an invocation of the `Solve` method for a solver. Section 4.8.2 describes `BenchmarkDotNet`, the benchmarking framework used to create runtime benchmarks, as well as lists the exact set of benchmarks that were run.

For the remainder of the report, the following notation is used:

- `nn` refers to `NearestNeighborSolver`.

- `td` refers to `TreeDoublingSolver`.
- `concorde` refers to the Concorde solver.
- `ss` refers to a pseudo-solver for the shortest Hamiltonian path problem that returns a path according to an S-shape scheme (i.e., always traverse through an aisle that has been entered). This solver is not implemented in code. Its solutions was found by manually creating paths as needed.
- `greedy` refers to `GreedySolver`.
- `brute-force` refers to `BruteForceSolver`.
- `random1000` refers to `RandomSolver` with $k = 1000$, i.e., a total of 1000 random batches were evaluated.
- `original` refers to a pseudo-solver for the batch optimization problem that always returns the originally created batch for the instances in Table 3.3. This solver was not implemented in code.

4.8.1 Quality benchmarks

The quality benchmarks were implemented in a semi-manual fashion. Using shell scripts, the evaluation program and the Concorde toolkit were used to acquire path lengths of solutions for various combinations of instances and solvers.

For quality benchmarks for the shortest Hamiltonian path problem, the instances listed in Table 3.1 and Table 3.2 were used. The evaluation program was used to run each solver listed in Table 4.1 for each instance, with the exception of `concorde`. For `concorde`, the Concorde toolkit was used. The outputs (i.e., the lengths of the constructed paths) were used as data points. The `concorde` solver is guaranteed to produce optimal solutions, and therefore its solutions were used as baselines when calculating approximation ratios.

Solver
`concorde`
`nn`
`td`

Table 4.1: Solvers used for quality benchmarks for the shortest Hamiltonian path problem. The highlighted row marks the solver whose solutions acted as baselines when calculating approximation ratios.

Note that `HeldKarpSolver` was not included for the quality benchmarks. The reason is that it was deemed infeasible to solve even moderately large instances using it. For example, during experimentation a 22-node instance of the shortest Hamiltonian path problem was solved using `HeldKarpSolver`, which took 9 minutes. Since the execution time grows exponentially for `HeldKarpSolver`, clearly larger instances would not be solved before the deadline of the thesis.

For quality benchmarks for the batch optimization problem, the instances listed in Table 3.3 were used. The evaluation program was used to run each combination of solvers listed in Table 4.2 for each instance. The exceptions are the combinations containing `concorde`, for which the Concorde toolkit was used. The outputs (i.e., the lengths of the constructed paths) were used as data points. It should be noted that no combination of solvers in Table 4.2 can be used as a baseline when calculating approximation ratios, since none of them are guaranteed to produce an optimal solution.

BOP solver	SHPP solver
`greedy`	`nn`
`greedy`	`td`
`greedy`	`concorde`
`random1000`	`nn`
`random1000`	`td`
`random1000`	`concorde`
`original`	`nn`
`original`	`td`
`original`	`concorde`
`original`	`ss`

Table 4.2: Combinations of solvers used for quality benchmarks based on the instances listed in Table 3.3.

Furthermore, the subinstances described in Section 3.3.2.5 were also used for quality benchmarks for the batch optimization problem. The evaluation program and the Concorde toolkit was used to run each combination of solvers listed in Table 4.3 in the same fashion as described above. The combination of the `brute-force` and `concorde` solvers is guaranteed to produce optimal solutions, and therefore its solutions were used as baselines when calculating approximation ratios.

BOP solver	SHPP solver
`brute-force`	`nn`
`brute-force`	`td`
`brute-force`	`concorde`
`greedy`	`nn`
`greedy`	`td`
`greedy`	`concorde`

Table 4.3: Combinations of solvers used for quality benchmarks based on the random subinstances described in Section 3.3.2.5. The highlighted row marks the combination of solvers whose solutions acted as baselines when calculating approximation ratios.

4.8.2 Runtime benchmarks

The runtime benchmarks were written using the `BenchmarkDotNet` framework. It is a framework for conveniently creating robust runtime benchmarks. The framework is well established and used, for example, to benchmark the .NET Runtimes[4] as well as the C# compiler[5]. The framework automatically scales the amount of iterations to run for each benchmark and helps with, for example, running warmups, removing outliers, and removal of unnecessary overhead.

The `BenchmarkDotNet` framework is based around writing methods for which benchmarks will be run. The methods should perform some work, and `BenchmarkDotNet` will measure the amount of time it took to run the method. A method is marked as a benchmark method by adding a special annotation. For the runtime benchmarks created for this thesis, one benchmark method was used for the shortest Hamiltonian path problem, and another benchmark method was used for the batch optimization problem. The benchmarked methods more or less only calls the `Solve` methods of the solvers, resulting in a benchmark that measures the runtime of the `Solve` methods with little overhead.

Furthermore, `BenchmarkDotNet` provides allows specifying a set of parameters for a benchmark method. A developer writing benchmarks provides a list of values for each parameter, and `BenchmarkDotNet` then creates one benchmark for each combination of parameters. For the runtime benchmarks created for this thesis, the parameters were used to specify which instances to run benchmarks for, which solver(s) to use, and (in the case of the batch optimization problem solvers) whether caching should be enabled.

For runtime benchmarks for the shortest Hamiltonian path problem, the instances listed in Table 3.1 and Table 3.2 were used. Each instance was combined with each solver listed in Table 4.4 to create one benchmark.

Solver
`nn`
`td`

Table 4.4: Solvers used for runtime benchmarks for the shortest Hamiltonian path problem.

For runtime benchmarks for the batch optimization problem, the instances listed in Table 3.3 were used. Each instance was combined with each combination of solvers listed in Table 4.5 to create one benchmark.

[4]`https://github.com/dotnet/performance`
[5]`https://github.com/dotnet/roslyn`

BOP solver	SHPP solver	Cache
`greedy`	`nn`	yes
`greedy`	`td`	yes
`random1000`	`nn`	yes
`random1000`	`td`	yes
`greedy`	`nn`	no
`greedy`	`td`	no
`random1000`	`nn`	no
`random1000`	`td`	no

Table 4.5: Combinations of solvers used for runtime benchmarks for the batch optimization problem.

5

Results

This chapter contains the results acquired from running the benchmarks described in Section 4.8. The results are primarily presented as plots for convenient visual comparison, and the underlying data are presented in tabular form in Appendix A.

Section 5.1 contains the results for the quality benchmarks. For all quality benchmarks, plots with approximation ratios are provided. Additionally, the actual path lengths are provided for some problem instances.

Section 5.2 contains the results for the runtime benchmarks. The average execution times are plotted against the size of the problem instances. The runtime benchmarks were executed with compiler optimizations enabled, on a computer with a Intel® Core™ i7-8650U processor clocked at 1.90 GHz, 32 GB of memory, running a 64-bit build of Windows 10.

5.1 Quality benchmarks

This section presents the results from the quality benchmarks for the shortest Hamiltonian path problem solvers and the batch optimization problem solvers respectively.

5.1.1 Shortest Hamiltonian path problem solvers

Figure 5.1 contains the values (i.e., path lengths) acquired for the quality benchmarks for the shortest Hamiltonian path problem, as described in Section 4.8.1 and Table 4.1. As can be seen by the different scales of the path length axes, the values varied significantly in magnitude between the TSPLIB instances and the induced instances, as well as among the TSPLIB instances. Therefore, they were separated into two plots to provide a somewhat clearer comparison among each group of instances.

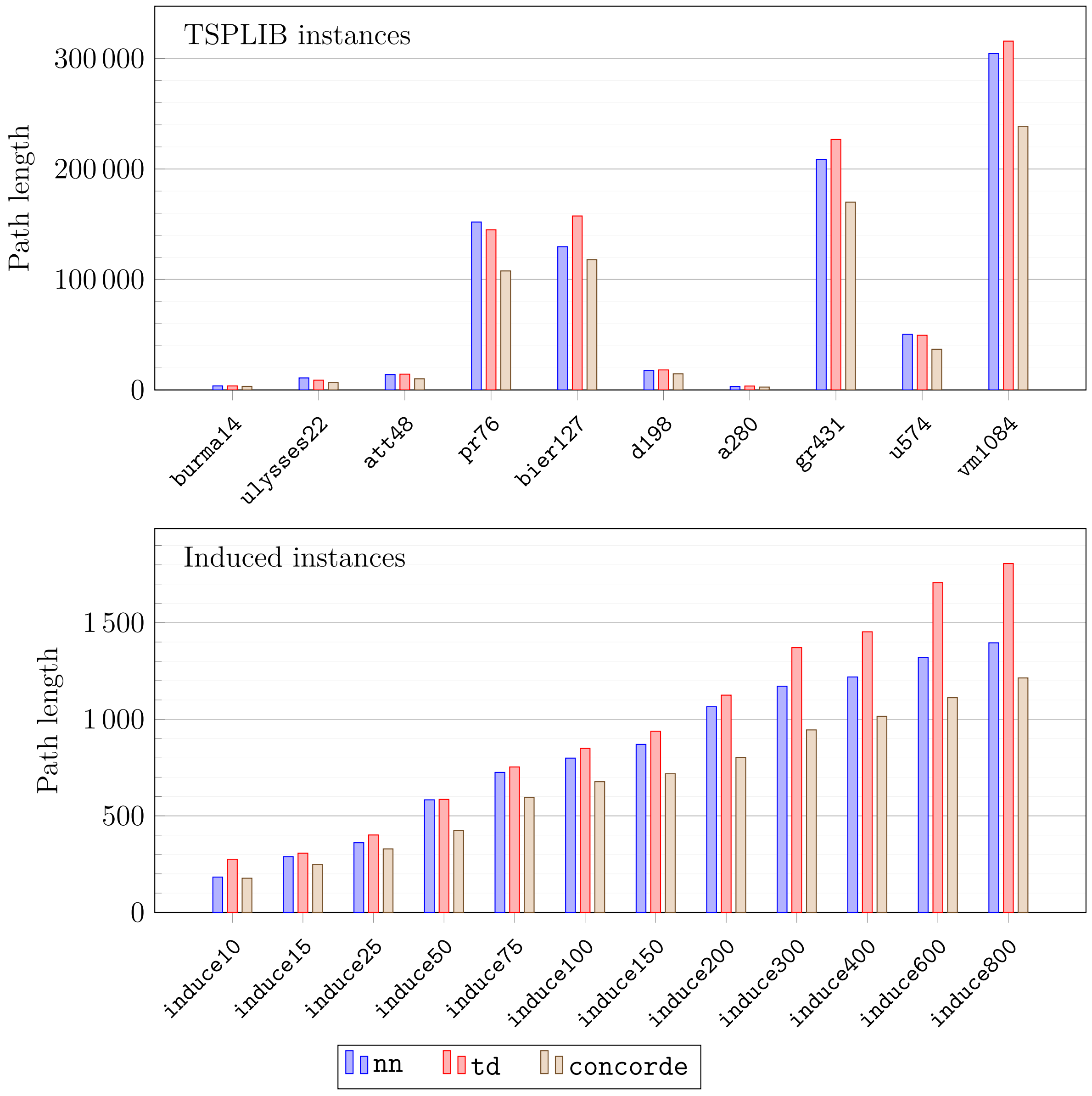


Figure 5.1: Values of the quality benchmarks for the shortest Hamiltonian path problem described in Section 4.8.1 and listed in Table 4.1. Lower is better.

From the values presented in Figure 5.1, approximation ratios were calculated. For each instance, the approximation ratio was calculated by dividing the path length of the `td` and `nn` solvers by the path length of the `concorde` solver. Figure 5.2 contains the calculated approximation ratios.

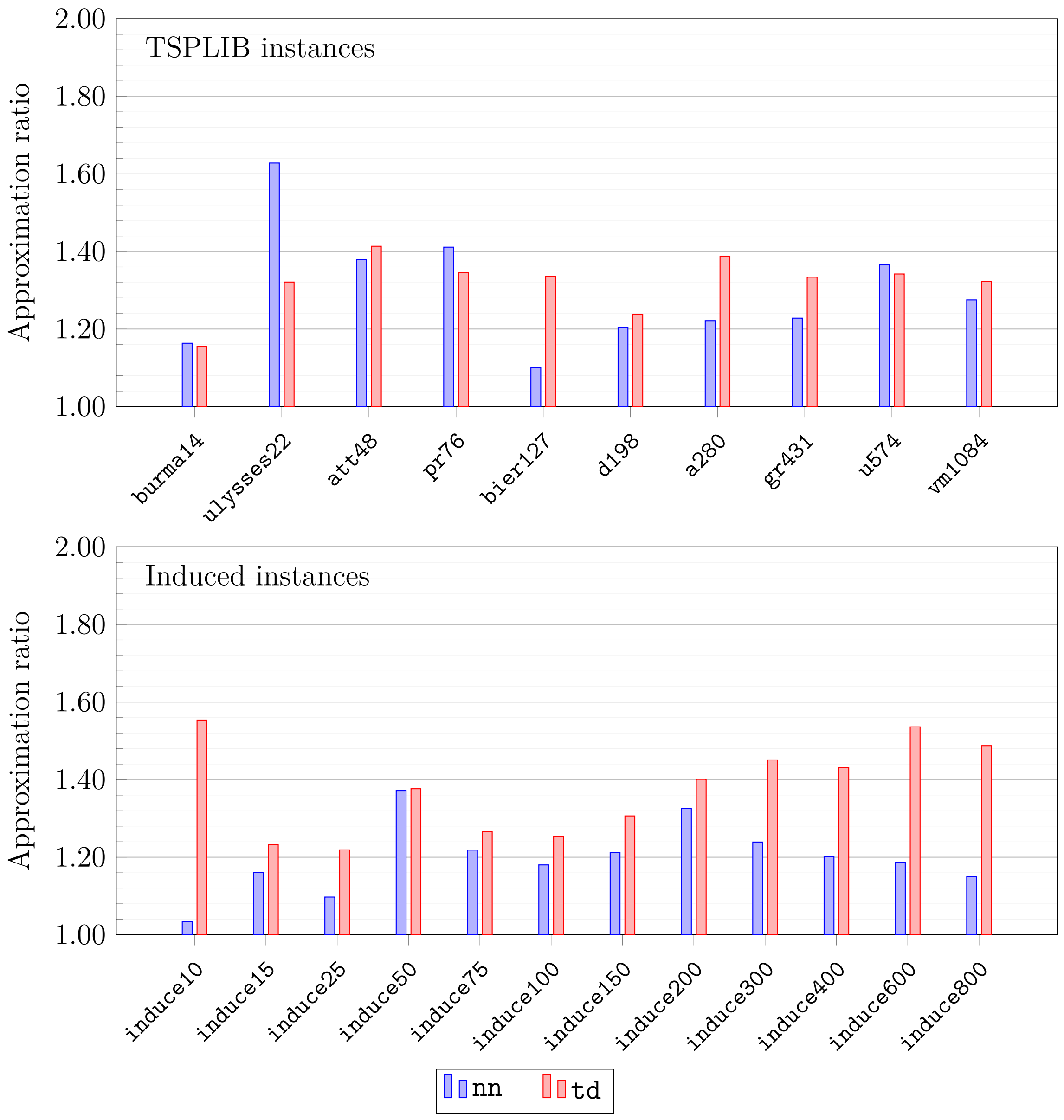


Figure 5.2: Calculated approximation ratios of the benchmarked solvers. The approximation ratios were calculated using the value of the `concorde` solver as baseline. Lower is better.

5.1.2 Batch optimization problem solvers

Figure 5.3 contains the values (i.e., path lengths) of the quality benchmarks defined for the full instances of the batch optimization problem, as described in Section 4.8.1 and listed in Table 4.2. There is one plot per benchmarked batch optimization problem solver, and each plot contains bars for the shortest Hamiltonian path problem solvers it was combined with.

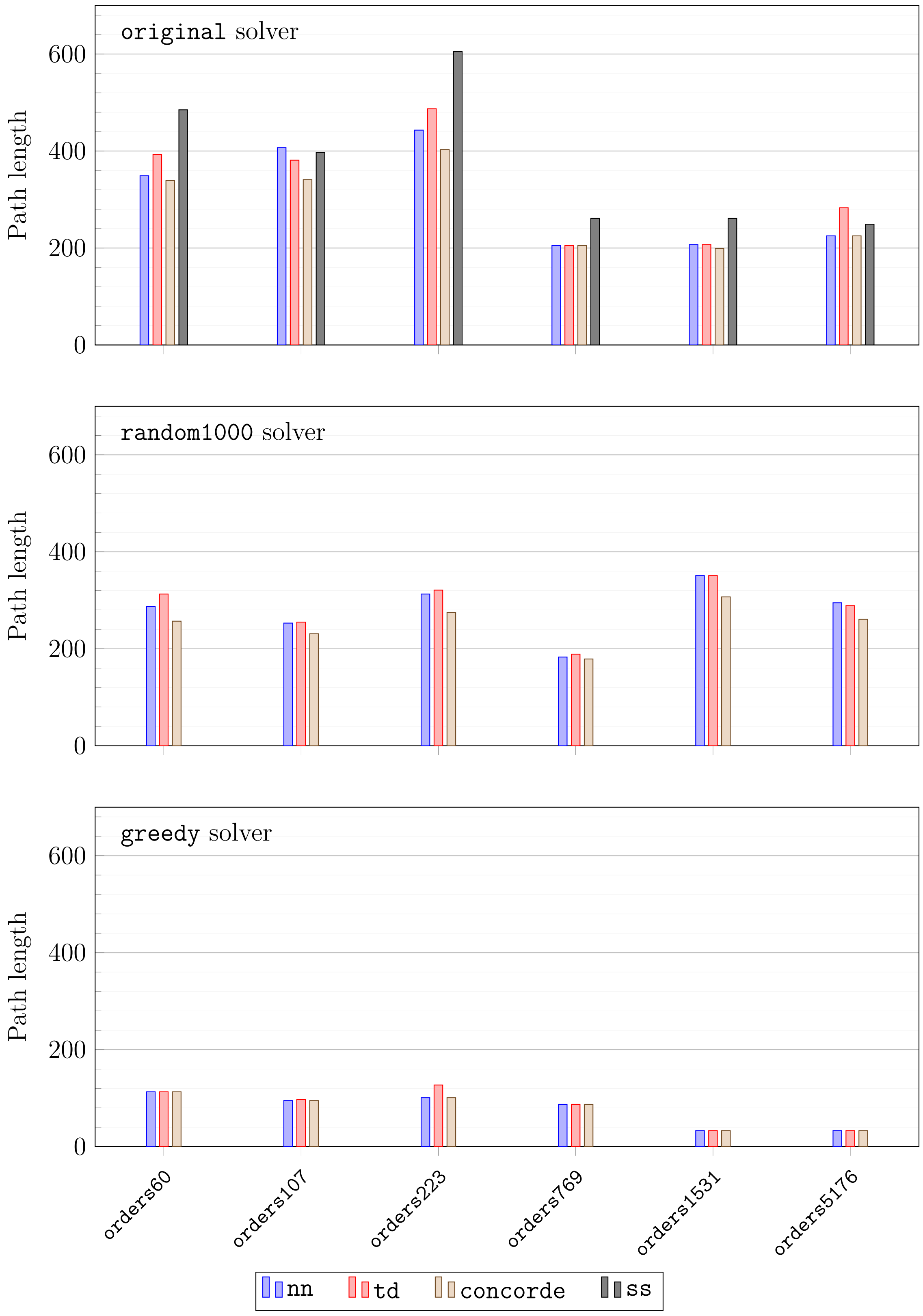


Figure 5.3: Values from the quality benchmarks for the batch optimization problem, based on the full instances, described in Section 4.8.1. Lower is better.

Figure 5.4 contains average approximation ratios calculated from the quality benchmarks defined for the random subinstances, as described in Section 4.8.1 and listed in Table 4.3. The subinstances were grouped by the full instance they were created from. For each subinstance, an optimal solution was found using `brute-force` combined with `concorde`. The approximation ratios were then calculated for the other combinations of solvers by dividing the path lengths of their solutions with the path length of the optimal solution. The calculated approximation ratios were then averaged within each group of subinstances. The standard deviations of the approximation ratios are available in Table A.3.

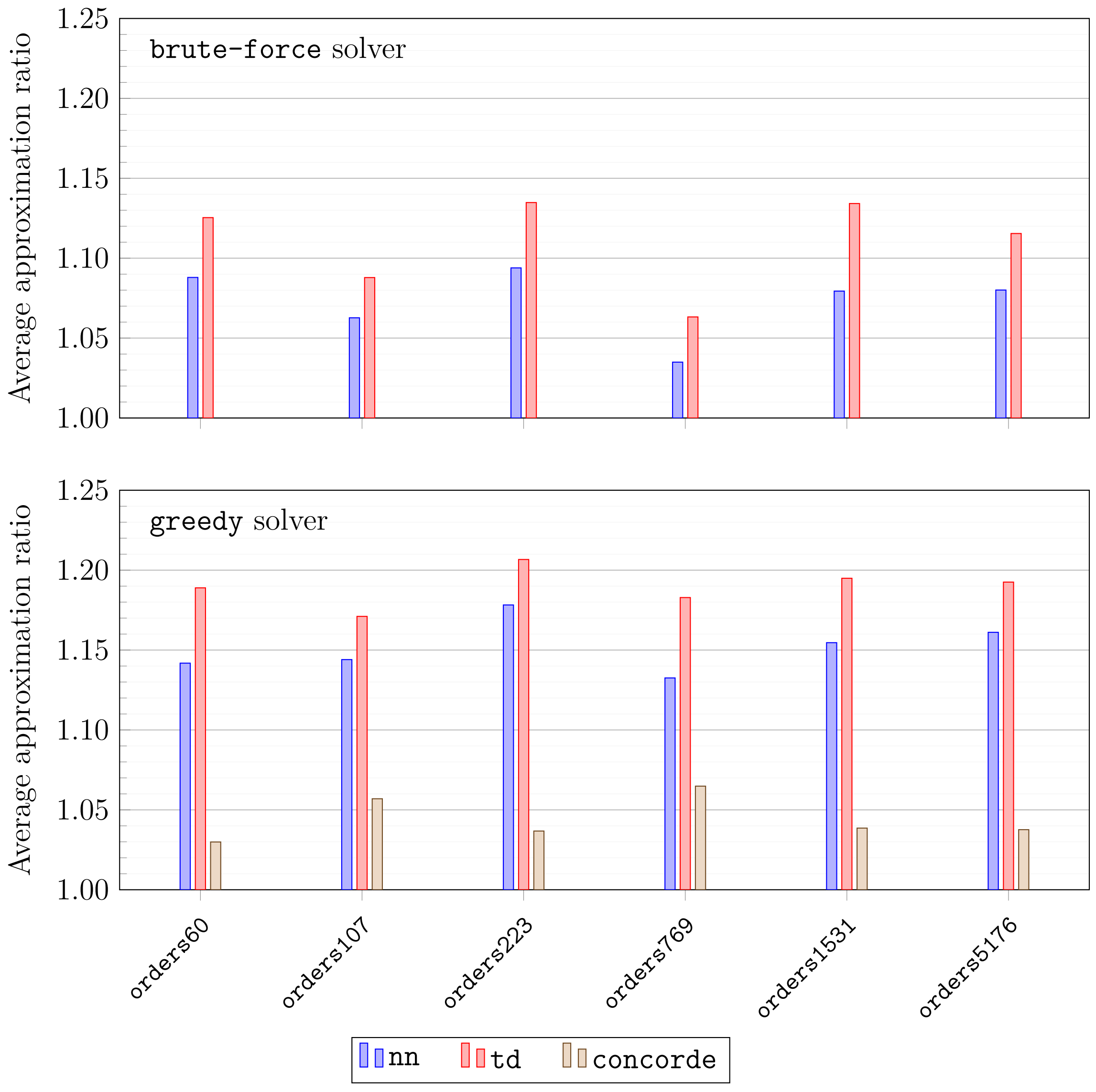


Figure 5.4: Average approximation ratios of combinations of batch optimization problem solver and shortest Hamiltonian path problem solvers, calculated for the random subinstances. The ratios are calculated based on the optimal solution obtained by `brute-force` combined with `concorde`. Lower is better.

5.2 Runtime benchmarks

This section presents the results from the runtime benchmarks for the shortest Hamiltonian path problem solvers and the batch optimization problem solvers respectively. The results for the shortest Hamiltonian path problem solvers are presented together as to allow for easy comparison. The results for the batch optimization problem solvers are presented separately for each of the solvers together with their specific available configurations.

5.2.1 Shortest Hamiltonian path problem solvers

Figure 5.5 contains the values of the runtime benchmarks (i.e., average runtime as measured by `BenchmarkDotNet`) described in Section 4.8.2 and listed in Table 4.4. The runtimes are plotted against the size (i.e., number of nodes) of the instances. Note that no distinction is made between the TSPLIB instances and the induced instances.

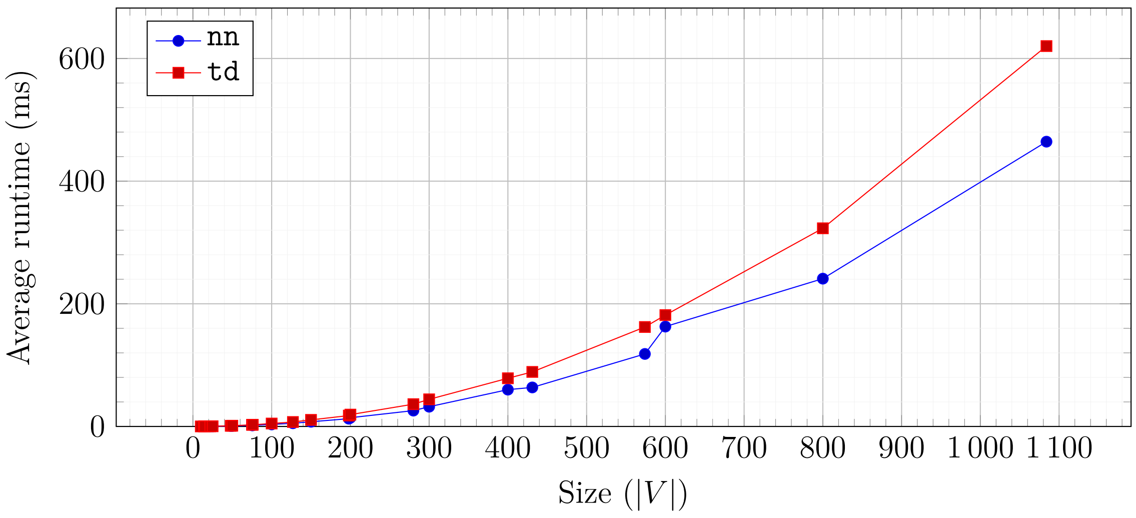


Figure 5.5: Average runtime for invocations of the `Solve` method of the `nn` and `td` solvers. Lower is better.

5.2.2 Batch optimization problem solvers

Figure 5.6 contains the values of the runtime benchmarks described in Section 4.8.2 and listed in Table 4.5. The runtimes are plotted against the size (i.e., number of available orders) of the instances. The figure contains one plot each for the `greedy` and `random1000` solvers, solely for the purpose of reducing the amount of visual clutter.

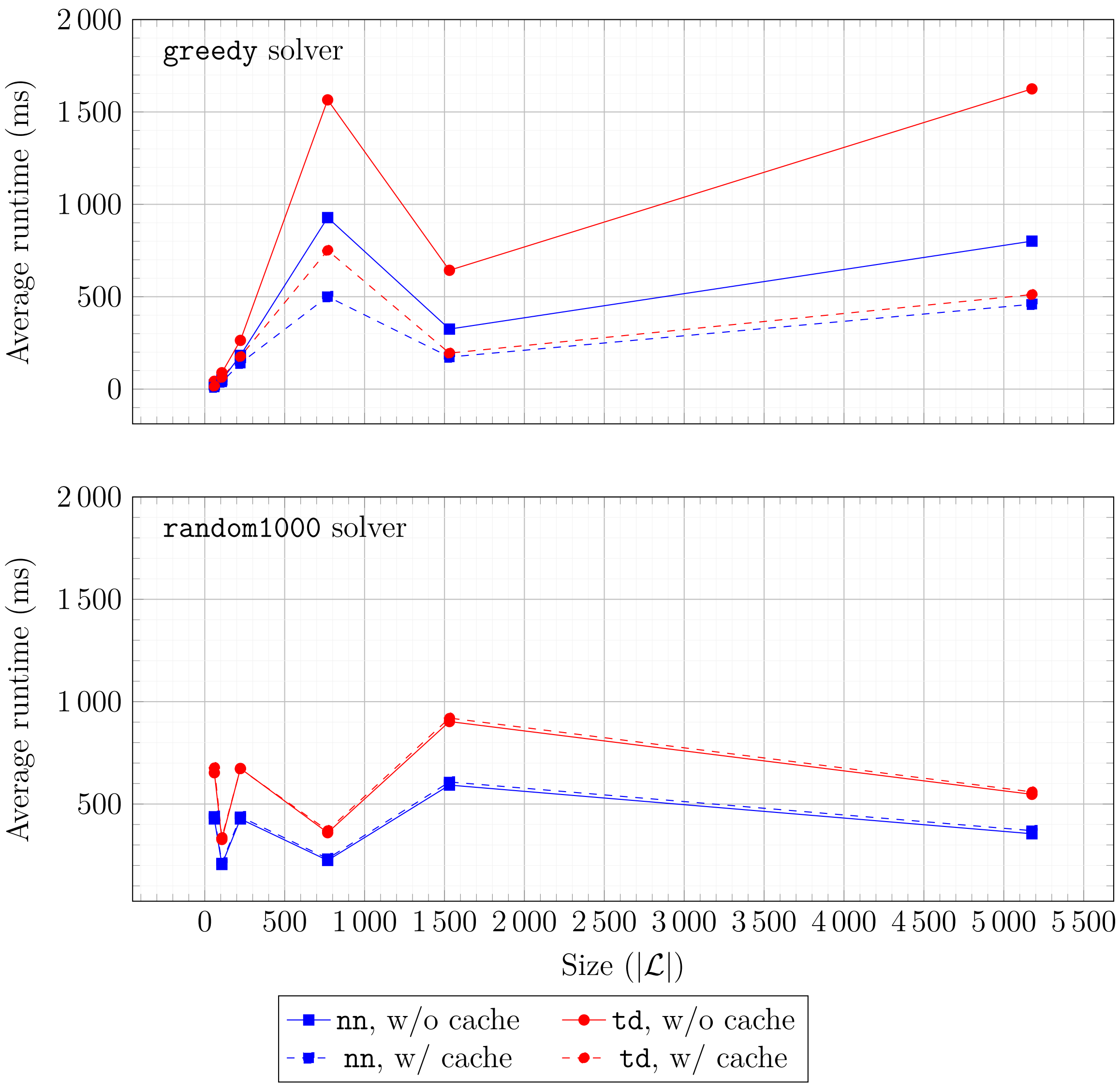


Figure 5.6: Average runtime for invocations of the `Solve` method for different combinations of batch optimization problem solvers, shortest Hamiltonian path problem solvers, and enabled/disabled cache. Lower is better.

6
Discussion

This chapter analyzes the results from Chapter 5. Interesting parts of the results are highlighted, such as which solvers appears to perform well (or not so well), and in which situations. Possible explanations for irregular or unexpected results are also discussed. In addition to analyzing the results, limitations of the abstract models and the benchmarks are discussed.

Section 6.1 and Section 6.2 analyze the results of benchmarks for the shortest Hamiltonian path problem and the batch optimization problem, respectively. Section 6.3 discuss the effect the choice of start and end points has on the result. Finally, Section 6.4 discusses limitations of the warehouse model, the order model, and the benchmarks.

6.1 Results for shortest Hamiltonian path problem

In this section the results for the shortest Hamiltonian path problem benchmarks presented in the previous chapter are discussed. This is done in regards to the approximation ratios for the quality benchmarks and time complexities for the runtime benchmarks.

6.1.1 Quality benchmarks

As the worst-case approximation ratio of `td` is bounded, it is of interest to compare the measured approximation ratios with this bound. As stated in Section 3.2.1.4, the worst-case approximation ratio is 2. From Figure 5.2 it is apparent that `td` generally achieves significantly better approximation ratios than 2. For the TSPLIB instances, the achieved approximation ratio is less than approximately 1.4. For the induced instances, the achieved approximation ratio is slightly worse: less than approximately 1.6.

Even so, it is apparent that `nn` performs better than `td` in almost all cases. There are a few exceptions, though. For the `ulysses22` instance, `nn` performs significantly worse; for `burma14`, `pr76` and `u574`, `nn` performs slightly worse. In all other cases, `nn` performs slightly or significantly better.

It is particularly interesting that `nn` generally performs better than `td`, seeing as the worst-case approximation ratio of `nn` ($\mathcal{O}(\log |V|)$) depends on the number of nodes, which the worst-case approximation ratio of `td` does not. Perhaps the instances where nn produces the worst results are unlikely to appear in practice (though this is just speculation).

6.1.2 Runtime benchmarks

As can be seen in Figure 5.5, both td and nn exhibit a quadratic growth in execution time as the size of the instance grows. The quadratic growth is a confirmation of the time complexities of the respective solvers, which are found to be $\mathcal{O}(|V|^2)$ for both solvers as described in Section 3.2.1.3 (for nn) and Section 3.2.1.4 (for td).

It continues to be the case that nn performs better than td, since nn achieves a lower average execution time in all runtime benchmarks. However, this is not necessarily an indication that the nn solver will always be faster than td. The implementation of nn is very simple, and it is easier to use optimal data structures that perform well. The implementation of td, on the other hand, is more complicated. When implementing td, the main focus was to achieve correctness, and less effort was spent on achieving efficiency. It is entirely possible td could be significantly more efficiently implemented than what has been done for this thesis.

Furthermore, the implementations of both nn and td exclusively use the standard library accompanying the C# language. The standard library is most likely of high quality, offering efficient implementations of many data structures. However, it is possible that there exist third-party implementations of equivalent data structures that are more heavily optimized. In that case, it might be possible to affect the runtime performance of both nn and td by utilizing these third-party implementations.

Using third-party code comes with an increase in complexity of the software, though. A maintainer of the nn and td implementations would need to keep the third-party code updated when new patches are released by its maintainers. There is also the risk that the maintainers of the third-party code stops supporting it, meaning that bug fixes and updates will not be released, which increases the maintenance burden on the maintainer of the nn and td implementations.

6.2 Results for batch optimization problem

This section contains discussion of the results for the batch optimization problem benchmarks presented in the previous chapter. The quality benchmarks section contain separate discussions for the full instance benchmarks and the random subinstances as the full instances are compared to the originally created batches, while the random subinstances are compared to the optimal solutions created with the brute force algorithm together with the Concorde solver. This is followed by discussion about the runtime benchmarks, with special focus on the inconsistency of the results and the effect of the cache mechanism.

6.2.1 Quality benchmarks

It is directly apparent from Figure 5.3 that both random1000 and greedy generally yield better solutions than the originally created batches. random1000 yields a better solution in four out of six cases, although the difference is not very large. `greedy`, on the other hand, yields significantly better solutions in all cases. For the `orders1531` and `orders5176` instances, `greedy` yields a solution that is 7-8 times shorter than the originally created batch.

Looking at the solutions created by `greedy`, it can also be seen that using either `nn` or `td` yields in all cases but one a solution that is equivalent to using `concorde`. This is a promising result, since it indicates that, in a real-world warehouse context, an approximation algorithm for the shortest Hamiltonian path problem can in many cases achieve as good results as an optimal algorithm. The sample size is a bit small, though; more benchmarks are required before such a conclusion can be drawn.

Turning the attention to the random subinstances, Figure 5.4 shows that `greedy` continues to yield good results. Using `greedy` together with `nn` or `td` yields solutions that are on average within approximately 12 % to 20 % of an optimal solution. It can also be seen that `nn` continues to perform slightly better than `td` in these benchmarks. Using `concorde` instead yields solutions that are on average within approximately 5 % of an optimal solution. If `brute-force` is used instead of `greedy`, then using `nn` or `td` yields solutions that are with approximately 5 % to 15 % of an optimal solution. `nn` performs better than `td` here as well.

Assuming that the approximation ratios for the full instances are approximately the same as the approximation ratios for the random subinstances, these results are encouraging. If a batch found by an approximation algorithm is 20 % worse than a batch found by an optimal algorithm, but is several times better than the originally created batch, then it will still be a significant improvement. Perhaps such a batch is “good enough”, meaning that the extra effort required to find an optimal batch is not worth it.

6.2.2 Runtime benchmarks

This section begins in Section 6.2.2.1 with a general analysis of the results of the runtime benchmarks for the batch optimization problem. It is followed by Section 6.2.2.2 which gives a more in-depth analysis on the effects of using a cache when running a batch optimization problem solver.

6.2.2.1 General analysis

While the runtime benchmarks for the shortest Hamiltonian path problem solvers yielded results that were in line with the predictions, the runtime benchmarks for the batch optimization problem solvers yield inconsistent results. It is expected from the time complexities of the relative algorithms that the execution time will grow as the size of the instance grows. However, the results of the runtime benchmarks do not show this behavior.

Examples of this inconsistency can be found in Figure 5.6. With `greedy`, and regardless of whether `nn` or `td` is used, the `orders769` instance has a significantly higher average execution time compared to the `orders1531` instance (which is twice as large), and a similar average execution time compared to the `orders5176` instance (which is several times larger). With `random1000`, an opposite relation exists between `orders769` and `orders1531`.

With such a small sample size it is difficult to draw any conclusions. In hindsight, more instances should have been used for the runtime benchmarks, with a more even size distribution. For example, subinstances of varying size could have been created from `orders5176` by randomly selecting subsets of the set of available orders. These subinstances could then have been used for, hopefully, more accurately showing the relation between instance size and execution time.

What is consistent, though, is that `nn` again performs better than `td`. Using `greedy` and not using a cache, solving the `orders769` instance using `nn` is approximately 50 % faster than using `td`; for `orders5176`, `nn` is approximately twice as fast as `td`. For `random1000`, the difference is less significant but still apparent.

6.2.2.2 Effect of using a cache

It can be seen that using a cache is quite effective for `greedy`. Using `nn` together with a cache is almost twice as fast compared to not using a cache. With `td`, the speedup is even more significant, where using a cache is approximately 2-3 times faster compared to not using a cache. What is interesting is that using a cache results in a markedly less significant difference between using `nn` and `td`. With a cache, using `nn` is only slightly faster than using `td`. This is an important result, as it means that the choice of whether to use `nn` or `td` can be made with much less concern for runtime.

On the other hand, using a cache does not improve the average runtime of `random1000`. In fact, using a cache actually *increases* the average runtime. This is an indication that the overhead of maintaining the cache outweighs the potential speedups; most likely, the number of cache hits is very low compared to the number of cache misses.

A possible explanation for the apparent inefficiency of the cache for `random1000` is that it is very unlikely that randomly selecting batches yield equal sets of nodes. Since the keys to the cache are instances of the shortest Hamiltonian path problem on induced subgraphs, two batches would have to result in the exact same of nodes to induce by, which is highly unlikely. It might be the case that a batch has a majority of its nodes in common with many other batches, but since the sets of nodes are not exactly equal, the cache cannot be used.

A related argument can be made to possibly explain the efficiency of using a cache for `greedy`. `greedy` constructs batches one order at a time, starting from an empty batch. When selecting the first order to include in the batch, it is likely that two or more orders will contain the exact same set of nodes. For example, a small subset of items that are more popular than other may frequently occur in orders with only a few items. This will result in many cache hits when selecting the first order. When selecting the second or later orders, it is likely that many one of the available orders contain only nodes that are already covered by the orders in the batch, resulting in many cache hits.

One thing not demonstrated by the runtime benchmarks is that a single cache can be reused among many instances of the batch optimization problem. For example, suppose that a batch of 16 orders has been selected for the `orders5176` instance, and that a cache was used. Now, a second batch should be created from the remaining 5160 orders. Invoking a batch optimization problem solver with the same cache would most likely result in very many cache hits, decreasing the runtime significantly. The biggest disadvantage of using a cache for many subsequent batch optimization problem instances is that the cache in its current design grows endlessly. Eventually, the memory required by the cache may exceed the amount of available memory, leading to a crash. If the cache should be reused, either a policy should be deployed for evicting entries from the cache, or it should be verified that the available memory is large enough that the cache cannot feasibly occupy it.

6.3 Importance of start and end points

The solutions returned by `greedy` for `orders1531` and `orders5176` can be verified to be optimal using a special argument. In the concrete model of the warehouse for which `orders1531` and `orders5176` is defined, the path length of the solutions for the aforementioned instances is 33 (see Table A.2). The shortest path between the start and end nodes is 31, as can be seen in Figure 3.4. There are no nodes on the shortest path between the start and end node that can be included in any order, since the path contains no locations where an item can be collected. However, adjacent to the path there are nodes where items can be collected. Being adjacent to the path, the distance from some node in the path to some node where items can be collected is therefore 1. As such, an optimal solution must have a path length of at least 33.

This indicates that the position of the start and end nodes can significantly affect which orders are included in a solution. For example, with the start and end nodes positioned as shown in Figure 3.4, orders whose items are all located towards the left ends of the upper-left shelves are "prioritized". Had the end node instead been placed in, e.g., the bottom-right corner of the warehouse, an entirely different type of orders would have been "prioritized", and the solutions could have been significantly different.

6.4 Limitations

This section discusses in Section 6.4.1 characteristics of the warehouse model and the order model that limit their usefulness in a real-world warehouse scenario. Then follows in Section 6.4.2 a discussion of the limitations of the benchmarks that make it more difficult to draw decisive conclusions from the results.

6.4.1 Limitations of the model

Using an undirected grid graph is a simple and flexible way to model a warehouse. It imposes no restrictions on the layout on the warehouse, making it suitable for a variety of warehouse types. However, being as simple as it is, the grid graph model makes it difficult to model constraints that may be relevant for a real-world warehouse. An example is the direction that a picker may be moving in. The grid graph model assumes, by virtue of being undirected, that a picker can move in any direction. This is not always true in practice: picking may be conducted with a trolley or some other vehicle that makes it impossible to turn around once an aisle has been entered, forcing the picker to traverse the entire aisle.

The model used for orders—i.e., modelling an order as a subset of nodes—is also very simple. Similarly to the grid graph model, this simplicity makes it difficult to model constraints relevant for a real-world warehouse. Below are some examples of such constraints.

- **Zero cost of picking items.** It is assumed that picking any number of items at a single location has a zero cost (i.e., takes a zero amount of time). This is obviously not the case in a real-world warehouse: picking a single small and/or light item (such as a kitchen knife) is clearly less costly than picking several large and/or heavy items (such as ten large cooking pots).
- **Space and weight constraints.** The order model carries no information about the space occupied by the items, or what their weight are. For this thesis it was assumed that a picker always has the space and weight capacity to pick all items, as long as those items belonged to at most N different orders. In a real-world warehouse, though, a trolley may not have the capacity to pick N different orders, all consisting of many large and heavy items. In that case, less than N orders can be placed on the trolley.
- **Precedence constraints.** Some items may be more fragile than others, and should therefore not be picked so that they risk being damaged by other items. A strategy for picking fragile items could be to pick them last, putting a precedence constraint between the picked items. No such constraint is modeled by the order model: all items are treated equally and can be picked in any order.
- **Tardiness constraints.** It may be the case that some orders should be picked earlier than other orders. For example, orders where the customer has paid extra for express delivery, or orders that, if not picked, will not be delivered within the deadlines the retailer guarantees their customers. In the order model, the set of available orders is not ordered in any way, and the included orders are all considered equal.

Some of these constraints can be taken into account by modifying the set of available orders. For example, for the space, weight, and precedence constraints, the set of available orders can be limited so that each order contains only items that are small (space), light (weight), and non-fragile (precedence). As another example, the set of available orders can be filtered to orders that have a priority status (e.g., when a customer has paid extra for express delivery), or orders that are at least d days old (e.g., orders that must be delivered soon to not violate delivery guarantees to the customer).

Another simplification made for this thesis is that the warehouse is static, in the sense that the only obstacles are the shelves and the walls. This is an implicit assumption that a picker is alone in the warehouse when picking a batch of orders. Having multiple pickers means that some paths may not be possible to take, due to another picker blocking the way. For example, an aisle may be so narrow that two pickers cannot pass each other. A real-world warehouse, where multiple simultaneously working pickers is common, is therefore more dynamic than can be captured by the current model.

Finally, by using the lengths of shortest paths between locations in the warehouse as the weights in the complete graph (as described Section 3.1.3), an implicit assumption is made that a human picker is "smart enough" to always choose such a shortest path. While not unreasonable for an experienced picker that knows the warehouse well, it is still a very strong assumption. If an optimal batch is constructed on the premise that the shortest path will always be used, but the picker picking the batch chooses a non-optimal path, there might exist a batch that is better suited to the picker's non-optimal path.

6.4.2 Limitations of the benchmarks

As touched upon in Section 6.2, the sample size is a bit small for the benchmarks for the batch optimization problem. This makes it more difficult to draw conclusions from the results.

For example, only data from a single warehouse was used for the quality benchmarks. It would be interesting to use data from other warehouses to create a more diverse set of benchmarks. Other warehouses could be different in a number of ways, for example:

- different layout;
- different placement of items (e.g., grouping items by popularity, by category, or similar);
- or different distribution of items among orders (e.g., some items that are highly popular and make up a majority of the orders, or a more even distribution of all items).

Using more warehouses could have surfaced more subtle characteristics of the solvers. For example, although `greedy` appears to work well for the quality benchmarks that were run, it might not necessarily work well in other benchmarks. Finding benchmarks were `greedy` works less well could provide more insight into the type of warehouses where it works better or worse.

The small sample size stems from the time-consuming and manual-labor-intensive process of creating the instances from snapshots, as described in Section 3.3.2.2, as well as general time limitations. If more time was available, more instances would have been created and used for quality benchmarks.

Furthermore, if more time was available, `random1000` would have been used for the quality benchmarks based on the random subinstances. Creating the quality benchmarks for the random subinstances was a time-consuming process. For the quality benchmarks to be fair, the set of random batches should be the same independent of which shortest Hamiltonian path problem solver was used. This would have been simple if only the evaluation program or only the Concorde toolkit was used, since both the C# standard library and the Go standard library provide ways to set a specific seed for the random number generator. However, using a specific seed for one random number generator provides no guarantees about using the same seed for a different random number generator. Therefore, the set of randomly generated batches was created once and stored to files. Using these files for the quality benchmarks turned out to be somewhat cumbersome, which made running the benchmarks time-consuming.

Finally, there is one important piece of functionality that is missing from the quality benchmarks, and that is the ability to visualize the solutions. The functionality is missing since, with the current implementation, it is not possible to map from an integer node identifier (belonging to the complete graph) to a row-column pair node identifiers (belonging to the grid graph). As such, it is not possible to visualize which nodes were included in a batch, or which path through the warehouse was used. Having such a visualization may grant additional intuition for and insight into how the various solvers perform for certain types of instances.

7
Conclusions

This chapter is devoted to summarizing the thesis. Section 7.1 considers the results from Chapter 5 and the discussions from Chapter 6, and makes a recommendation as to which solvers should be used for a real-world warehouse deployment. Section 7.2 attempts to answer the research questions posed in Section 1.3. Finally, Section 7.3 points to future work that may be carried out using this thesis as a basis.

7.1 Recommendation of solvers

As stated repeatedly in the discussion, `nn` performs better than `td` in almost all cases. For the quality benchmarks, `nn` achieves lower approximation ratios, with exception for a few instances where `td` achieves slightly lower approximation ratios. For the runtime benchmarks, `nn` achieves a shorter average execution time. An additional advantage (in the authors' opinion), from the perspective of a software engineer, is that `nn` is easier to understand and implement than `td`, both in terms of correctness and efficiency. As such, the implementation of `nn` is easier than `td` to maintain over a long timespan.

As also stated repeatedly in the discussion, `greedy` appears to perform well. The only situation where `random1000` performs better is in the runtime benchmarks. However, the runtime advantage of `random1000` is negligible as both solvers are "fast enough", according to Ongoing Warehouse. As such, it appears that `greedy` is the superior solver.

Furthermore, using a cache in the form of `CachingSolver` shortens the average execution times significantly when using `greedy`. The final recommendation to Ongoing Warehouse is then to use `greedy` together with a `CachingSolver` wrapping `nn`. However, it might be desirable to limit the cache's growth so as to not exhaust the available memory.

7.2 Answers to research questions

Based on the results of this thesis, this section attempts to give concrete answers to the research questions posed in Section 1.3.

1. **Can abstract models and algorithms from the existing literature on warehouse optimization, as well as combinatorial optimization, be modified, adapted, or combined to solve the optimization problems as defined in Section 1.2?**

The idea to use a graph model for representing warehouses is fairly natural, and is commonly occurring in the existing literature in various forms. The grid graph model is (as far as the authors are aware) novel, at least within the literature that was researched prior to this thesis. The algorithms for the shortest Hamiltonian path problem were adapted from algorithms for the travelling salesperson problem that were found in existing literature. On other hand, the algorithms for the batch optimization problem are (again, as far as the authors are aware) novel. The exception is the relative brute force algorithm, as brute force algorithms are generally trivial and closely related to the actual definition of the problem. Taking the above into account, the answer to this question appears to be “yes”.

2. **How well do such algorithms perform when applied to real-world warehouse data, when considering measurements such as running time, memory requirements, and approximation ratios? How does this compare to theoretical results such as average-case and worst-case scenarios?**

   Generally, the algorithms and their implementations perform “well enough”. For the shortest Hamiltonian path problem, the approximation algorithms achieved good approximation ratios, considering the worst-case bound for them. For the batch optimization problem, the approximation algorithms performed well when compared to the originally created batches. Additionally, forr two of the instances—`orders1531` and `orders5176`—the solutions produced by `greedy` could be verified to be optimal. In terms of runtime, the longest average execution time for a batch optimization problem instances was less than 2 seconds, which is well within what Ongoing Warehouse considers acceptable. As such, the answer to this question is “well enough”.

3. **Are such models and algorithms suitable to solve the batch optimization problem in a real-world warehouse? If not, what assumptions and delimitations need to be lifted for the models and algorithms to become suitable?**

   As discussed in Section 6.4.1, the models of the warehouse and the orders make assumptions that are somewhat unrealistic for a real-world warehouse scenario. Some of the assumptions can be adapted, such as filtering the set of available orders to take into consideration constraints such as precedence or tardiness constraints. Other assumptions are more difficult to adapt, such as the assumption that a picker is alone in the warehouse, or that the picker will always choose the shortest path between two locations. Without evaluating the model and the algorithms by using them for actual warehouse operations, it is difficult to tell which of the assumptions are most important to lift or adapt; it might also differ from warehouse to warehouse. The answer to the first part of this question is then “probably not”. The answer to the second part of this questions is “it depends”.

7.3 Future work

This section discusses three directions for which future work based on this thesis can be carried out. Section 7.3.1 discusses what is needed for deploying the implemented models and algorithms in real-world warehouse operations. Section 7.3.2 discusses some aspects through which the current set of benchmarks can be improved and extended. Finally, Section 7.3.3 discusses the need for an optimal algorithm for the batch optimization problem that is faster than the relative brute force algorithm.

7.3.1 Real-world warehouse deployment

This thesis has focused on doing evaluation using historical data. It would therefore be interesting to compare the results from historical data with results from live warehouse operations.

For example, the combination of `greedy`, `nn` and `CachingSolver` could be integrated into the warehouse management system and used by one of Ongoing Warehouse's customers for, say, a week. The results could then be measured with metrics such as number of orders picked or time to pick a batch. However, such an experiment is a “high risk, high reward” investment. It is possible that the warehouse increases its efficiency during the course of the experiment, but it is also possible that the opposite occurs, in which case experiment could quickly become costly.

Regardless, for such an experiment to be feasible, the implementations of the models and algorithms need to be improved. Both shortest Hamiltonian path problem solvers and batch optimization problem solvers need to be updated to return the constructed path and batch, respectively. This is not a change that is difficult to make or takes a long time to do, but it needs to be made nonetheless. As with the visualizations described in Section 6.4.2, a missing piece of functionality is being able to convert integer node identifiers back to the actual locations in the warehouse. Furthermore, the orders included in a batch need to be correlated with the order identifiers as they are stored in the database of the warehouse management system. This change is also not difficult to make, but will require a significant amount of time and effort.

7.3.2 More extensive benchmarking

A fairly straightforward to carry out future work based on this thesis is to perform more extensive benchmarks. As discussed in Section 6.4.2, there are a number of ways in which the existing benchmarks can be improved upon. For example, the authors identified the following, major aspects through which the existing benchmarks could be extended:

1. **Increasing the sample sizes.** For the shortest Hamiltonian path problem, there are many more instances in TSPLIB that can be used for both quality and runtime benchmarks of `nn` and `td`. For the batch optimization problem, subinstances of varying sizes can be created from a large instance, as described in Section 6.2.2.1. Furthermore, more instances can be created from snapshots,

both from the warehouse used for the existing benchmarks and from other warehouses.

2. **Vary other parameters.** The most obvious parameter to vary is the batch size for the batch optimization problem, as it varied very little in the instances used for the existing benchmarks. For example, the time complexity of `greedy` depends on the batch size, and it would be interesting to see how the average execution time varies as the batch size is varied. However, it is unlikely that a real-world warehouse would have a trolley that can carry large numbers of orders, so benchmarks with high values for batch size would then be interesting primarily from a scientific perspective.

   For the shortest Hamiltonian path problem, the most significant parameter is the size of the instances (i.e., the number of nodes), but there are also other, more indirect parameters.

   For example, the "shape" of the input graph can be varied. Assume that the nodes are points in a plane, and that Euclidean distances are used as weights. How well do the algorithms perform when the points are evenly distributed within some fixed area? How well do they perform when the points are grouped into a few clusters?

   Another indirect parameter is the magnitude of the weights. Is there any difference in the performance—both quality and runtime—of the algorithms when applied to a graph whose weights lie in the range $[0, 100]$ compared to a similar graph where the weights lie in the range $[10\ 000, 20\ 000]$?

7.3.3 Faster optimal algorithm

A major improvment over the results of this thesis would be to find an optimal algorithm for the batch optimization problem that is faster than relative brute force algorithm combined with an optimal shortest Hamiltonian path problem algorithm. The brute force algorithm has a time complexity of at least $\mathcal{O}(|\mathcal{L}|!)$, so even an optimal algorithm that has an exponential time complexity would be a significant improvement. Such an algorithm could provide optimal solutions for larger instances than the brute force algorithm in "semi-feasible" time: short enough time to be used as the baseline for quality benchmarks (i.e., run only once), but not short enough to be used in a real-world warehouse scenario.

7.4 Ethical considerations

Warehousing is a common business throughout the world due to their importance in logistic flows. Due to the often repetitive and straightforward tasks of a picker, it is a job available to many people as it does not require any higher education. As such it serves as an important opportunity for almost anyone just entering the labor market.

As most business fields warehouseing has a focus on efficiency and profitability. However big leaps in efficiency may reduce the amount of work necessary and as a direct effect reduce the amount of available jobs. This could potentially affect a lot of people as warehouses exists everywhere.

Another aspect of an algorithm centered solution to a problem like this is that it might be a better fit for robots than humans. This could lead to humans successively being replaced also yielding fewer work opportunities. Another issue with algorithms could be that yielded paths might seem unreasonable, which in turn could lead to the directives not being followed at all. Essentially rendering the attempted optimization useless.

In order to avoid such a situation it is important to thoroughly test solutions and continously interact with the warehouse personnel intended to use the system to get valuable input on the human perspective.

It should be stated explicitly that the above arguments are based purely on speculation, and not backed by research.

8 Related work

The field of warehouse optimization has been covered through many different perspectives throughout the years and as such it has yielded a lot of different focus areas. Some classifications of different areas with focus on warehouse operations can be found in a review paper by Gu, Goetschalckx, and McGinnis [13] which gives insight into the complexity of the problem areas encountered.

Koster, Le-Duc, and Roodbergen [19] gives a thorough introduction to the different methods used in order picking. They also cover related areas like warehouse layout design and storage assignment, giving a broad perspective of the field and its respective theories and problems.

The joint picker routing and batching problem appears in a paper by Won and Olafsson [30]. The problem combines choosing the picker's route through the warehouse, with choosing which orders to pick. This is similar to the combination of the shortest Hamiltonian path problem and the batch optimization problem considered for this thesis. Won and Olafsson consider the combined problem with the addition of a focus on customer demand responsiveness. They present two heuristic algorithms as potential effective solutions to the problem.

Another take on the joint picker routing and batching problem is presented by Matusiak et al. [23] who additionally take precedence constraints into consideration. This results in a solution containing two subalgorithms: an optimal A*-algorithm for routing, and a simulated annealing algorithm for batching.

Scholz and Wäscher [28] take a step away from commonly assumed picker routing schemes (e.g., the S-shape scheme) and attempt to solve the problem by integrating different routing algorithms into an iterated local search algorithm used for creating batches.

In recent years there have also been attempts at using heuristic algorithm techniques to solve the combined problem. Chen et al. [4] attempt to solve the combined order batching, sequencing, and routing problem by using a combination of a genetic algorithm together with an ant-colony algorithm for solving the travelling salesperson problem, reaching the conclusion that runtime might become an issue in practice.

Kulak, Sahin, and Taner [21] also present an interesting take on the joint picker routing and batching problem. They present a heuristic solution based on a combination of a cluster-based tabu search algorithm together with the nearest neighbour and 2-opt algorithms.

Lin et al. [22] attempt to solve the joint picker routing and batching problem with the aid of particle swarm optimization, giving each order a virtual order center that is used to combine similar orders into batches.

Bibliography

[1] Richard Bellman. “Dynamic Programming Treatment of the Travelling Salesman Problem”. In: *Journal of the ACM (JACM)* 9.1 (Jan. 1962), pp. 61–63. ISSN: 0004-5411. DOI: `10.1145/321105.321111`.

[2] Mandell Bellmore and George L. Nemhauser. “The Traveling Salesman Problem: A Survey”. In: *Operations Research* 16.3 (June 1968), pp. 538–558. ISSN: 0030-364X. DOI: `10.1287/opre.16.3.538`.

[3] Markus Bläser. “Metric TSP”. In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. Boston, MA: Springer US, 2008, pp. 517–519. ISBN: 978-0-387-30770-1. DOI: `10.1007/978-0-387-30162-4_230`.

[4] Tzu Li Chen et al. “An efficient hybrid algorithm for integrated order batching, sequencing and routing problem”. In: *International Journal of Production Economics* 159 (2015), pp. 158–167. ISSN: 09255273. DOI: `10.1016/j.ijpe.2014.09.029`.

[5] Nicos Christofides. *Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem*. Tech. rep. RR-388. Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[6] Koen Claessen and John Hughes. “QuickCheck”. In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming - ICFP ’00*. ICFP ’00. New York, New York, USA: ACM Press, 2000, pp. 268–279. ISBN: 1581132026. DOI: `10.1145/351240.351266`.

[7] William Cook. *Benchmark Information*. 2003. URL: `http://www.math.uwaterloo.ca/tsp/concorde/benchmarks/bench.html` (visited on 03/31/2020).

[8] William Cook. *Concorde TSP Solver*. 2015. URL: `http://www.math.uwaterloo.ca/tsp/concorde/index.html` (visited on 06/03/2020).

[9] William Cook. *Download Information*. 2005. URL: `http://www.math.uwaterloo.ca/tsp/concorde/downloads/downloads.htm` (visited on 06/03/2020).

[10] Edsger Wybe Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0029-599X. DOI: `10.1007/BF01386390`.

[11] Marco Dorigo and Luca Maria Gambardella. “Ant colony system: a cooperative learning approach to the traveling salesman problem”. In: *IEEE Transactions on Evolutionary Computation* 1.1 (Apr. 1997), pp. 53–66. ISSN: 1089778X. DOI: `10.1109/4235.585892`.

[12] Harold N. Gabow. “An Efficient Implementation of Edmonds’ Algorithm for Maximum Matching on Graphs”. In: *Journal of the ACM (JACM)* 23.2 (Apr. 1976), pp. 221–234. ISSN: 0004-5411. DOI: 10.1145/321941.321942.

[13] Jinxiang Gu, Marc Goetschalckx, and Leon F. McGinnis. “Research on warehouse operation: A comprehensive review”. In: *European Journal of Operational Research* 177.1 (Feb. 2007), pp. 1–21. ISSN: 03772217. DOI: 10.1016/j.ejor.2006.02.025.

[14] Gregory Gutin, Anders Yeo, and Alexey Zverovich. “Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP”. In: *Discrete Applied Mathematics* 117.1-3 (Mar. 2002), pp. 81–86. ISSN: 0166218X. DOI: 10.1016/S0166-218X(01)00195-0.

[15] Michael Held and Richard M. Karp. “A Dynamic Programming Approach to Sequencing Problems”. In: *Journal of the Society for Industrial and Applied Mathematics* 10.1 (Mar. 1962), pp. 196–210. ISSN: 0368-4245. DOI: 10.1137/0110015.

[16] Keld Helsgaun. “An effective implementation of the Lin–Kernighan traveling salesman heuristic”. In: *European Journal of Operational Research* 126.1 (Oct. 2000), pp. 106–130. ISSN: 03772217. DOI: 10.1016/S0377-2217(99)00284-2.

[17] J.A. Hoogeveen. “Analysis of Christofides’ heuristic: Some paths are more difficult than cycles”. In: *Operations Research Letters* 10.5 (July 1991), pp. 291–295. ISSN: 01676377. DOI: 10.1016/0167-6377(91)90016-I.

[18] Bernhard Korte and Jens Vygen. “Weighted Matching”. In: *Combinatorial Optimization: Theory and Algorithms.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 235–260. ISBN: 978-3-662-21711-5. DOI: 10.1007/978-3-662-21711-5_11.

[19] René de Koster, Tho Le-Duc, and Kees Jan Roodbergen. “Design and control of warehouse order picking: A literature review”. In: *European Journal of Operational Research* 182.2 (Oct. 2007), pp. 481–501. ISSN: 03772217. DOI: 10.1016/j.ejor.2006.07.009.

[20] Joseph B. Kruskal. “On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem”. In: *Proceedings of the American Mathematical Society* 7.1 (Feb. 1956), pp. 48–50. ISSN: 00029939. DOI: 10.2307/2033241.

[21] Osman Kulak, Yusuf Sahin, and Mustafa Egemen Taner. “Joint order batching and picker routing in single and multiple-cross-aisle warehouses using cluster-based tabu search algorithms”. In: *Flexible Services and Manufacturing Journal* 24.1 (Mar. 2012), pp. 52–80. ISSN: 1936-6582. DOI: 10.1007/s10696-011-9101-8.

[22] Chun Cheng Lin et al. “Joint order batching and picker Manhattan routing problem”. In: *Computers and Industrial Engineering* 95 (2016), pp. 164–174. ISSN: 03608352. DOI: 10.1016/j.cie.2016.03.009.

[23] Marek Matusiak et al. “A fast simulated annealing method for batching precedence-constrained customer orders in a warehouse”. In: *European Journal of Operational Research* 236.3 (Aug. 2014), pp. 968–977. ISSN: 03772217. DOI: 10.1016/j.ejor.2013.06.001.

[24] Tobias Mömke and Ola Svensson. “Approximating Graphic TSP by Matchings”. In: *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. IEEE, Oct. 2011, pp. 560–569. ISBN: 978-0-7695-4571-4. DOI: 10.1109/FOCS.2011.56. arXiv: 1104.3090.

[25] Robert C. Prim. “Shortest Connection Networks And Some Generalizations”. In: *Bell System Technical Journal* 36.6 (Nov. 1957), pp. 1389–1401. ISSN: 00058580. DOI: 10.1002/j.1538-7305.1957.tb01515.x.

[26] Gerhard Reinelt. “TSPLIB—A Traveling Salesman Problem Library”. In: *ORSA Journal on Computing* 3.4 (Nov. 1991), pp. 376–384. ISSN: 0899-1499. DOI: 10.1287/ijoc.3.4.376.

[27] Daniel J. Rosenkrantz, Richard Edwin Stearns, and Philip M. Lewis. “Approximate algorithms for the traveling salesperson problem”. In: *15th Annual Symposium on Switching and Automata Theory (swat 1974)*. IEEE, Oct. 1974, pp. 33–42. DOI: 10.1109/SWAT.1974.4.

[28] André Scholz and Gerhard Wäscher. “Order Batching and Picker Routing in manual order picking systems: the benefits of integrated routing”. In: *Central European Journal of Operations Research* 25.2 (June 2017), pp. 491–520. ISSN: 1435-246X. DOI: 10.1007/s10100-017-0467-x.

[29] Vijay V. Vazirani. *Approximation Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. ISBN: 978-3-642-08469-0. DOI: 10.1007/978-3-662-04565-7.

[30] Jaeyeon Won and Sigurdur Olafsson. “Joint order batching and order picking in warehouse operations”. In: *International Journal of Production Research* 43.7 (Apr. 2005), pp. 1427–1442. ISSN: 0020-7543. DOI: 10.1080/00207540410001733896.

A
Raw benchmark results

A.1 Quality benchmarks

Table A.1: Results of quality benchmarks for the shortest Hamiltonian path problem solvers.

Instance	Solver	Path length	Approx. ratio
`induce10`	`concorde`	177	—
`induce10`	`nn`	183	1.0339
`induce10`	`td`	275	1.5537
`burma14`	`concorde`	3 170	—
`burma14`	`nn`	3 688	1.1634
`burma14`	`td`	3 661	1.1549
`induce15`	`concorde`	249	—
`induce15`	`nn`	289	1.1606
`induce15`	`td`	307	1.2329
`ulysses22`	`concorde`	6 679	—
`ulysses22`	`nn`	10 874	1.6281
`ulysses22`	`td`	8 825	1.3213
`induce25`	`concorde`	329	—
`induce25`	`nn`	361	1.0973
`induce25`	`td`	401	1.2188
`att48`	`concorde`	10 095	—
`att48`	`nn`	13 923	1.3792
`att48`	`td`	14 269	1.4135
`induce50`	`concorde`	425	—
`induce50`	`nn`	583	1.3718
`induce50`	`td`	585	1.3765
`induce75`	`concorde`	595	—
`induce75`	`nn`	725	1.2185
`induce75`	`td`	753	1.2655
`pr76`	`concorde`	107 739	—
`pr76`	`nn`	152 032	1.4111

Table A.1 – continued from previous page.

pr76	td	145 026	1.3461
induce100	concorde	677	—
induce100	nn	799	1.1802
induce100	td	849	1.2541
bier127	concorde	117 834	—
bier127	nn	129 693	1.1006
bier127	td	157 482	1.3365
induce150	concorde	718	—
induce150	nn	870	1.2117
induce150	td	938	1.3064
d198	concorde	14 641	—
d198	nn	17 627	1.2039
d198	td	18 133	1.2385
induce200	concorde	803	—
induce200	nn	1 065	1.3263
induce200	td	1 125	1.4010
a280	concorde	2 559	—
a280	nn	3 126	1.2216
a280	td	3 552	1.3880
induce300	concorde	945	—
induce300	nn	1 171	1.2392
induce300	td	1 371	1.4508
induce400	concorde	1 015	—
induce400	nn	1 219	1.2010
induce400	td	1 453	1.4315
gr431	concorde	169 965	—
gr431	nn	208 730	1.2281
gr431	td	226 727	1.3340
u574	concorde	36 858	—
u574	nn	50 327	1.3654
u574	td	49 465	1.3420
induce600	concorde	1 112	—
induce600	nn	1 320	1.1871
induce600	td	1 708	1.5360
induce800	concorde	1 214	—
induce800	nn	1 396	1.1499
induce800	td	1 806	1.4876
vm1084	concorde	238 723	—
vm1084	nn	304 432	1.2753

Table A.1 – continued from previous page.

`vm1084`	`td`	315 767	1.3227

Table A.2: Results of quality benchmarks for the batch optimization problem solvers on the full instances.

Instance	BOP Solver	SHPP Solver	Path length
`orders60`	`greedy`	`concorde`	113
`orders60`	`greedy`	`nn`	113
`orders60`	`greedy`	`td`	113
`orders60`	`original`	`concorde`	339
`orders60`	`original`	`nn`	349
`orders60`	`original`	`ss`	485
`orders60`	`original`	`td`	393
`orders60`	`random1000`	`concorde`	257
`orders60`	`random1000`	`nn`	287
`orders60`	`random1000`	`td`	313
`orders107`	`greedy`	`concorde`	95
`orders107`	`greedy`	`nn`	95
`orders107`	`greedy`	`td`	97
`orders107`	`original`	`concorde`	341
`orders107`	`original`	`nn`	407
`orders107`	`original`	`ss`	397
`orders107`	`original`	`td`	381
`orders107`	`random1000`	`concorde`	231
`orders107`	`random1000`	`nn`	253
`orders107`	`random1000`	`td`	255
`orders223`	`greedy`	`concorde`	101
`orders223`	`greedy`	`nn`	101
`orders223`	`greedy`	`td`	127
`orders223`	`original`	`concorde`	403
`orders223`	`original`	`nn`	443
`orders223`	`original`	`ss`	605
`orders223`	`original`	`td`	487
`orders223`	`random1000`	`concorde`	275
`orders223`	`random1000`	`nn`	313
`orders223`	`random1000`	`td`	321
`orders769`	`greedy`	`concorde`	87
`orders769`	`greedy`	`nn`	87
`orders769`	`greedy`	`td`	87
`orders769`	`original`	`concorde`	205
`orders769`	`original`	`nn`	205

Table A.2 – continued from previous page.

`orders769`	`original`	`ss`	261
`orders769`	`original`	`td`	205
`orders769`	`random1000`	`concorde`	179
`orders769`	`random1000`	`nn`	183
`orders769`	`random1000`	`td`	189
`orders1531`	`greedy`	`concorde`	33
`orders1531`	`greedy`	`nn`	33
`orders1531`	`greedy`	`td`	33
`orders1531`	`original`	`concorde`	199
`orders1531`	`original`	`nn`	207
`orders1531`	`original`	`ss`	261
`orders1531`	`original`	`td`	207
`orders1531`	`random1000`	`concorde`	307
`orders1531`	`random1000`	`nn`	351
`orders1531`	`random1000`	`td`	351
`orders5176`	`greedy`	`concorde`	33
`orders5176`	`greedy`	`nn`	33
`orders5176`	`greedy`	`td`	33
`orders5176`	`original`	`concorde`	225
`orders5176`	`original`	`nn`	225
`orders5176`	`original`	`ss`	249
`orders5176`	`original`	`td`	283
`orders5176`	`random1000`	`concorde`	261
`orders5176`	`random1000`	`nn`	295
`orders5176`	`random1000`	`td`	289

Table A.3: Results of quality benchmarks for the batch optimization problem solvers on the random subinstances.

Full instance	BOP Solver	SHPP Solver	Avg. approx. ratio
`orders107`	`brute-force`	`concorde`	—
`orders107`	`brute-force`	`nn`	1.0627
`orders107`	`brute-force`	`td`	1.0879
`orders107`	`greedy`	`concorde`	1.0570
`orders107`	`greedy`	`nn`	1.1440
`orders107`	`greedy`	`td`	1.1711
`orders1531`	`brute-force`	`concorde`	—
`orders1531`	`brute-force`	`nn`	1.0794
`orders1531`	`brute-force`	`td`	1.1342
`orders1531`	`greedy`	`concorde`	1.0386
`orders1531`	`greedy`	`nn`	1.1546

Table A.3 – continued from previous page.

orders1531	greedy	td	1.1949
orders223	brute-force	concorde	—
orders223	brute-force	nn	1.0939
orders223	brute-force	td	1.1348
orders223	greedy	concorde	1.0368
orders223	greedy	nn	1.1782
orders223	greedy	td	1.2067
orders5176	brute-force	concorde	—
orders5176	brute-force	nn	1.0801
orders5176	brute-force	td	1.1154
orders5176	greedy	concorde	1.0376
orders5176	greedy	nn	1.1611
orders5176	greedy	td	1.1925
orders60	brute-force	concorde	—
orders60	brute-force	nn	1.0880
orders60	brute-force	td	1.1254
orders60	greedy	concorde	1.0299
orders60	greedy	nn	1.1418
orders60	greedy	td	1.1889
orders769	brute-force	concorde	—
orders769	brute-force	nn	1.0349
orders769	brute-force	td	1.0632
orders769	greedy	concorde	1.0648
orders769	greedy	nn	1.1326
orders769	greedy	td	1.1829

A.2 Runtime benchmarks

Table A.4: Results of runtime benchmarks for the shortest Hamiltonian path problem solvers.

		Runtime (ms)	
Instance	Solver	Avg.	Std. dev.
induce10	concorde	—	—
induce10	nn	0.0276	0.0006
induce10	td	0.0536	0.0008
burma14	concorde	—	—
burma14	nn	0.0511	0.0015
burma14	td	0.0946	0.0013

Table A.4 – continued from previous page.

induce15	concorde	—	—
induce15	nn	0.0660	0.0036
induce15	td	0.1109	0.0020
ulysses22	concorde	—	—
ulysses22	nn	0.1482	0.0029
ulysses22	td	0.2329	0.0043
induce25	concorde	—	—
induce25	nn	0.1836	0.0027
induce25	td	0.3039	0.0090
att48	concorde	—	—
att48	nn	0.6780	0.0099
att48	td	1.0302	0.0182
induce50	concorde	—	—
induce50	nn	0.8038	0.0146
induce50	td	1.1952	0.0267
induce75	concorde	—	—
induce75	nn	1.8827	0.0287
induce75	td	2.6710	0.0412
pr76	concorde	—	—
pr76	nn	1.8612	0.0292
pr76	td	2.7539	0.0794
induce100	concorde	—	—
induce100	nn	3.3030	0.0687
induce100	td	4.7568	0.1331
bier127	concorde	—	—
bier127	nn	5.2136	0.1524
bier127	td	7.2544	0.1320
induce150	concorde	—	—
induce150	nn	7.6542	0.1037
induce150	td	10.7168	0.2839
d198	concorde	—	—
d198	nn	12.6624	0.2066
d198	td	17.7469	0.2874
induce200	concorde	—	—
induce200	nn	14.2563	0.2757
induce200	td	19.3187	0.5030
a280	concorde	—	—
a280	nn	25.8199	0.4383
a280	td	36.3313	0.9853

Table A.4 – continued from previous page.

induce300	concorde	—	—
induce300	nn	32.1902	0.5644
induce300	td	44.2235	0.5809
induce400	concorde	—	—
induce400	nn	60.0223	0.9050
induce400	td	78.5651	0.7326
gr431	concorde	—	—
gr431	nn	63.6226	1.1183
gr431	td	88.9238	2.2934
u574	concorde	—	—
u574	nn	118.2385	1.4502
u574	td	162.1611	1.2603
induce600	concorde	—	—
induce600	nn	162.8426	1.5506
induce600	td	181.5653	1.9214
induce800	concorde	—	—
induce800	nn	240.9934	2.2008
induce800	td	323.1154	1.8290
vm1084	concorde	—	—
vm1084	nn	464.3692	1.3664
vm1084	td	620.2350	12.0491

Table A.5: Results of runtime benchmarks for the batch optimization problem solvers.

			Runtime (ms)			
			without cache		with cache	
Instance	BOP solver	SHPP solver	Avg.	Std. dev.	Avg.	Std. dev.
orders60	greedy	concorde	—	—	—	—
orders60	greedy	nn	23.1631	0.3339	10.9326	0.1221
orders60	greedy	td	42.3346	0.2511	16.0978	0.1083
orders60	original	concorde	—	—	—	—
orders60	original	nn	—	—	—	—
orders60	original	ss	—	—	—	—
orders60	original	td	—	—	—	—
orders60	random1000	concorde	—	—	—	—
orders60	random1000	nn	427.1704	1.9523	440.1575	1.1847
orders60	random1000	td	652.8562	4.6928	677.7756	1.7451
orders107	greedy	concorde	—	—	—	—

Table A.5 – continued from previous page.

orders107	greedy	nn	49.7403	0.6913	38.5130	0.3528
orders107	greedy	td	88.5420	0.6858	62.5354	0.7389
orders107	original	concorde	—	—	—	—
orders107	original	nn	—	—	—	—
orders107	original	ss	—	—	—	—
orders107	original	td	—	—	—	—
orders107	random1000	concorde	—	—	—	—
orders107	random1000	nn	204.7135	1.5489	212.1029	1.1829
orders107	random1000	td	326.8217	2.5275	336.2013	0.7780
orders223	greedy	concorde	—	—	—	—
orders223	greedy	nn	182.2618	1.8502	140.2121	0.5095
orders223	greedy	td	263.6376	2.3911	175.9792	0.8859
orders223	original	concorde	—	—	—	—
orders223	original	nn	—	—	—	—
orders223	original	ss	—	—	—	—
orders223	original	td	—	—	—	—
orders223	random1000	concorde	—	—	—	—
orders223	random1000	nn	427.0813	1.2968	438.0867	1.4155
orders223	random1000	td	673.7034	3.0283	671.3524	3.0032
orders769	greedy	concorde	—	—	—	—
orders769	greedy	nn	928.1278	6.2639	500.7164	1.4056
orders769	greedy	td	1 565.3835	7.5620	750.7682	1.7795
orders769	original	concorde	—	—	—	—
orders769	original	nn	—	—	—	—
orders769	original	ss	—	—	—	—
orders769	original	td	—	—	—	—
orders769	random1000	concorde	—	—	—	—
orders769	random1000	nn	224.5199	2.1913	233.4668	1.8433
orders769	random1000	td	359.9413	3.1072	370.6562	2.0535
orders1531	greedy	concorde	—	—	—	—
orders1531	greedy	nn	324.7300	3.3597	173.5984	1.1849
orders1531	greedy	td	642.9733	7.4783	194.6437	1.2925
orders1531	original	concorde	—	—	—	—
orders1531	original	nn	—	—	—	—
orders1531	original	ss	—	—	—	—
orders1531	original	td	—	—	—	—
orders1531	random1000	concorde	—	—	—	—
orders1531	random1000	nn	592.5162	3.9704	607.7865	3.0803
orders1531	random1000	td	902.8181	20.6404	919.9067	4.4830
orders5176	greedy	concorde	—	—	—	—
orders5176	greedy	nn	800.8034	2.9596	459.1361	1.6740
orders5176	greedy	td	1 624.7831	10.1797	511.5910	0.9987
orders5176	original	concorde	—	—	—	—

Table A.5 – continued from previous page.

`orders5176`	`original`	`nn`	—	—	—	—
`orders5176`	`original`	`ss`	—	—	—	—
`orders5176`	`original`	`td`	—	—	—	—
`orders5176`	`random1000`	`concorde`	—	—	—	—
`orders5176`	`random1000`	`nn`	354.7399	3.1616	369.8379	1.5883
`orders5176`	`random1000`	`td`	547.2488	4.1344	559.2276	1.1916