Utrecht University

Faculty of Sciences

Local Search and Machine Learning for Capacitated Vehicle Routing

Abstract

In a fast-changing economy, demand for high-quality logistics solutions increases rapidly. The number of parcels shipped worldwide has doubled since 2014 and is expected to have doubled again by 2025. With this growing demand comes the need for first-class vehicle scheduling, both from an environmental as well as an economic standpoint. The goal is to find optimal planning solutions, but since these problems are $\mathcal{NP}$-hard, this is not always possible in reasonable time. Therefore, Local Search methods, like Simulated Annealing, are often used to find good (not necessarily optimal) solutions. This study aims to find a way to combine machine learning and simulated annealing by looking for good features with machine learning of existing solutions and guiding the SA algorithm towards good solutions.

In this thesis, we look into capacitated vehicle scheduling in the context of garbage collection. We propose a new variant of the SA algorithm, Smart Simulated Annealing (SSA). This algorithm uses *OrderVectors* found by AdaGrad, a stochastic gradient descent algorithm. We show that the number of iterations needed to find solutions compared to SA can be reduced by 10 percent for short runs and by 17 percent for long runs. Our results indicate that SSA can reduce the number of iterations needed to find good solutions. Future research can focus on optimizing SSA and its parameters and on the robustness of the solutions to check if theoretical solutions can be used in real-life.

Keywords: Capacitated vehicle routing, Simulated Annealing, Machine Learning

Bachelorthesis

Teun Druijf

Mathematics

Supervisor:

Dr. J.A. Hoogeveen
Utrecht University

June 12, 2020

Contents

1 Introduction

In a fast-changing economy, demand for high-quality logistics solutions increases rapidly. The number of parcels shipped worldwide has doubled since 2014 and is expected to have doubled again by 2025 [2]. With this growing demand comes the need for high-quality vehicle scheduling, both from an environmental as well as an economic standpoint. In this thesis, we will consider vehicle scheduling in the context of garbage collection. This is a capacitated single depot multi-vehicle scheduling problem, a special case of the single-depot multi-vehicle scheduling problem. Solutions we find in this thesis can also be applied to package delivery, medicine distribution, or food and water distribution in third world countries among many other applications.

Single Depot Multi-Vehicle scheduling

In a single depot multi-vehicle scheduling problem, there is one depot and multiple vehicles. Furthermore, there are customers that need a delivery or need something picked up, a package for example. For each vehicle, a trip has to be made. Each trip is represented by an ordered series of nodes between stops at the depot. The goal is to minimize the driving costs and optional penalties. Penalties may be added to the problem to allow declining orders or working overtime. Also, in some cases, the number of vehicles is variable and in that case, vehicle costs are also part of the objective function.

In a capacitated single depot multi-vehicle scheduling problem, we use vehicles with a certain capacity. For example, a garbage truck. We add a constraint to the problem that does not allow the truck to use more than its capacity for the trip. So there can not be more garbage picked up than the capacity allows. In the case of parcel delivery, the truck can only deliver as many packages in one trip as the capacity of the truck allows.

These problems are all $\mathcal{NP}$-hard [20], meaning that they are hard to solve to optimality. Hence, we might not be able to find an optimal solution within a reasonable time. For our case study, we use a $\mathcal{NP}$-hard problem and thus we will use Local Search since we won't be able to find an optimal solution in reasonable time.

Applying Local Search for vehicle scheduling problems has been previously studied by Groër et al. [16] and by Mazidi et al. [21]. For a full overview of the different vehicle scheduling problems we refer to the overview by Eksioglu et al. [12] or by Braekers et al. [4].

We will extend the Local Search algorithm and use machine learning to guide our algorithm on the basis of earlier found solutions. By identifying good features of solutions, we can reduce the number of iterations needed to find a good solution. With an increasing number of vehicles and orders, the problem becomes even harder to solve and this increases the need for a reduction of the time needed to find a good solution. There has been little research into this combination of machine learning and simulated annealing.

Chou et al. [6] used a guiding function to find solutions for protein models and found the lowest energy state compared to other methods. This is related to our research since their guiding function is based on features of proteins and stimulates adding those features to the solution. Vermeulen [30] proposed a new variant of SA, SACos, which is comparable to our algorithm SSA and found that SACos outperformed SA. Solutions of lower-cost were found in the same number of iterations. We will extend this research and look into whether the number of iterations can be reduced while still finding solutions of equal quality.

In our case study, we will focus on garbage collection. We will make a schedule for two garbage trucks for one week (Monday - Friday). There is an overview of 1177 companies who want their garbage picked up with a given frequency (number of times per week). We can decline these orders, but this comes at a cost. The garbage trucks have a limited capacity and can dump during the day at a single landfill. In addition, this landfill is also the starting and endpoint of each trip.

The objective is to minimize the total time that a garbage truck is used, which includes driving, dumping, and emptying time. The dumping time is the time needed to empty the garbage truck at the landfill and the emptying time is the time needed to pick up the garbage for an order. The penalty for declining orders, expressed in minutes, is added to this time.

We have chosen to use this case study since we already have a good solver in place, capable of generating a dataset of good solutions. Remark that we can easily interchange the context in which we apply the optimization and machine learning techniques. On the scheduling side, there is no difference between picking up or delivering at a given set of locations. This model can be used for package delivery and airline scheduling among many other applications.

Based on a dataset of more than three thousand good solutions we will try to find good solutions for the capacitated vehicle scheduling problem in fewer iterations. For this, we have developed the Smart Simulated Annealing algorithm. This algorithm is based on Simulated Annealing and has a new acceptance function.

In this thesis, we will give a formal model description in Section 2. In Section 3 we will explain Local Search and Simulated Annealing, the algorithm that we will use in this thesis. We will give an overview of the data mining and machine learning techniques used in Section 4. In this section, we will also give an example for milk pickup, which is comparable to our case study. Then, in Section 5, we explain the experiments conducted and present the results. We discuss those results in Section 6.

Note to the reader: in this thesis, you may find some red or blue text boxes, just like with the 'Single depot multi-vehicle scheduling' at the beginning of the introduction. These boxes function as explainers (red boxes) or give an insight into the implementation of some of the algorithms (blue boxes). These boxes can be omitted on first reading, but provide interesting insights or can help understanding subjects that are not familiar to the reader.

2 Problem description

In this section, we will give a brief introduction into the problem that we will use as a case study for this thesis. Thereafter we will give a complete problem description.

2.1 Introduction

The Van Gansewinkel Group, nowadays known as Renewi after a merger in 2017, is a large waste management company based in the Netherlands with operations in northwest Europe. They had to find the most cost-efficient vehicle schedule for industrial garbage collection in the province 'Noord-Brabant', the Netherlands. Companies can place orders and Van Gansewinkel can decide whether they accept or decline the order. For each order, the location of the company, emptying time, amount of garbage, and frequency is given. The frequency is equal to the number of times that the garbage should be picked up per week.
We consider the scheduling problem for two trucks and one week (Monday - Friday). The costs are equal to the total driving, emptying, and dumping time and penalties for declining orders. The cost of declining an order is equal to three times the emptying time for an order.

2.2 Problem description

Assumptions

We assume that there are no constraints regarding the crew of the garbage trucks. So we do not take working times and breaks into account.

Hard Constraints

The following hard constraints must be satisfied if not, a solution is called infeasible:

- Garbage trucks can only be used on weekdays between 6 a.m. and 6 p.m. and should be empty at the landfill outside of those working hours.
- When a garbage truck is not used, it should be empty at the landfill.
- There can not be more garbage in the truck than the capacity allows.
- For each order, a frequency is given. This is the number of times garbage must be collected. The collection has to be done according to a frequency pattern, for each frequency there is at least one pattern and we are free to choose which pattern we will use for each order. A frequency pattern prescribes on which days of the week garbage must be collected. The different patterns for all possible frequencies are given in Table 1. Garbage collection must take place on each of the 'green' days in the pattern.

Pattern	Mon	Tue	Wed	Thu	Fri
1	🟢	🔴	🔴	🔴	🔴
2	🔴	🟢	🔴	🔴	🔴
3	🔴	🔴	🟢	🔴	🔴
4	🔴	🔴	🔴	🟢	🔴
5	🔴	🔴	🔴	🔴	🟢

(a) **Frequency 1**

Pattern	Mon	Tue	Wed	Thu	Fri
1	🟢	🔴	🔴	🟢	🔴
2	🔴	🟢	🔴	🔴	🟢

(b) **Frequency 2**

Pattern	Mon	Tue	Wed	Thu	Fri
1	🟢	🔴	🟢	🔴	🟢

(c) **Frequency 3**

Pattern	Mon	Tue	Wed	Thu	Fri
1	🔴	🟢	🟢	🟢	🟢
2	🟢	🔴	🟢	🟢	🟢
3	🟢	🟢	🔴	🟢	🟢
4	🟢	🟢	🟢	🔴	🟢
5	🟢	🟢	🟢	🟢	🔴

(d) **Frequency 4**

Pattern	Mon	Tue	Wed	Thu	Fri
1	🟢	🟢	🟢	🟢	🟢

(e) **Frequency 5**

Table 1: Frequency patterns for different frequencies

Soft Constraints

If possible, each order should be fulfilled. A penalty for non-fulfilled orders is added to the operating time if this constraint is not met for a specific order, remark that orders can not be partially fulfilled. This penalty is equal to three times the emptying time for that order.

Objective

The goal is to find a solution with minimum cost. As previously mentioned, the costs are equal to the sum of traveling, emptying, and dumping times of the trucks. If an order is not fulfilled (and per definition declined), the penalty for declining the order is added to the total cost.

Data and variables

The following input data is given and can not be changed:

- Two identical garbage trucks are available, with a capacity of 20.000 liter.
- A list of 1177 orders. For each other, the frequency, amount of garbage (not compacted, will be compacted in the garbage truck and decrease volume by 80 percent), emptying time, and location is known.
- An asymmetrical distance matrix containing traveling times for each pair of companies and for each order and the landfill.
- Dumping the garbage at the landfill takes 30 minutes independent of how full the truck is. Both trucks can dump at the same time.

3 Local Search

In this chapter, we will give an introduction into Local Search and its many implementations, explain Simulated Annealing, and demonstrate the working of neighborhoods.

3.1 Introduction

One thing we are interested in when we create an algorithm is the running time of this algorithm. This is a metric for how long it will take to find a solution. We are interested in this metric since we only have limited time available. The running time, expressed in the number of operations, of an algorithm is also known as the complexity. When we analyze an algorithm, we want to determine whether the complexity is polynomial with respect to the size of the input or not. This means that we want to know if the complexity of our algorithm with an input of size n is equal to $\mathcal{O}(n^k)$ for some positive real k. If the complexity of the fastest algorithm for our problem is polynomial we say that our problem is part of the class $\mathcal{P}$.

If the fastest algorithm for some problem is not part of the class $\mathcal{P}$, but the solutions can be verified and saved in polynomial time, then we say that the problem is part of the class $\mathcal{NP}$. We can verify a solution in polynomial time if we can check in polynomial time if the solution is feasible and meets our requirements. Remark that $\mathcal{P} \subseteq \mathcal{NP}$.

$\mathcal{P}$ versus $\mathcal{NP}$

One of the Millenium prize problems is the "$\mathcal{P}$ versus $\mathcal{NP}$" problem. The question at hand is to prove whether $\mathcal{P} = \mathcal{NP}$ or if $\mathcal{P} \subsetneq \mathcal{NP}$. It is hard to prove, but it is assumed that $\mathcal{P} \subsetneq \mathcal{NP}$. To prove that all problems in $\mathcal{NP}$ can be solved in polynomial time, we have to show that there exists a polynomial algorithm for a $\mathcal{NP}$-complete problem. Intuitively, a $\mathcal{NP}$-complete problem is as 'hard' as all problems in $\mathcal{NP}$ and if we can find a solution in polynomial time for a $\mathcal{NP}$-complete problem, then we can also find a solution in polynomial time for all problems in $\mathcal{NP}$ and thus that $\mathcal{P} = \mathcal{NP}$. Remark that if we can prove that there is no polynomial algorithm for a $\mathcal{NP}$-complete problem, then $\mathcal{P} \subsetneq \mathcal{NP}$ [8, 9].

As mentioned in the introduction, a problem can also be $\mathcal{NP}$-hard. A problem is $\mathcal{NP}$-hard if it is as 'hard' as all problems in $\mathcal{NP}$, but necessarily part of $\mathcal{NP}$ [9].

Consider the case where we have a problem that can be solved using an exponential algorithm. Assume that instances of this problem can be verified in polynomial time. This means that this problem is part of $\mathcal{NP}$. It could be that the exponential algorithm is still quite fast for small instances. With larger instances however, this will become increasingly more difficult. This is where we could use Local Search.

Local Search is an iterative optimization technique where a current (possibly infeasible) solution is iteratively changed and the result of this change is either accepted or declined. We might not find an optimal solution using Local Search but we can get a solution close to the optimal solution. Although, in contrast to approximation algorithms, we are not guaranteed to find a solution within a certain distance from the optimal solution.

There are multiple variants of Local Search. In this thesis, we will use Simulated Annealing. We will discuss this algorithm and two other algorithms, Hill Climbing and Tabu Search, that are examples of Local Search algorithms. First, we have to define a neighborhood and a neighbor.

Definition 3.1. Neighborhood & Neighbor A neighborhood is a set of neighbors. All these neighbors are solutions that are identical to the input solution except for one specific change. These changes can be small, like adding or removing an item to or from the solution, or they can be bigger, like reversing the order of (a part of) the solution. For more examples of neighborhoods in the context of vehicle routing problems, see Section 3.3.

With Local Search, we define neighborhoods and procedures to find the neighbors in these neighborhoods. Algorithms that use Local Search will prescribe how the neighborhoods should be evaluated and how the iterative process can use these neighborhoods to find a solution. Below, we will discuss three variants of Local Search, Hill Climbing, Tabu Search, and Simulated Annealing.

Minimization problem

In this thesis, we will consider minimization problems. The goal in these types of problems is to minimize a given objective function. This function is also known as the cost function. With this function, we can calculate the cost of a solution of the problem. This function depends on the so-called decision variables. These are variables that we can change in order to modify a solution and hopefully decrease the cost of the solution.

Hill Climbing ([14, 26]) The Hill Climbing method will explore all neighbors and choose the one that is the best improvement over the current cost. If there exists no neighbor that will improve that cost, the algorithm will stop. There are variants of this algorithm that will explore bigger neighborhoods, this means that the neighbors are 'further away' from the current solution. For example, this could mean that more variables are changed or that some variable is changed more compared to the standard Hill Climbing algorithm. This will still result in a local optimum in most cases. This algorithm is good for problems with only a few local optima, since using multiple restarts with random starting solutions will likely give an optimal solution. This makes Hill Climbing especially useful for convex optimization problems.

Convex Optimization

Convex problems are optimization problems which have a convex objective function $f_0(x)$, convex inequality functions $f_i(x)$ and affine equality constraints. This means that for the convex real functions $f_0(x), f_1(x), \ldots, f_m(x)$ we can create a convex optimization problem of the form:

$$\begin{aligned} &\min\ f_0(x) \\ &\text{subject to } f_i(x) \leq 0,\ i = 1, \ldots m \\ &A\vec{x} = \vec{b} \end{aligned}$$

We will state without proof[a] that for convex optimization problems, a local optimum is also a global optimum. This is something that can be used when using Hill Climbing optimization methods since these algorithms will find a local optimum and thus the optimal solution for convex problems [3].

[a] A proof can be found in [3] in paragraph 4.2.2

Example 3.1. Hill Climbing We can use hill climbing to find the minimum of a real valued function f depending on x. The neighborhood is defined by changing the value of x by a small amount and checking whether the function returns a lower value. We can continue this process until we have found a local minimum. This is the case if there are no neighbors left which give a lower function value. Functions with a few (or preferably only one) local minima are more suited for this technique (See Figure 1). Finding the minimum of $f(x) = (x-3)^2$ is independent of the starting position, but with $f(x) = \sin(x) + \sin\left(\frac{3x\pi}{2}\right) + \sin(3x) + 3$ it depends on the starting position which local minimum we will find. ♢

Tabu Search ([14, 15]) When using Tabu Search, a step that increases the cost is allowed if there are no neighbors that decrease the current cost. Furthermore, the algorithm will keep track of the solutions it has already 'visited' in a Tabu list and won't look at these solutions for a number of iterations. This algorithm will give deterministic results, meaning that given a begin solution, running the algorithm twice will result in the same solution.

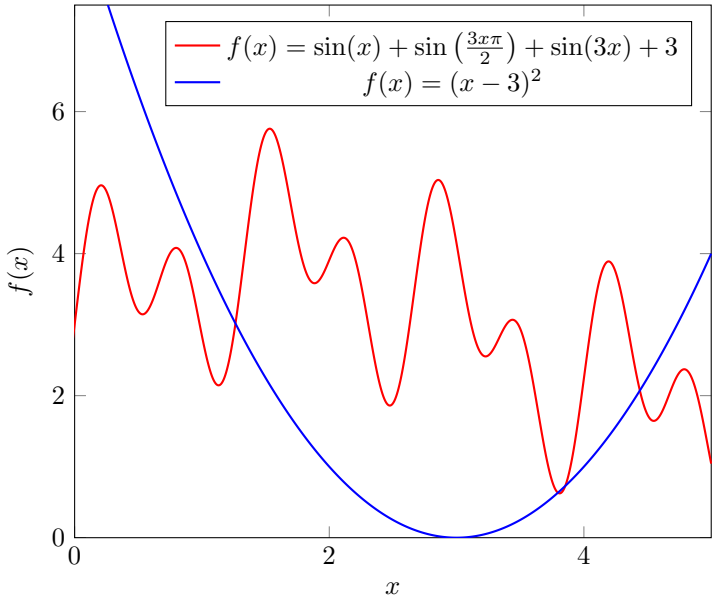

Figure 1: Example 3.1

Simulated Annealing ([7, 26]) Simulated Annealing will allow 'worsening' steps during the process of finding the optimal solution. If a step is not an improvement, the algorithm will conduct a chance experiment. This will determine, based on how much worse the solution will get and on the current state of the process if the neighbor is accepted. In the beginning, more 'worsening' steps are allowed compared to later in the run. The algorithm will not explore all possible neighbors each step but will randomly pick one and evaluate the outcome. We will go over Simulated Annealing and the acceptance function more extensively in the next section.

3.2 Simulated Annealing

As mentioned previously, we will use Simulated Annealing in this thesis. This is because our model has many different local optima. Also, since we have defined many neighborhoods with each numerous neighbors, we expect the number of iterations to be low if we explore each neighborhood and neighbor at each iteration. In our case study, this would imply that we would have to do more than a hundred thousand times more calculations per iteration.

We already have given a brief introduction into Simulated Annealing. With Simulated Annealing, we can find solutions for problems that have a large number of different local optima. This is the case since Simulated Annealing will allow in some circumstances for 'worsening' iterations which we need to escape local optima.

At the beginning of a run, a new starting solution will be generated using a given procedure. Depending on the use case, this can be an empty solution or a randomly generated solution given some constraints. After all, the starting solution does not need to be a good solution, it should be a solution that can be modified using the neighbors. A starting solution can fail to meet the hard constraints, but as long as there exists neighborhoods that can change the solution so that it meets the hard constraints, there is no problem. See Section 5.1 for the starting solutions that we will use in our case study.

Example 3.2. Knapsack The Knapsack problem is a $\mathcal{NP}$-hard optimization problem. The problem is defined as follows: Given a set $\mathcal{I}$ of n items, each item i of $\mathcal{I}$ has a value v_i and a size s_i. The problem is to find a subset $\mathcal{K}$ of $\mathcal{I}$ of the highest value such that the total size is smaller than or equal to S. Remark that we can only add complete items to $\mathcal{K}$ [24].

When we solve this problem with Local Search we can define a neighborhood that adds items to $\mathcal{K}$ (the Knapsack) and a neighborhood that removes items from $\mathcal{K}$. As a starting solution, we can use an empty set $\mathcal{K}$ or we can put a random selection of $\mathcal{I}$ into $\mathcal{K}$. $\diamondsuit$

At each iteration, the algorithm will randomly select a neighborhood and a neighbor out of that neighborhood to explore. If this leads to an improvement, the change will be accepted, if not, a chance experiment will be conducted. Let v_1 be the cost of the solution before changing and v_2 be the cost after the change. Then this change will be accepted if for a uniformly chosen real-valued $r \in [0, 1]$, and the current temperature T, the following holds:

$$\exp\left\{\frac{v_1 - v_2}{T}\right\} \geq r$$

The probability that we will accept a 'worsening' step is given in Figure 2. We see that for larger values of T we accept certain increases in cost with a higher probability compared to smaller values of T.

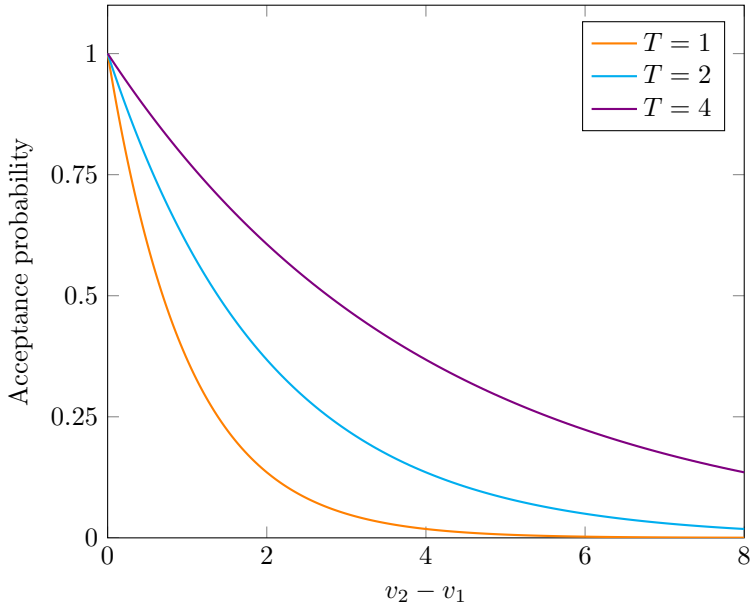

Figure 2: Acceptance test

The temperature T will be updated by multiplying it with a constant factor α. This constant α must be real-valued and part of the interval $[0, 1]$. In this thesis, we will update T 8000 times during a run. The number of iterations between each run will be calculated using this condition. Furthermore, we will use $T = 20$ minutes and $\alpha = 0.9992$, for more information about the values used for our case study, see Section 5. In Figure 3 we have plotted the development of T over time, we have also identified the number of updates needed to halve T. The SA algorithm is stated in Algorithm 1 [22]. Remark that since we allow the cost of the solution to increase we also save the best solution found. This way, if we escape a local minimum but end up in another worse local minimum we still have the best solution found up to that point.

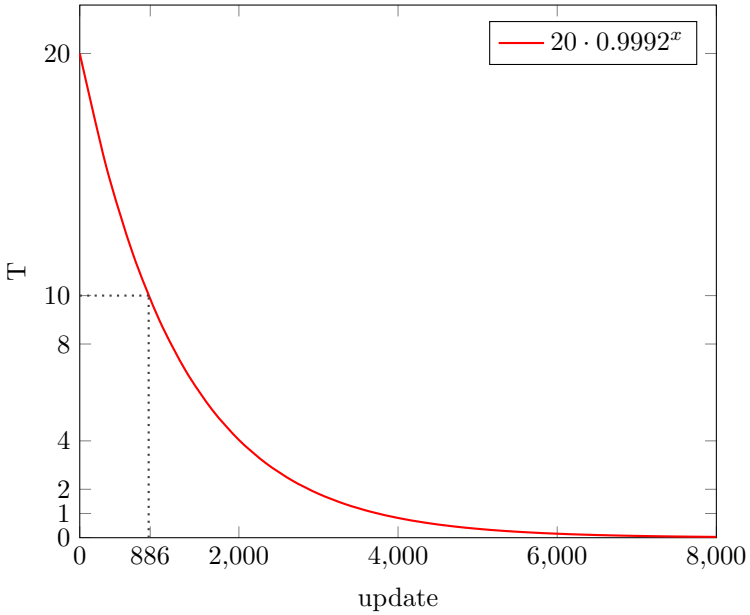

Figure 3: Temperature over time

Annealing

The name of Simulated Annealing is closely connected to the process of annealing metals. When a metal is heated to a temperature where the solid melts all atoms can move freely. If the temperature is then slowly decreased the atoms can arrange themselves. After the metal has solidified again and the annealing has been performed correctly the system will be in its minimum energy state [29].
Simulated Annealing follows the same procedure wherein the beginning many 'worsening' steps are allowed and as the temperature decreases, the cost will converge to a local optimum as less 'worsening' steps are allowed. This will hopefully lead to a global minimum or local minimum with a cost close to the global minimum.

Algorithm 1 Simulated Annealing algorithm

1: Create a new starting solution $\mathcal{S}$
2: Set $\mathcal{S}_{\text{best}}$ equal to $\mathcal{S}$
3: **while** Fixed number of iterations has not been reached **do**
4: If a fixed fraction[1]of the number of iterations has passed, multiply T by α.
5: Select a random neighbor n.
6: Calculate the cost with the new neighbor, the new cost is equal to $\mathcal{S}_{\text{new}}$.
7: If the new cost is less than the old cost, accept the change. If the new cost is not less than the old cost, select a random number $\mathcal{R}$ between 0 and 1. Now accept the change if

$$\exp\left\{\frac{\text{cost}(\mathcal{S}) - \text{cost}(\mathcal{S}_{\text{new}})}{T}\right\} \geq \mathcal{R}$$

and reject otherwise.
8: If the change is accepted, set $\mathcal{S}$ is $\mathcal{S}_{\text{new}}$.
9: If cost($\mathcal{S}$) is smaller than cost($\mathcal{S}_{\text{best}}$), set $\mathcal{S}_{\text{best}}$ is $\mathcal{S}$
10: When finished, save $\mathcal{S}_{\text{best}}$.

[1]In this thesis we will use $\frac{1}{8000}$, for a total of 8000 updates of T independent of the total number of iterations.

3.3 Traveling Salesman Problem

We will demonstrate the working of neighborhoods via an example about a traveling salesman. The Traveling Salesman Problem (TSP) is about a salesman who wants to visit a given set of cities exactly one time and return to the city where he started. Assume a possibly asymmetric distance matrix is given. In this example, we will use the Euclidean distance between two points on the map. Assume that the salesman can reach all nodes from each other node. This problem is $\mathcal{NP}$-complete [17]. When we are solving this problem using Local Search we should define the neighborhoods. In this example, we will go over three neighborhoods for this problem, Push, Mirror, and Swap[2] .

Definition 3.2. (Trip) For n cities, a trip is a finite sequence of length n, $(a_1 a_2 \dots a_n)$. Here, each a_i represents a city and after visiting a_n the salesman traveling will return to a_1.

Push A neighbor in the the Push-neighborhood (also known as $2,5$-opt) will pick a point in the trip and move it to a different spot in the sequence.
Formally this means that if point i is moved to position j in trip A, the new trip will become $(a_1 \dots a_{i-1} a_{i+1} \dots a_j a_i a_{j+1} \dots a_n)$ if $i < j$ and $(a_1 \dots a_{j-1} a_i a_j \dots a_{i-1} a_{i+1} \dots a_n)$ if $j < i$.

See for example the trip $(ADBCE)$ represented[3] in Figure 4a. It is clear that this trip is not an optimal solution. Using the Push-neighborhood, we will change the sequence by moving D in between C and E. This will give the new sequence $(ABCDE)$, which is drawn in Figure 4b. This represents an optimal solution of this case[4].

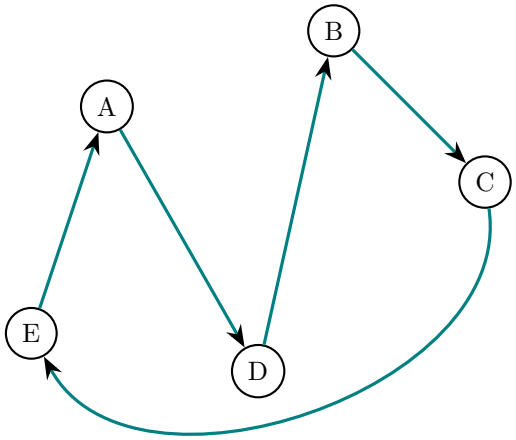

Figure 4a: Trip $(ADBCE)$

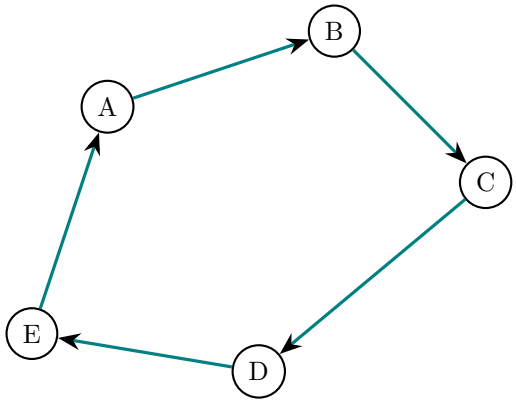

Figure 4b: Trip $(ABCDE)$

Mirror A neighbor in the Mirror-neighborhood (also known as 2-opt) will select a subsequence of sequence A and reverse the order of this subsequence within the main sequence.
As a result, for i and j, with $1 \leq i < j \leq n$ and trip A of length n, the new trip will become $(a_1 \dots a_{i-1} a_j a_{j-1} \dots a_{i+1} a_i a_{j+1} \dots a_n)$.

Consider the trip $(ADCBE)$ represented in Figure 5a. It is clear that is is not an optimal solution. Using the Mirror-neighborhood we will reverse the subtrip (DCB). This results in the trip $(ABCDE)$, which is drawn in Figure 5b and represents an optimal solution[5].

[2]Please note that some neighborhoods could be known by a different name.
[3]For clarity, some edges have been curved. We will use the Euclidean distance between two nodes, not the length of the edges.
[4]There is no guarantee to find an optimal solution after using a neighborhood for a certain number of times.
[5]See Footnote 4

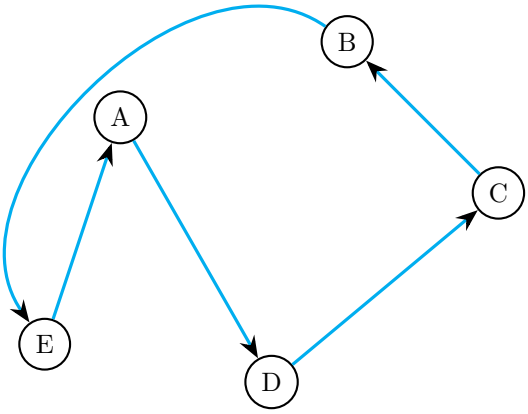

Figure 5a: Trip $(ADCBE)$

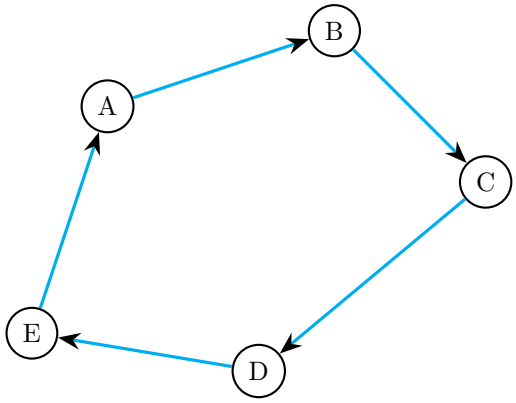

Figure 5b: Trip $(ABCDE)$

Swap A neighbor in the Swap-neighborhood will swap two cities in the trip. This means that for i and j with $i < j$ the trip A will be changed to $(a_1 \dots a_{i-1} a_j a_{i+1} \dots a_{j-1} a_i a_{j+1} \dots a_n)$.

Consider the trip $(AECDBF)$ in Figure 6a. By using the Swap-neighborhood to swap the positions of B and E, we get the trip $(ABCDEF)$ which is an optimal solution[6] and is drawn in Figure 6b.

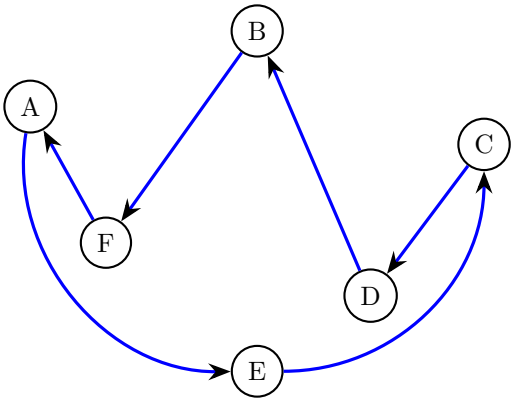

Figure 6a: Trip $(AECDBF)$

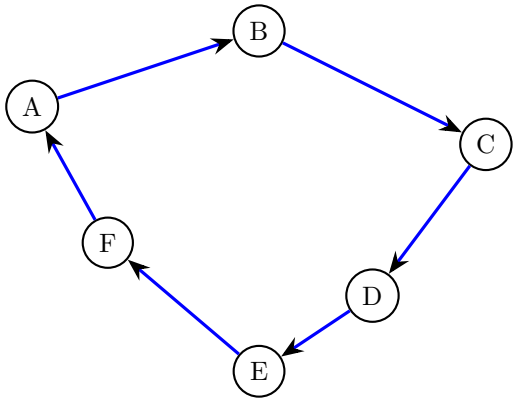

Figure 6b: Trip $(ABCDEF)$

In this example we have given some examples of neighborhoods that can be used to find a solution for the Traveling Salesman Problem with Local Search. ♢

[6]See Footnote 4

4 Data Mining and Machine Learning

In this section, we will give an introduction into the data mining technique that we will use and how the data is evaluated. In the first section, we will go over the data generation and data extraction. In the second section, we will explain the machine learning techniques that have been used in our case study. In the third section, we will formally introduce *OrderVectors*. These vectors will be used to guide the Smart Simulated Annealing (SSA) algorithm.

4.1 Data Mining

When we have found multiple good solutions for an optimization problem using Local Search, we are interested in features of these solutions that make a solution a good solution. If we can identify these features, we can guide our SSA algorithm. Neighbors that increase the number of good features will have a higher probability of being accepted if the new cost is greater than than the old cost, compared to SA. Also, neighbors that decrease the number of good features will have a lower probability of being accepted if the new cost is worse compared to the previous cost. This is done by changing the value of the temperature used in the acceptance test (Line 7 of Algorithm 1). See Section 4.3.1 for more explanation of SSA.

We will give some intuition into what makes a feature a good feature using a simple example. Remark that an instance of a single depot multi-vehicle scheduling problem can be represented as a graph, where the nodes of the graph correspond to the location of the stops. When we compare the distances between two nodes in the graph with the actual driving distances, we expect to see a correlation. There are situations in which this is not the case. For example, if there is a barrier such as a canal between two nodes.

Example 4.1. Consider the situation where we have to solve a capacitated single depot multi-vehicle scheduling problem for milk pickup. Every three days, the dairy company will pick up milk at farms and return to the depot. The company has three milk trucks and must visit all farms. The total distance driven should be minimized.

We can easily visualize this problem by creating a node for each farm (nodes $1, 2, \ldots, 9$) and an extra node for the depot (node D) (See Figure 7). We assume that there exists a road between each pair of nodes and that the driving distance is related to the Euclidean distance between two nodes in this graph.

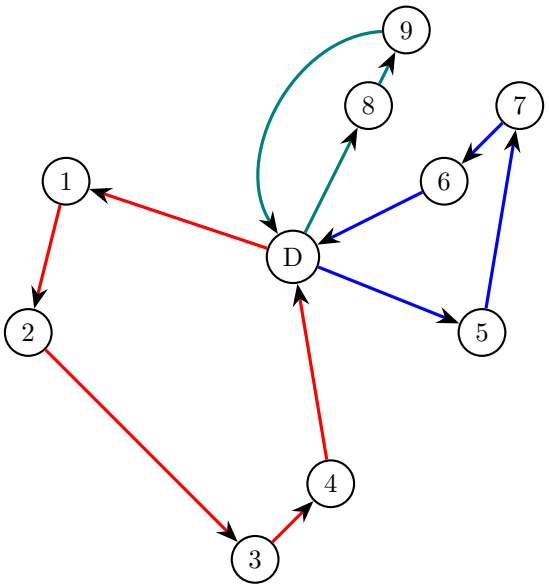

Figure 7a: Solution **A**

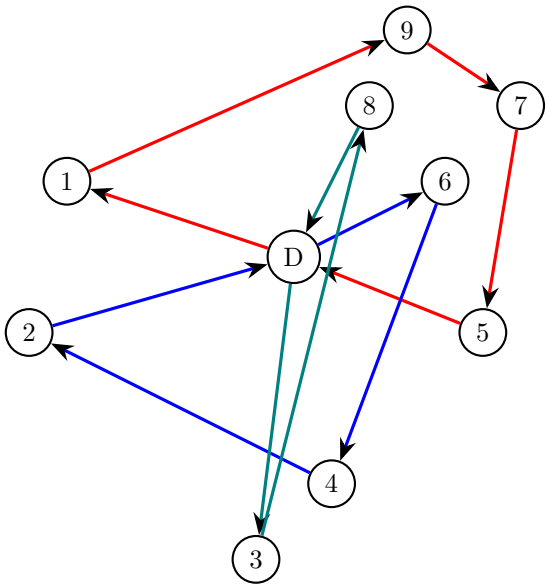

Figure 7b: Solution **B**

Two solutions for this problem are given in Figures 7a and 7b. Each set of colored arrows corresponds to a trip for one truck. Each trip starts and ends at the depot. When we compare these solutions it becomes clear that the total driving distance of solution **A** is significantly lower than the total driving distance of solution **B**.

For this example, we can say that having farms 3 and 4 in one trip is a good feature for a solution since both farms are close to each other and relatively far away from the other farms. Also, not having farms 3 and 8 in one trip could be a good feature since these farms are far apart and are not in the same region. Furthermore, consider farms 2 and 3, these farms are also far apart but are both in the southwest region of the graph. It could be a good feature to have both farms 2 and 3 in one trip because they are in the same region and they both have only a few other farms nearby. Of course, many more relations can be identified.

We are not able to create this kind of rules for significantly larger instances, so we will use an implicit way to identify whether two nodes should be together in one trip. Remark that although most good solutions include some specific features, it is still possible to have comparable or even better results without these features. ♢

To find good features, we need to have a dataset of good solutions. Since the case study for this thesis is the same that has been used for the 'Grote Opdracht' of the course 'Optimalisering en Complexiteit' we already have a good solver[7] in place. This solver is able to run at a speed of 3 million iterations per second[8]. We have done runs of 300 million, 3 billion, and 15 billion iterations to generate the following dataset:

Iterations	Number of runs	Average	Standard dev.	Best solution	Worst solution
300 million	3046	5492	29	5400	5595
3 billion	54	5409	19	5366	5470
15 billion	7	5379	17	5348	5409

Table 2: Good solutions dataset

Solver

The solver used to generate the dataset is programmed in C++ and can generate solutions of a certain number of iterations, store these solutions, and restart fully automatically to create new solutions. The solutions will be stored and can then be evaluated by another program verifying the cost and feasibility of the solutions. Between fifteen and twenty five percent of the simulations will be infeasible, due to capacity and/or time restrictions, this is dependant on the number of iterations and the penalties for capacity and/or time restrictions.

In our solver, almost all infeasible solutions are infeasible due to a relatively small penalty for going over the maximum capacity of the trucks. We have found that this small penalty results in significantly better solutions, compared to a situation where larger penalties were used, but at the cost of having some infeasible solutions. We have chosen not to keep simulating until a solution would become feasible. If we would have done this, the number of iterations is not equal for all simulations making it not possible to compare sets of simulations.

Using these solutions we have created a co-occurrence matrix for all nodes. We will use these co-occurrences to find *OrderVectors* which we will use to predict whether a neighbor will introduce a good feature.

[7] This solver was also used to generate the best solution found by a student team during the 2019 Utrecht University 'Optimalisering en Complexiteit' course.

[8] Tested on a desktop with an AMD Ryzen 5 2600 processor and 16GB of RAM.

Definition 4.1. Co-occurrence matrix For $i, j, n \in \mathbb{N}$ and $i, j \leq n$, a co-occurrence matrix for n nodes is a $n \times n$ symmetrical matrix where a_{ij} is equal to the number of the times that node j was in the context of node i in a given dataset. ♦

For our case study, we will use that nodes i and j are in each other's context if they are part of the same trip in a solution.

Example 4.2. Co-occurrence matrix Given four trips as seen in Figures 8a, 8b, 8c, and 8d we can create the co-occurrence matrix given in Table 3.

It is clear that node 1 is part of a trip if and only if node 2 is part of a trip. Assuming that all four trips represent good solutions we find that it is a good feature to have either both nodes 1 and 2 in a trip or to skip both nodes 1 and 2. Less clear is the connection between nodes 1, 4, and 5. If node 5 is part of a trip, then it is unlikely that nodes 1 and 4 are also part of that trip. Using *OrderVectors*, we will try to find these relations implicitly. ♢

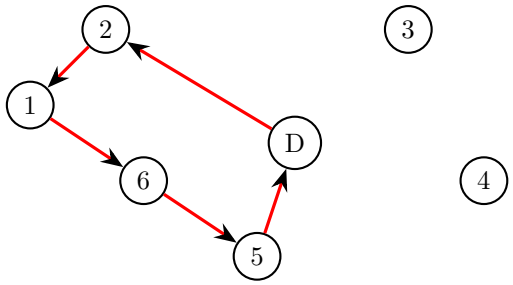

Figure 8a: Trip **A**

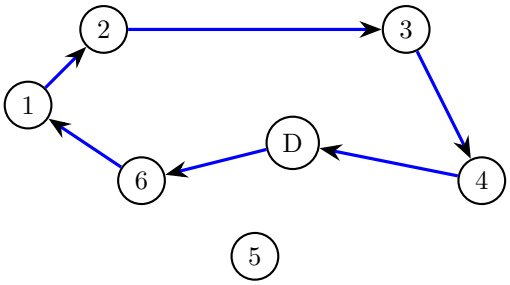

Figure 8b: Trip **B**

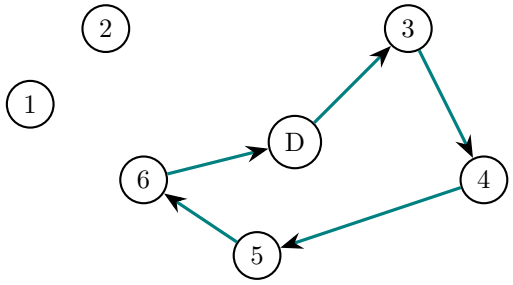

Figure 8c: Trip **C**

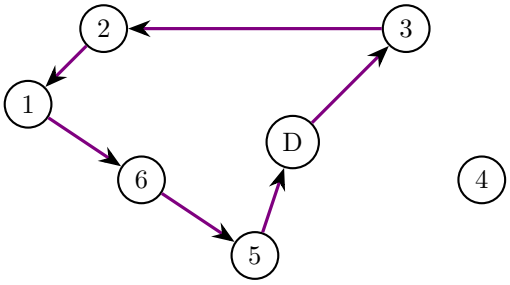

Figure 8d: Trip **D**

	node 1	node 2	node 3	node 4	node 5	node 6
node 1	×	3	2	1	2	3
node 2	3	×	2	1	2	3
node 3	2	2	×	2	2	3
node 4	1	1	2	×	1	2
node 5	2	2	2	1	×	3
node 6	3	3	3	2	3	×

Table 3: Co-occurrence matrix for trips **A**, **B**, **C**, and **D** in Figures 8 - 8d.

Since not all nodes are part of a trip each time, we introduce a weighted co-occurrence matrix. This will help find relations between orders that are not part of a trip frequently.

Definition 4.2. Weighted co-occurrence matrix For $i, j, n \in \mathbb{N}$ and $i, j \leq n$, a weighted co-occurrence matrix for n nodes is a $n \times n$ matrix. For this matrix, a_{ij} is equal to the number of times that node j was in the context of node i divided by the total number of times that node i was part of an item in the dataset. See Equation 1 in Section 4.2, in this equation P_{ij} is equal to entry of the weighted co-occurrence matrix. ♦

Remark that each value a_{ij} in the weighted co-occurrence matrix corresponds to the probability that orders i and j are part of a trip if order i is in a trip. We will discuss this probability further in Section 4.2.

Example 4.3. Weighted co-occurrence matrix For an example of a weighted co-occurrence matrix, we will use the co-occurrence matrix from Example 4.2, given in Table 3 and the trips from Figure 8. We have that node 4 is part of a trip two times, nodes $1, 2, 3, 5$ are part of trip three times, and node 6 is four times part of a trip. This gives the weighted co-occurrence matrix given in Table 4. Remark that by definition the co-occurrence matrix is symmetrical and that the weighted co-occurrence matrix can be symmetrical, but is not always symmetrical.

	node 1	node 2	node 3	node 4	node 5	node 6
node 1	$\times$	$\frac{3}{3}$	$\frac{2}{3}$	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{3}{3}$
node 2	$\frac{3}{3}$	$\times$	$\frac{2}{3}$	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{3}{3}$
node 3	$\frac{2}{3}$	$\frac{2}{3}$	$\times$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{3}{3}$
node 4	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{2}{2}$	$\times$	$\frac{1}{2}$	$\frac{2}{2}$
node 5	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{1}{3}$	$\times$	$\frac{3}{3}$
node 6	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{2}{4}$	$\frac{3}{4}$	$\times$

Table 4: Weighted co-occurrence matrix for trips **A**, **B**, **C**, and **D** in Figures 8 - 8d.

$\diamondsuit$

4.2 Machine Learning

We will use the GloVe model to predict whether the neighbor will have a positive effect on the cost of the solution in the long-term. The GloVe model was initially developed for applications in word representation. Each word in a corpus[9] is represented by a word vector. Using these word vectors, predictions can be made on the probability that two words will be used in each other's context. The context is defined as an area of a fixed distance around a specific word.

For each corpus a co-occurrence matrix is made where X_{ij} is equal to the number of times that word j occurs in the context of word i. Furthermore, X_i is equal to the number of times any word occurs in the context of word i, so

$$X_i = \sum_j X_{ij}.$$

Now, define for all i with X_i strictly greater then zero

$$P_{ij} = \mathbb{P}(j|i) = \frac{X_{ij}}{X_i}. \tag{1}$$

Remark that this probability distribution is well-defined, since

$$\sum_j \mathbb{P}(j|i) = \sum_j \frac{X_{ij}}{X_i} = \frac{\sum_j X_{ij}}{X_i} = \frac{X_i}{X_i} = 1$$

and

$$0 \leq \mathbb{P}(j|i) \leq 1.$$

So P_{ij} is equal to the probability that word j appears in the context of word i [5, 23]. Remark that P_{ij} is equal to the entry a_{ij} of the weighted co-occurrence matrix.

[9] A collection of written text. For example, all Wikipedia pages.

Example 4.4. As seen in Table 1 and Section 3 of [23], we can already get some information from these co-occurrence probabilities. Take for example the words *ice* and *steam*. We expect that the probability that *solid*, which is more related to *ice* than it is to *steam*, occurs more frequently in the context of *ice* than *steam*. So we expect the ratio

$$\frac{P_{solid,ice}}{P_{solid,steam}}$$

to be greater than one. We expect the opposite if we change *solid* for *gas*. For words that are either related to both *ice* and *steam*, like *water*, or words that are not related to one of them, like *fashion*, we expect that the ratio of co-occurrence probabilities is equal or close to one. See Table 5. Here we observe that the predicted behavior coincides with the co-occurrence probabilities. ♢

Probability and Ratio	k=*solid*	k=*gas*	k=*water*	k=*fashion*
$\mathbb{P}(k\|ice)$	$1.9 \cdot 10^{-4}$	$6.6 \cdot 10^{-5}$	$3.0 \cdot 10^{-3}$	$1.7 \cdot 10^{-5}$
$\mathbb{P}(k\|steam)$	$2.2 \cdot 10^{-5}$	$7.8 \cdot 10^{-4}$	$2.2 \cdot 10^{-3}$	$1.8 \cdot 10^{-5}$
$\mathbb{P}(k\|ice)/\mathbb{P}(k\|steam)$	8.9	$8.5 \cdot 10^{-2}$	1.36	0.96

Table 5: Co-occurrence probabilities and ratios for target words *ice* and *steam* [23]

.

Word vectors found by GloVe have dimensions of meaning. This means that the vectors capture analogies. An analogy is a comparison between two words. Analogies can be used to explain the meaning of a word using a familiar word. So for example "Berlin is to Germany as Paris is to France" or "Land is to River as Body is to Veins".

If we now take the vectors for each these words, $w_{Berlin}, w_{Germany}, w_{Paris}$ and w_{France}, then these vectors must satisfy $w_{Berlin} - w_{Germany} = w_{Paris} - w_{France}$, since if we remove the country from the two cities we expect to find that they are both capitals and have features that correspond to being a capital of a country. Another example is "bat is to ball as bow is to arrow", for this analogy we must have that that $w_{bat} - w_{ball} = w_{bow} - w_{arrow}$.
For words a, b, c and d and corresponding wordvectors w_a, w_b, w_c and w_d we have that if "a is to b as c is to d" these vectors must satisfy $w_a - w_b = w_c - w_d$ [23].

Pennington, Socher and Manning use this equation in [23] to find a missing word. If we want to know "'what' is to foot as fingers is to hand" we need to find the word vector that is closest to $w_{fingers} - w_{hand} + w_{foot}$. Remark that the closest vector in this case is the vector w that has the highest cosine similarity (See Definition 4.3) with $w_{fingers} - w_{hand} + w_{foot}$. We expect that this is the word vector w_{toes}.

To find these word vectors, we have to train the GloVe model [23]. We will train this model using AdaGrad, a stochastic gradient descent algorithm. We will explain stochastic gradient descent and AdaGrad in Sections 4.2.1 and 4.2.2. If we train the model using the co-occurrences X_{ij} from the word co-occurrence matrix, we have[10] that $w_i^T w_j + b_i + b_j = \log(X_{ij})$, in which b_i is an additional bias term. The bias terms are introduced to compensate for the total number of times (X_i) that some order is observed and allows us to 'shift' all terms in which a particular order is involved. We will train the model to find the word vectors and bias terms.

To find the vectors, we will optimize the following objective function, with d being the dimension of the co-occurrence matrix:

$$J = \sum_{i=1}^{d} \sum_{j=1}^{d} f(X_{ij}) \left(w_i^T w_j + b_i + b_j - \log(X_{ij}) \right)^2$$

This function is equal to the sum of squared errors with bias correction. The weight function f can be chosen freely as long as $f(0) = 0$, $f(X)$ is non-decreasing, and relatively small for large values

[10]See full derivation in Section 3 of [23].

of x. This last constrained is added to make sure that pairs of orders with high co-occurences are not overweighed. [5, 23].
In this thesis we will test two different weight functions,

$$f(x) = x$$

and (for different real x_{max} in $(0, 1]$)

$$f(x) = \begin{cases} \frac{x}{x_{max}} & \text{if } x < x_{max} \\ 1 & \text{else} \end{cases}$$

The first function is the most basic function that satisfies the constraints. The second function is proposed for the GloVe model [23].

4.2.1 Stochastic Gradient Descent

In order to define stochastic gradient descent (SGD), we first need to define batch gradient descent (the most basic gradient descent algorithm and the basis for stochastic gradient descent). Gradient descent is an algorithm used to optimize a minimization problem with a differentiable objective function. This function is also known as the error function and represents the sum of squared errors. For each data point, the distance between the value using the current parameters and the actual value is squared and summed. The algorithm will find a local minimum, and if there is only one extreme value, this is equal to the global minimum.

Gradient descent will update the decision variables by using the gradient of the objective function with respect to these variabels. Given the one-dimensional minimization problem with θ as decision variable, the gradient of the objective function will be equal to $\nabla_\theta J(\theta) = \frac{\mathrm{d}J(\theta)}{\mathrm{d}\theta}$.

Define θ_t as the value of θ at the tth iteration. Every step, the new value of θ, θ_{t+1}, will be equal to θ_t minus the gradient times the learning rate η. The learning rate determines how 'big' the steps that the gradient descent algorithm takes are to find the optimal value. So,

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t)$$

and if θ is one-dimensional, this is equal to

$$\theta_{t+1} = \theta_t - \eta \frac{\mathrm{d}J(\theta)}{\mathrm{d}\theta}$$

We can now define the gradient descent algorithm. First we have to define the objective function

$$J : \mathbb{R}^n \to \mathbb{R}_{\geq 0}$$

This function is equal to the sum of squared errors. Now that we have defined this function, we can take its gradient, defined by ∇J which is a real-valued function of θ. This gives Algorithm 2 [1, 25].

Algorithm 2 Gradient Descent

Create a new random vector θ.
while Fixed number of iterations has not been reached **do**
For each sample in the training set, update θ to

$$\theta - \eta \times \nabla J(\theta)$$

Example 4.5. Marketing and Sales Assume that we are interested in finding the relationship between marketing expenses and revenue. We have three datapoints from three comparable companies selling the same product at the same price. Company **A** spends 3 euro on marketing and has a revenue of 30 euro, company **B** spends 9 euro for a revenue of 42, and company **C** spends 10 for a total revenue of 48. We want to make a prediction using the formula:

$$\text{revenue} = \alpha \cdot \text{marketing expenses} + \text{base revenue}$$

Assume that there the *base revenue* is known and constant at 20 euros. Now, we are iterested in the value of α. This is the extra revenue in euros for each euro spend on marketing. First, we have to find the error function. This function is equal to

$$J(\alpha) = (30 - (\alpha \cdot 3 + 20))^2 + (42 - (\alpha \cdot 9 + 20))^2 + (48 - (\alpha \cdot 10 + 20))^2$$

and thus we have that

$$\nabla J(\alpha) = -6(30 - (\alpha \cdot 3 + 20)) - 18(42 - (\alpha \cdot 9 + 20)) - 20(48 - (\alpha \cdot 10 + 20))$$

which is the same as

$$\nabla J(\alpha) = 380\alpha - 1016$$

We will use a learning rate η equal to 0.005. We first guess that $\alpha = 3$, which gives Figure 9a. Now we have that the gradient is equal to $\nabla J(3) = 380 \cdot 3 - 1016 = 124$, we will update α to $3 - 0.005 \cdot 124 = 3 - 0.62 = 2.38$, which gives Figure 9b. For this new value of α we have that $\nabla J(2.38) = -111.6$ so we can now update α to 2.938 (See Figure 9c). We can keep updating α until we find the value for α where the gradient is equal to zero and thus where is sum of squared errors is minimal. In this case, the optimal value for α is equal to 2.67 and gives Figure 9d. Note that we have skipped some steps between Figure 9c and Figure 9d. Looking at these plots we see that the distance between the intermediate results and the optimal value converges to zero. In Figure 10, the value of $J(\alpha)$ is given for each value of α within a reasonable interval. In this chart, it becomes clear that at every step ($a, b, c,$ and d), the updated value of α resulted in a lower value for the objective function. ♢

Stochastic gradient descent Now that we have presented gradient descent we can define SGD. Since the gradient algorithm optimizes for every parameter of θ and uses every data point available, the algorithm is slow for large datasets. Also, in large datasets, there will be redundancies in the data and thus redundant calculations. SGD tackles this problem by selecting a random parameter and data point to calculate the value of the objective function and optimize correspondingly. Remark that this algorithm does need more iterations before it finds the optimal value. However, the iterations are much faster, so the algorithm will find the optimal value sooner compared to batch gradient descent.

In the dataset of Example 4.5, we have two data points that are quite comparable being the data from companies **B** and **C**. Here, we can train or model using one of those data points at each iteration without having to accept worse results.

This gives a modified version of Algorithm 2 in which only one data point is used at each iteration. Remark that for each iteration a new selection is made [1, 25, 33].

Algorithm 3 Stochastic Gradient Descent

Create a new random vector θ.
while Fixed number of iterations has not been reached **do**
Select a random sample of the training set and update θ to

$$\theta - \eta \times \nabla J(\theta)$$

Figure 9a: $\alpha = 3$

Figure 9b: $\alpha = 2.38$

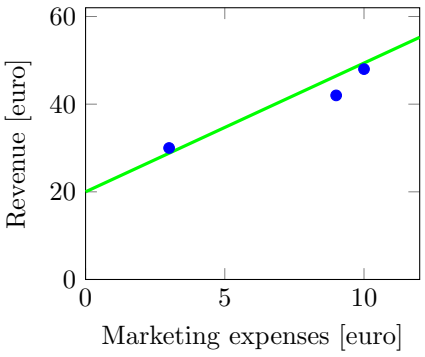

Figure 9c: $\alpha = 2.938$

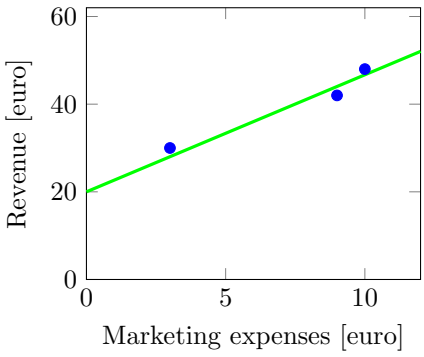

Figure 9d: $\alpha = 2.67$

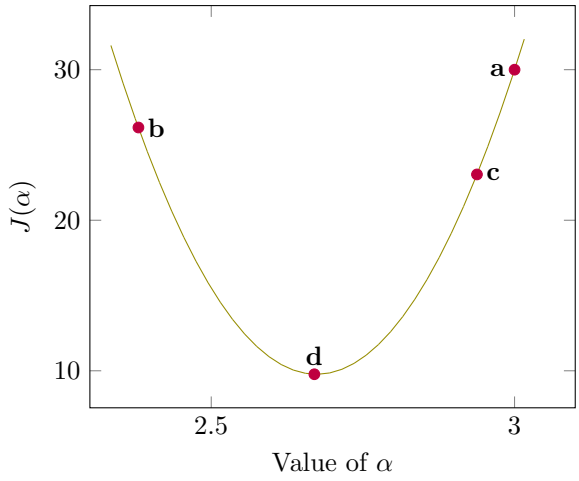

Figure 10: Development of sum of squared erros

Mini-batch gradient descent Another type of gradient is descent is mini-batch gradient descent. This algorithm is useful in situations where data points are grouped in sparse groups, meaning that there are groups of closely related data points where from an optimization standpoint, it is not relevant which variable is used in the training set. For example, companies **B** and **C** in Example 4.5. In mini-batch gradient descent, a small batch of data points is randomly chosen

to use for optimization. These data points are now used to do an iteration of the gradient descent algorithm. After this iteration has been completed, a new random batch is selected. This process is repeated for a set number of iterations [25].

4.2.2 AdaGrad

All gradient descent algorithms considered so far use a fixed learning rate for each iteration and each variable. We will now introduce AdaGrad. This is an **Ada**ptive **Grad**ient descent algorithm that changes the learning rate depending on the gradient. An adaptive gradient can be used for batch, mini-batch, and stochastic gradient descent. We will use it in the context of stochastic gradient descent.

Parameters for frequently occurring features will have low learning rates and parameters for infrequent occurring features will use a high learning rate. Intuitively, this means that the algorithm is able to find features that are predictive but that do have only a few occurrences [11].

AdaGrad has proven to be successful in object recognition and speech recognition applications. It was compared to other algorithms available in 2012 and showed better results on the ImageNet object recognition challenge [10]. ImageNet is a large-scale image database with over 14 million sorted images [18]. The ImageNet object recognition challenge is a way for researchers to compare their image recognition algorithms [19].

AdaGrad uses the history of gradients for each parameter to determine the learning rate η. First, we will define some variables and then we will give the modified SGD algorithm. Define for each parameter i and iteration t the gradient $g_{t,i} = \nabla_\theta J(\theta_{t,i})$. Now, we can define the sum of squared gradients for each parameter at iteration t as follows:

$$G_{t,i} = \sum_{n=1}^{t} \left(g_{n,i}\right)^2.$$

The learning step for AdaGrad is given by

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,i} + \epsilon}} \nabla_\theta J(\theta).$$

Here, ϵ is a very small number to avoid dividing by zero. Remark that compared to gradient descent we now divide the learning rate by the square root of the sum of squared gradients [11, 25].

AdaGrad works well with sparse data. This means that there are some entries in the co-occurrence matrix with relatively high values and a lot of entries with low values. This is the case in our case study, as seen in Table 6. For 71.6% of the entries[11] in our co-occurrence matrix, the number of observed co-occurrences is less than 301 (which is 10% of the dataset). The location of the orders and depot is comparable to the orders and depot in Figure 7. In good solutions, there will be groups of orders that are frequently together in one trip or that are only together in a trip in one or two solutions in the dataset.

Co-occurences	Observed
$0 - 50$	507434
$51 - 300$	484154
$301 - 2000$	380400
$2001 - 3000$	10338
3001+	1826

Table 6: Observed co-occurences

[11] Excluding the a_{ii} entries.

AdaGrad

For this thesis we use a C# implementation of AdaGrad. Our implementation will read in the co-occurrence matrix and run for a pre set number of iterations. The program runs on a single core and does not support parallelization. Depending on the vector length, the program will need between six (for 5-dimensional vectors) and nine (for 10-dimensional vectors) hours to complete.

4.3 OrderVectors

We will use *OrderVectors* for our case study. For each order i let $\mathcal{OV}_i$ be the *OrderVector* for order i. So instead of word vectors w_i for each word i we have *OrderVectors* $\mathcal{OV}_i$ for each order i. We have found these *OrderVectors* with AdaGrad and have trained the vectors so that for two orders i and j we have that $\mathcal{OV}_i^T \mathcal{OV}_j$ is a good approximation of the number of co-occurrences in good solutions (X_{ij}). Remark that we have used the same objective function as used for the word vectors.

With these *OrderVectors* we can calculate a prediction for each neighbor indicating the likelihood that a certain move will have a positive effect on the cost in the long term (See Example 4.6).

As mentioned previously, we can use the cosine similarity between two vectors to find pairs of vectors that are most alike. We can use this to predict whether a neighbor will have a positive effect on the cost of the solution in the long term.

Definition 4.3. (**Cosine Similarity**) When we compare two vectors of length n in a n-dimensional vector space we can calculate the cosine of the angle between the two vectors. For vectors v and w we do this by using the formula

$$\cos(v, w) = \frac{\langle v, w \rangle}{|v| \cdot |w|}$$

Remark that $\cos(v, w) \in [-1, 1]$ for all $v, w \in \mathbb{R}^n$. To express the cosine similarity between two orders, we use the formula

$$\text{sim}(v, w) = \frac{1 + \cos(v, w)}{2}$$

this way, we find an expression for the similarity of two orders in the interval $[0, 1]$. ♦

Now that we have the similarity between two orders, we want to use this in our case study. To do this, we developed the *Predictor*. For each neighborhood, we will create a specific *Predictor* with a procedure to calculate a prediction for a certain move. This is a prediction of how likely it is that the neighbor will have a positive effect on the cost. A prediction p is real-valued, with $p \in [0, 1]$. As input, the predictors will use the similarities between the *OrderVectors* for each node, an exploration distance, and a threshold. We will discuss those last two variables in the following example.

Example 4.6. Add order to trip In this example, we will give the intuition for the calculation of a prediction. Given the path in Figure 11a and the situation where we would like to check whether inserting order 4 between orders 3 and 5 would be a good move. First, we need to define the exploration distance.

The exploration distance is used since we are only interested in the orders close to the place where we want to add a new order. Recall Example 4.1 in which we pointed out some combinations of orders close to each other. The clusters can be combined with other clusters in a trip. In Figure 7 for example, orders 6 and 7 are likely to be in the same trip in a good solution. This trip can further contain orders 8 and 9 or order 5. If we want to add an order between 6 and 7, we are only interested in the probability that the new order and orders 6 and 7 are a good fit, independently of the rest of the orders in the trip.

Definition 4.4. (**Exploration distance**) For a trip of length n, $(a_1 \dots a_{i-2} a_{i-1} a_i a_{i+1} a_{i+2} \dots a_n)$ and a neighbor that changes this trip at position i, the exploration distance is equal to the number

of orders checked before and after the involved order. If the exploration distance is equal to two, orders $a_{i-2}, a_{i-1}, a_i, a_{i+1},$ and a_{i+2} and similarities between these orders will be used to make a prediction. ♦

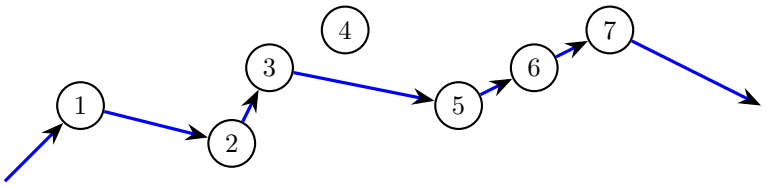

Figure 11a

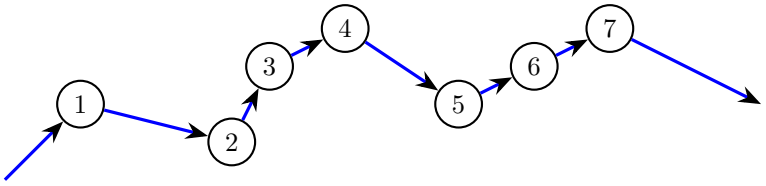

Figure 11b

We continue our example and will use an exploration distance equal to 2 in our algorithm. Hence, in our example we are only interested in the similarities between the *OrderVectors* of orders $2, 3, 5,$ and 6 and the *OrderVector* of order 4 since only these orders are within distance 2 of the order we would like to add.

Since the predictions are unlikely to reach values in the interval $[0.8, 1]$, we do scale the predictions using a threshold. This threshold will keep the predictions in $[0, 1]$ but will scale the lower predictions linearly and cap the predictions at 1.

Definition 4.5. (**Threshold**) The threshold is used to scale predictions. It is a real-valued constant in the interval $(0, 1]$. If a prediction is smaller than the threshold, the prediction will be divided by the threshold; if the prediction is greater than the threshold, it will be set to 1. This way, the prediction will stay in $[0, 1]$. ♦

In this example, we set the threshold to 0.9. This means that we will round up all predictions greater than 0.9 to 1 and increase all predictions smaller than 0.9 by about 11%. We do this to correct for the fact that in practice, in almost all cases the prediction will be less than 0.9 due to the fact that there are outliers in the dataset.

Since we would like that orders close to the change in the trip have a stronger influence on the prediction compared to orders that are further away, we introduce a weight function $w : \mathbb{N} \to \mathbb{R}$. Each similarity will be multiplied by this weight factor which depends on the distance from that order to the added/changed order. Different weight functions will be tested. Some examples are $w(d) = 1, w(d) = \frac{1}{d}$, and $w(d) = \frac{1}{\sqrt{d}}$. See Figure 12. We have chosen to use these functions since we are interested in the orders close to the update.

Now, the prediction before threshold correction is equal to

$$\text{Pred} = \frac{\text{sim}(2,4)w(2) + \text{sim}(3,4)w(1) + \text{sim}(5,4)w(1) + \text{sim}(6,4)w(2)}{2\sum_{n=1}^{\text{ex.dist.}} w(n)}$$

After this, if Pred is smaller then 0.9 we will divide it by 0.9 and set it to 1 otherwise.

♢

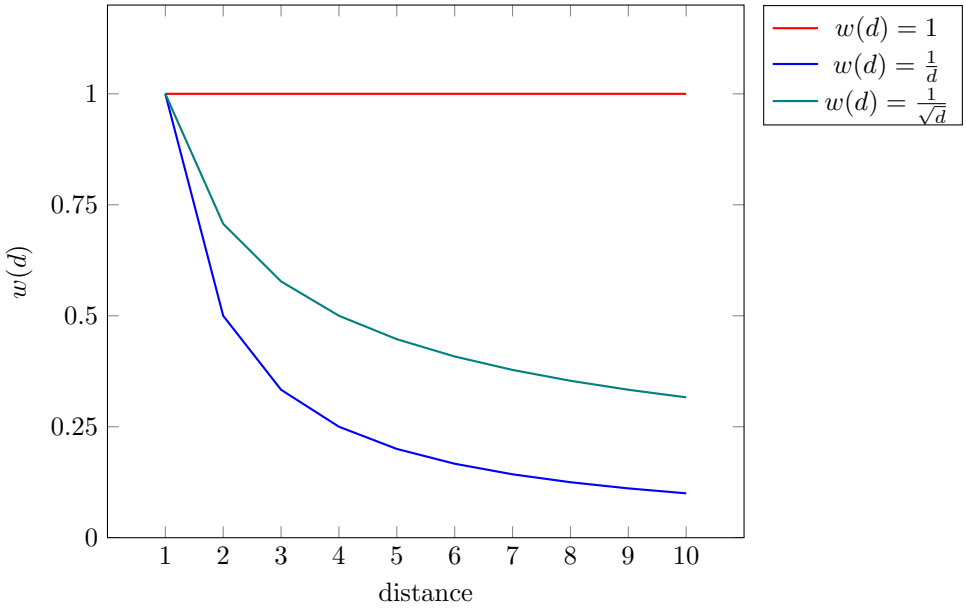

Figure 12: Distance weight functions

Furthermore, in this thesis, we will look at whether non-linear scaling functions can help improve the predictions. They have the same purpose as the threshold. By applying these functions to our predictions we hope to increase the prediction if it is already quite high (for example greater than 0.8) and to leave low predictions quite untouched. One of the functions that we will test is the function

$$s : [0, 1] \to [0, 1], x \mapsto \sin\left(\frac{x\pi}{2}\right) \tag{2}$$

Predictions that were initially around 0.8 are now closer to 0.95. Furthermore, we will test the function $x \mapsto x^2$ as a scaling function. This function will make sure that only neighbors with a high probability will have a higher T value compared to SA (See Figure 13).

4.3.1 Smart Simulated Annealing

With these *OrderVectors* and *Predictions* in place, we can formally define the SSA algorithm. This algorithm is a modified version of Algorithm 1 and is given in Algorithm 4. Remark that we have added lines 7 and 8. Here, we calculate and use the prediction made using the *OrderVectors*.
We can see the effect of using T_{temp} in Figure 14. If $p(\mathcal{P}) = 0.0$, we predict that a 'worsening' step won't have a positive effect on the cost in the long term. In that case, we decrease the probability that we will accept this 'worsening' step compared to the standard SA acceptance probability given by $p(\mathcal{P}) = 0.5$. However, if we expect that a 'worsening' step will improve the cost in the long term, we have for example $p(\mathcal{P}) = 1.0$ and then, we see that the probability that we accept this step is greater than with the SA algorithm.

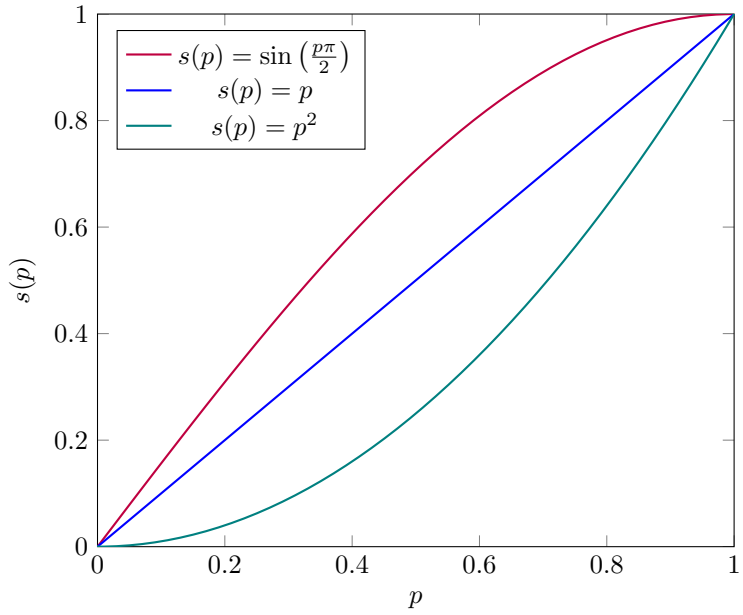

Figure 13: Scaling functions

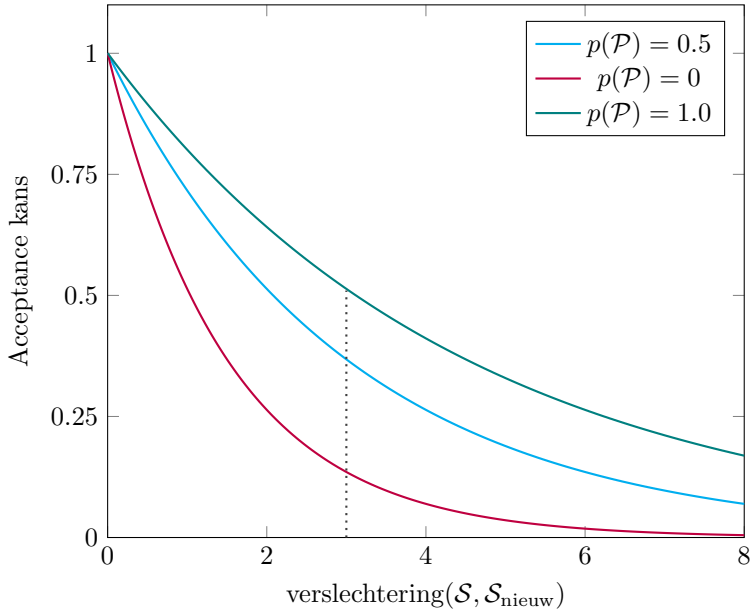

Figure 14: Effect of different predictions for $T = 3$.

Example 4.7. Consider the case where we have the trip presented in Figure 15 and have a neighbor where we add order 4 to this trip. We assume that this is a good feature. If, however, we add order 4 between the start point and order 3, we get the trip presented in Figure 16. This is not an optimal trip. By accepting this neighbor, we can rearrange the orders within the trip and get the trip presented in Figure 17. It is clear that by accepting a neighbor that had a negative effect on the cost we can get a better solution in the long term by rearranging the orders within the trip. ♢

Algorithm 4 Smart Simulated Annealing algorithm

1: Create a new starting solution $\mathcal{S}$
2: Set $\mathcal{S}_{\text{best}}$ equal to $\mathcal{S}$
3: **while** Fixed number of iterations has not been reached **do**
4: If a fixed fraction of the number of iterations has passed, multiply T by α.
5: Select a random neighbor n.
6: Compute the cost with the new neighbor, the new cost is equal to $\text{cost}(\mathcal{S}_{\text{new}})$.
7: Create a predictor $\mathcal{P}$ for n, calculate the prediction $p(\mathcal{P})$.
8: Calculate T_{temp} which is equal to T multiplied by $p(\mathcal{P}) + 0.5$.
9: If the new cost is less than the old cost, accept the change. If the new cost is not less than the old cost, select a random number $\mathcal{R}$ between 0 and 1. Now accept the change if

$$\exp\left\{\frac{\text{cost}(\mathcal{S}) - \text{cost}(\mathcal{S}_{\text{new}})}{T_{temp}}\right\} \geq \mathcal{R}$$

and reject otherwise.
10: If the change is accepted, set $\mathcal{S}$ is $\mathcal{S}_{\text{new}}$.
11: If $\text{cost}(\mathcal{S})$ is smaller than $\text{cost}(\mathcal{S}_{\text{best}})$, set $\mathcal{S}_{\text{best}}$ is $\mathcal{S}$
12: When finished, save $\mathcal{S}_{\text{best}}$.

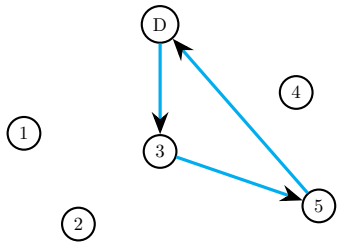

Figure 15: Start trip

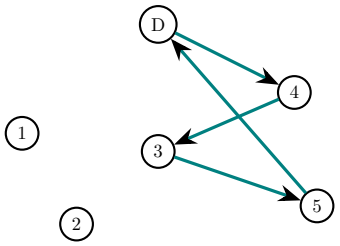

Figure 16: Add order 4.

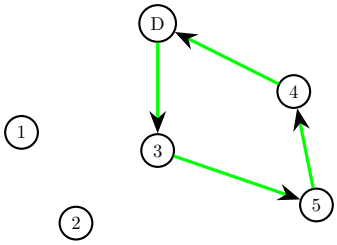

Figure 17: Improved order within trip of Figure 16

5 Experiments

In this section, we will give a deeper insight into the SA Solver, go over some other approaches we have used, and finally present the results of the SSA algorithm.

5.1 SA Solver

For the base experiment, we used the solver that was also used for generating the dataset used to get the *OrderVectors*. This solver functioned as the starting point for the solver that used SSA. The neighborhoods that we will present are the same used for all solvers. Without proof in this thesis[12] we will state that the optimal values of α and T for the SA Solver are given by $\alpha = 0.9992$ and $T = 20$. We have used the following neighborhoods:

`AddOrder` This neighborhood adds an order to a randomly selected point in a randomly selected trip. If an order has a frequency not equal to one, this neighborhood will take that into account and add the order following a selected pattern. Remark that a pattern prescribes on which days the order must be fulfilled (See Table 1). Within these days, the order is placed in a randomly selected trip and at a randomly selected spot within that trip.

`RemoveOrder` All neighbors in this neighborhood will remove exactly one order from a trip. If the frequency of this order is not equal to one, the order will be removed in all trips of which this order is a part. This way, we ensure that there are no partially fulfilled orders.

`AddWaste` This neighborhood will add a visit to a landfill between two orders in a trip, this trip is then split into two trips. A visit to the landfill cannot be added at the beginning or at the end of a trip. We add this last constraint since we assume that each trip starts and ends at the landfill.

`RemoveWaste` This neighborhood will combine two trips by removing a visit to the landfill.

`LocalPush` This neighborhood is given as 'Push' in Section 3.3.

`LocalPushTruck` This neighborhood is equal to `LocalPush` except that now, the order will be moved from a trip on one truck to a trip of another truck on the same day.

`LocalSwap` This neighborhood is given as 'Swap' in Section 3.3.

`LocalPushTruck` This neighborhood is equal to `LocalSwap` except that now, an order will be swapped with an order from a trip of the other truck on the same day.

`PermuteOrder` With this neighborhood, the neighbors will change the frequency pattern of an order. This means that if the frequency of an order is for example 2 and the order was fulfilled on Tuesday and Friday, we will change this to Monday and Thursday.

To generate a starting solution, we start with an empty solution, i.e., all orders are declined, then we will for each order apply the `AddOrder` neighborhood. So we will add each order to the solution and no orders are declined after generating the starting solution.

During the SA process, we do allow violations of the hard constraints. This way, the algorithm has more freedom to find a good solution. However, since we are only interested in feasible solutions, we encourage the algorithm to find feasible solutions. We do this by adding extra penalties to the cost function. For the different types of violations, we add an extra cost to the total cost, if a neighbor reduces the infeasibility of the solution, these penalty costs will decrease and this will encourage the algorithm to find a good solution, while, if necessary, these constraints can be violated. In the SA solver, we have two violations and penalties.

[12] These values were found during many experiments for the 'Optimalisering and Complexiteit' course for which this solver was initially developed.

The first is the time constraint, if for one truck, the total time for all trips on a certain day is longer than $t_{\max}$ (12 hours in our case), we will add a penalty. The penalty is given by

$$\text{penalty}_{time}(t) = 70.5 \left(\frac{t - t_{\max}}{t_{\max}} \right).$$

The other violation that we allow at a cost, is going over waste capacity. This is the case if a truck collects more waste on a trip than the capacity of the truck allows. Remark that the time penalty is for one truck and a whole day, where the waste penalty is for a single trip. In order to define the waste penalty, we first have to define the function $\text{waste}_{\text{over}}$, this function is given by

$$\text{waste}_{\text{over}}(w) = \frac{w - w_{\max}}{w_{\max}}.$$

Here, $w_{\max} = 20.000$. The penalty for going over capacity is given by

$$\text{penalty}_{waste}(w) = \begin{cases} 188 \cdot \text{waste}_{\text{over}}(w) & \text{if } \text{waste}_{\text{over}}(w) < 0.2 \\ 564 \cdot \text{waste}_{\text{over}}(w) & \text{else} \end{cases}$$

To calculate the total cost of a solution, we sum the driving, emptying, and dumping times. We add the penalties for the declined orders (recall that this is three times the emptying time times the frequency) and then we add the extra penalties. These are the time violation penalties penalty_{time} for each day in the week and each truck and the waste violation penalties penalty_{waste} for each trip. We will add these penalties in order to encourage the algorithm to find a good solution while giving it the freedom to violate the hard constraints. These penalties with the same constants were also used for all other solvers described below.

With this solver, we have generated the dataset given in Table 2. In an additional experiment, we have also saved the intermediate costs at 0, 10, ..., and 90 percent of the total number of iterations. This experiment consisted of 100 runs of which 76 were feasible solutions. This data is presented in Figures 18 and 19.

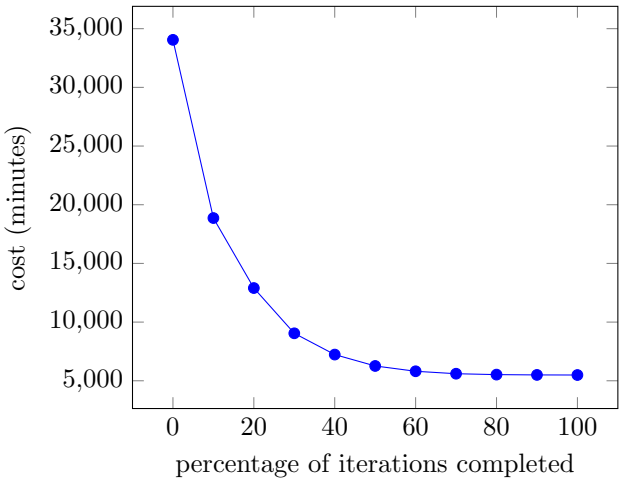

Figure 18: Development of the cost in the base experiment

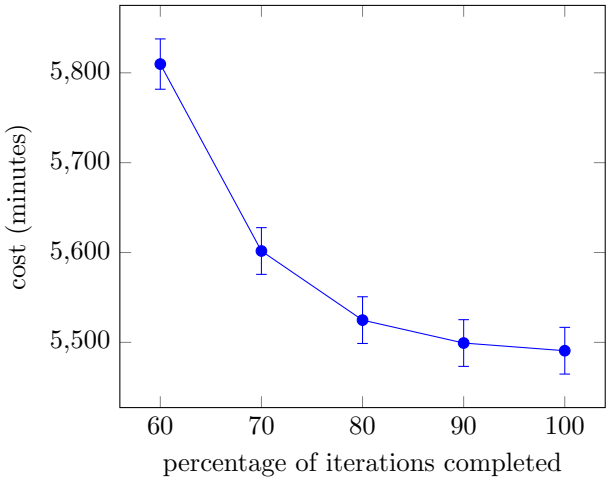

Figure 19: Development of the cost in the base experiment. One standard deviation is presented.

5.2 Other approaches

Subcost-guided approach

The subcost-guided simulated annealing algorithm is introduced by Wright ([32]) and is proven to be successful for School timetabling problems. The subcost-guided approach will adjust the temperature to increase the probability that 'worsening' steps for the total cost are accepted if they have a good effect on a subproblem. Each of these subproblems will have its own cost function, the subcost. In the school timetabling problem, a subcost can be defined by the number of double bookings a teacher or class has.

School Timetabling Problem

In school timetabling problems, teachers and classes should be matched. Each class should get a fixed and predetermined number of lessons taught from a set of departments, within each department, there is a set of teachers. The teachers are interchangeable as long as each class has only one teacher per department. Each class can only have one lesson at the time and teachers can only teach one lesson per time block. When we solve this problem, we will allow overlapping lessons at a cost. This way, we can use local search to find a solution [32].

The subcost-guided simulated annealing algorithm is identical to Algorithm 1 except for line 7. In this line, we change the acceptance test. Let $\mathcal{B}$ be the maximum decrease of all subcosts. If $\mathcal{B}$ is greater than zero, the new acceptance test is given by:

$$\exp\left\{\frac{\big(\text{cost}(\mathcal{S}) - \text{cost}(\mathcal{S}_{\text{new}})\big)\exp\left\{\left(\frac{-\theta\mathcal{B}}{\text{cost}(\mathcal{S})-\text{cost}(\mathcal{S}_{\text{new}})}\right)\right\}}{T}\right\} \geq \mathcal{R}$$

If $\mathcal{B}$ is negative, we will reject the neighbor.

In his paper, Wright uses different values for θ and finds that values between zero and ten can have positive results, depending on the exact problem used. In later work, he shows that this subcost-guided approach also can improve solutions for Sports timetabling and Flowshop problems [31].

Flow Shop

In the Flow shop machine scheduling problem, we have n jobs and m machines. Each job consists of exactly m ordered operations, we must process each i-th operation of each job on the i-th machine. For each job i the processing time of operation j is given by t_{ij}. We must execute the operations in the given order, which is the same for all jobs. We are free to choose the order in which we execute the jobs. The objective is to make a planning for all the machines and operations so that the time between starting with the operations and completing all operations, the makespan, is minimized. Each machine can only work on one operation at the time and work on the i-th operation of each job can only start after operation $i-1$ is completed [27].
A good example of these kinds of problems are assembly lines. For example a car factory; the doors can only be fitted after the chassis is completed, and while all cars look the same, each model has its characteristics which make that the time to finish the different operations can vary.

We have tested this approach for our case study. We defined the driving costs for each trip (including penalties for capacity) as subcosts, so the number of subcosts is equal to the number of trips. We have tested this with different values for θ, with 50 runs for each experiment. The exact results can be found in Table 7.

θ	0	0.25	0.5	1.5	3
Average	5490.12	5558.85	5569.58	5589.23	5567.25
Standard deviation	29	32	34	37	37

Table 7: Test results from the subcost-guided approach

We anticipated that the subcosts of our case study were independent enough to benefit from this approach. In the end, we found that this was not the case. We attribute this to the fact that most of the neighborhoods used in our case study involves moving orders from one trip to another or removing orders from a trip which increases the overall decline penalty. By decreasing a subcost, we would most definitely increase another subcost. This means that a subcost guided approach, which depends on the fact that a decrease in one subcost will in the long run also decrease another subcost, does not work for our case study.

TruckAppointer

The second approach we tested is one where we used the *OrderVectors* in a different way compared to SSA. In SSA, we use the vectors after we have selected a single neighbor and determine if this neighbor is likely to have a positive effect on the cost. With *TruckAppointers* we select a neighbor from a randomly selected subset of neighbors, using the *OrderVectors*.

In the situation where we make use of the *TruckAppointer*, we provide the *TruckAppointer*-object with multiple neighbors from the same neighborhood. Those neighbors are related in the sense that there is only one variable changed between the different options, so for example, the order we add in the `AddOrder` neighborhood stays the same. If we want to add order q to a trip on a certain day, we can add this order to either truck 1 or truck 2. Also, multiple options for each truck are considered. The *TruckAppointer* will return the neighbor of which the probability that it improves the cost in the long term is the highest. To make these predictions it uses the same *Predictors* as used in the SSA algorithm. With the *TruckAppointer* approach, we would make predictions for both our trucks to see for which truck the order is the best fit. Hence the name *TruckAppointer*. We will then add the order to this truck and provide the neighbor to the SA algorithm.

We discovered that the results of this approach were comparable to the results found by SA. We decided not to move forward with this approach since the running time of the algorithm was slowed down twelvefold by the additional calculations needed for the *TruckAppointer*.

Also, since we did not change the SA algorithm, we found that, if a neighbor would have a positive effect in the long term, it still was not accepted if it increased the cost too much. This became a larger problem in the second part of the runs where T became smaller. In this stage of the annealing process, the maximum increase in cost is very low. A solution to this could be to integrate the *TruckAppointer* into the SSA algorithm, but since we have seen that the *TruckAppointer* approach slows the program down considerably, we decided not to use this combination.

5.3 Smart Simulated Annealing

OrderVectors With the SSA solver, we have found promising results. We have found a configuration that is capable of finding comparable results compared to the SA solver in fewer iterations.

We have used two sets of vectors. In the first set, we used the co-occurrences between orders; in the second set, we used the co-occurrences between the matrix-IDs of the orders. These IDs are identifiers for the location of the order in the distance matrix. In the last, we decoupled the frequency and amount of garbage from the location of the order and used the latter to find the *OrderVectors*. The co-occurrences found using this approach differ from the co-occurrences between the orders since there multiple sets of orders that use the same matrix-ID. Remark that two vectors co-occur in a solution if they are part of the same trip.

For the *OrderVectors*, we have used two ways to find the vectors. For the first set of vectors (which we will call V_1), we used the co-occurrences between the orders to find the vectors. For the second set (V_2), we use the co-occurrences between the matrix-IDs of the orders. We have used our own AdaGrad implementation to find the vectors. We ran the algorithm for 150 million iterations. We set $\eta = 0.001$ and we found that the optimal weight function f, is given by

$$f(x) = x.$$

Remark that it takes about 6 (for 5-dimensional vectors) up to 9 hours (for 10-dimensional vectors) to find the vectors. We found that 15-dimensional vectors gave the best results.

We have found that optimal results are found using $T = 20$ and $\alpha = 0.9993$. Remark that this value for α is bigger than the one used in the SA solver. We did this in order to increase the effectiveness of the predictions, especially in the last half of a run (See Figures 20 and 21).

Short runs In the short runs, we found that the V_2 vectors gave better results compared to the V_1 vectors. We tested the different scaling function described earlier in Section 4.3. We found that the best scaling function in our case is the function given by

$$s(p) = p^2.$$

For the distance weight function, we will use

$$w(d) = \frac{1}{d}$$

in combination with $d_{exp} = 4$ and threshold= 1. Furthermore, as mentioned earlier, we will use $T = 20$ minutes and $\alpha = 0.9993$. We present the results in Table 8.

Solver	Runs	Iterations	Feasible	Average	Standard deviation
SA solver	100	300 million	80	5491.85	28
SSA solver, V_1	50	280 million	7	5491.20	23
SSA solver, V_2	100	270 million	74	5492.13	30

Table 8: Test results long runs with different solvers.

In Figures 22 and 23 we have visualized the development of the cost over time. We find that for short runs, the number of iterations needed compared to the SA solver is 7 to 10 percent less with the SSA Solver.

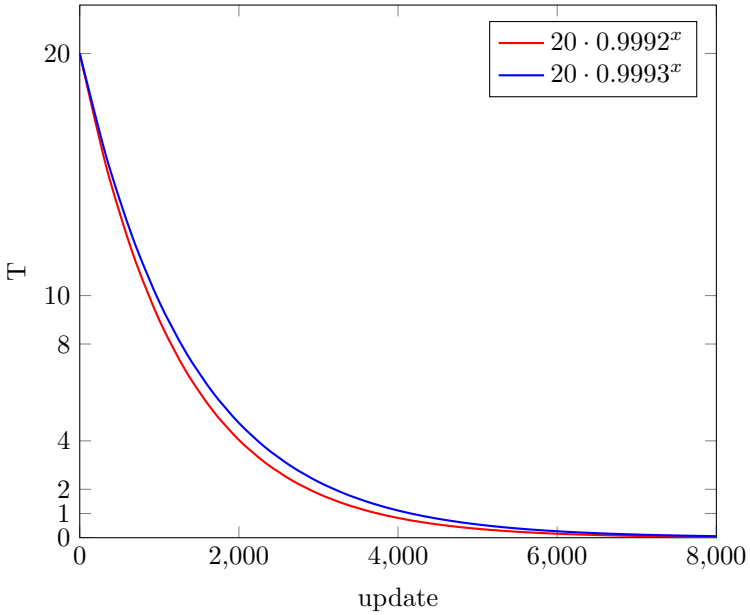

Figure 20: Comparison of temperature over time, for $\alpha = 0.9992$ and $\alpha = 0.9993$.

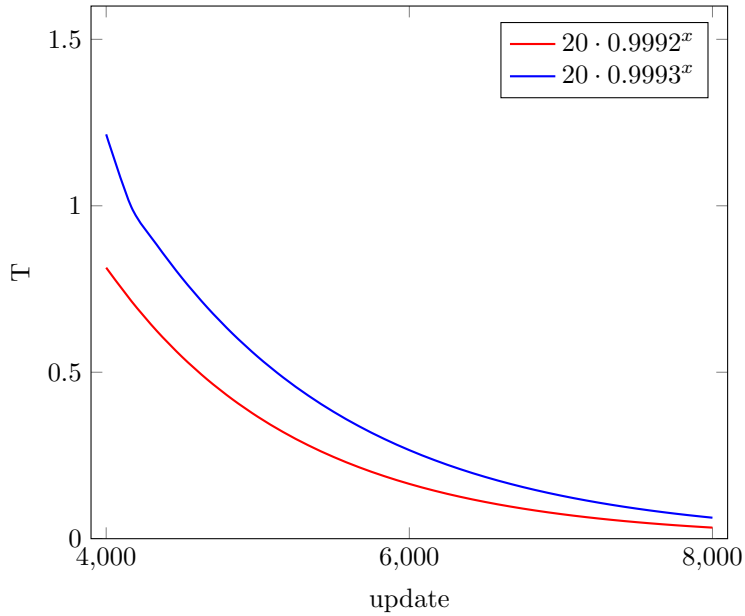

Figure 21: Comparison of temperature over time, for $\alpha = 0.9992$ and $\alpha = 0.9993$.

Long runs For long runs, we found even better results. In line with the short runs showed the runs using the V_2 vectors better results compared to the runs using the V_1 vectors. We will compare our results to the results found in the dataset. To recall, the SA Solver needed 15 billion iterations to find solutions with an average cost of 5378.74 minutes and a standard deviation of 17 minutes. In total, 7 runs were feasible.

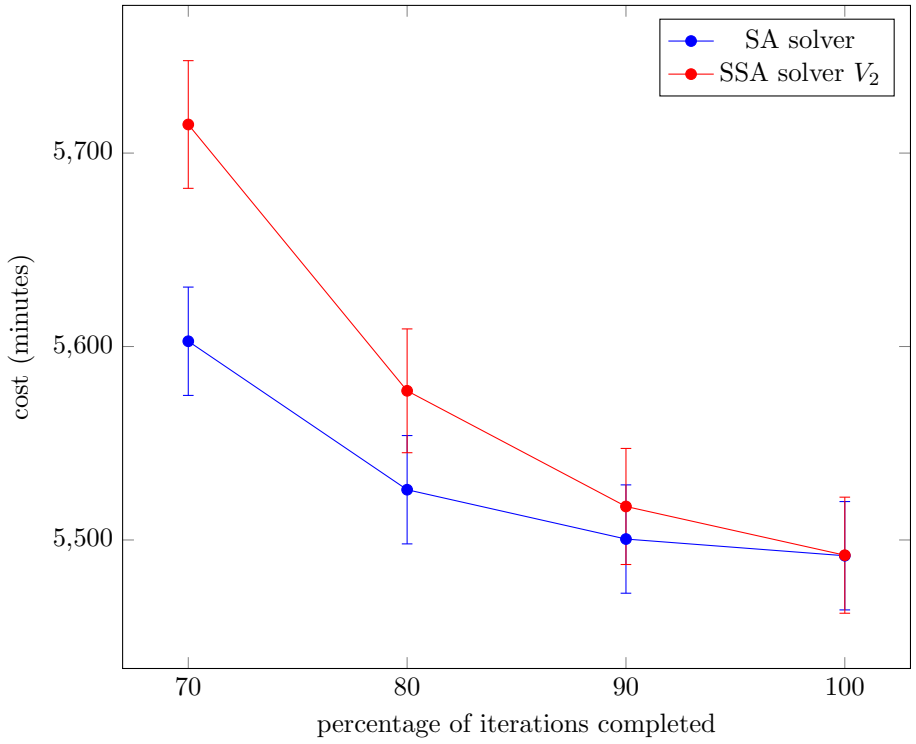

Figure 22: Development of the cost in short runs. One standard deviation is presented.

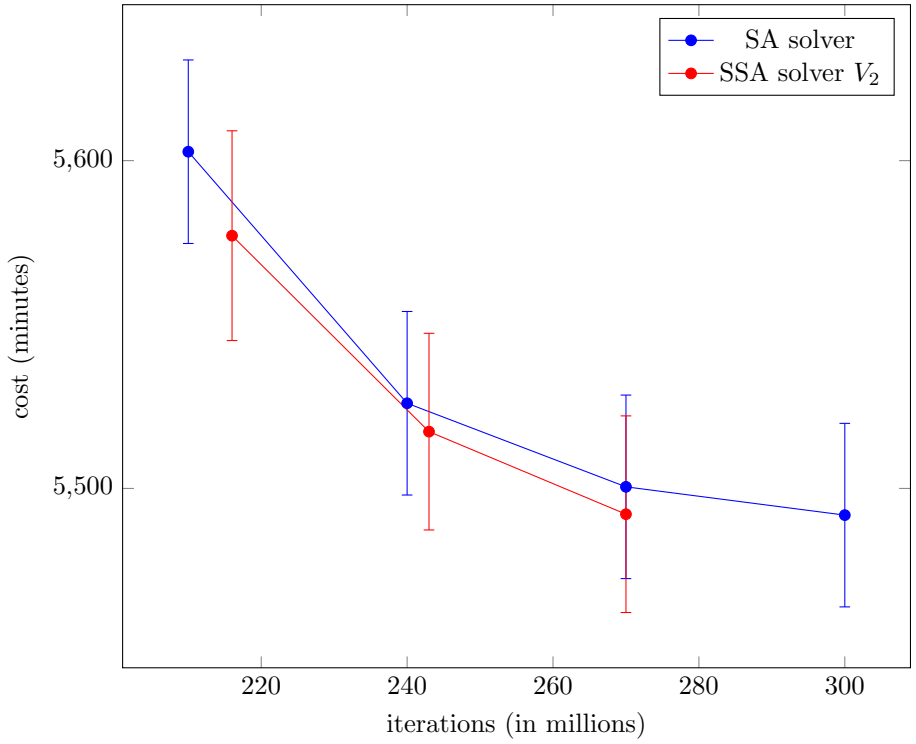

Figure 23: Development of the cost in short runs. One standard deviation is presented.

For the runs that we did with the V_1 and V_2 vectors we used the following parameters

$$T = 20 \qquad \alpha = 0.9993 \qquad s(p) = p^2$$
$$w(d) = \frac{1}{d} \qquad d_{exp} = 4 \qquad \text{threshold} = 1$$

In the runs with the V_1 vectors, we did 14 billion iterations (7% less) and in the runs with the V_2 vectors, we did 12.5 billion iterations (17% less) (see Table 9 for a summary of the results).

Solver	Runs	Iterations	Feasible	Average	Standard deviation
SA solver	10	15 billion	7	5378.74	17
SSA solver, V_1	9	14 billion	7	5378.77	17
SSA solver, V_2	10	12.5 billion	9	5386.94	14

Table 9: Test results long runs with different solvers.

For the last runs with the SSA Solver, we have plotted the score for the last 30% iterations in Figure 24. We find that for long runs, the number of iterations needed compared to the SA solver is 7 to 17 percent less with the SSA Solver.

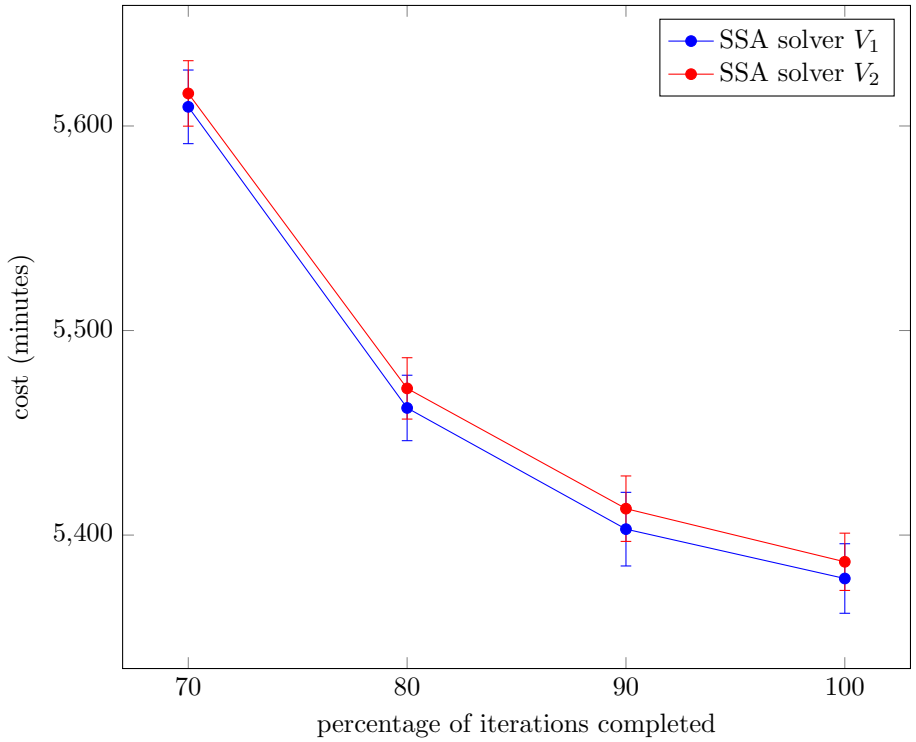

Figure 24: Development of the cost in long runs with the SSA Solver. One standard deviation is presented.

6 Discussion

In this thesis, we found that we can find comparable results with the SSA algorithm compared to the SA solver with 7 to 17 percent fewer iterations. Furthermore, we believe that there is room to improve the SSA algorithm to improve even further. While these percentages may sound like a small difference, in a world where costs are high and margins are small, like the parcel business, even small differences can have a big effect. The total revenue of the two parcel companies in the world, UPS and FedEx is over 130 billion US dollar [13, 28].

Remark that if we would change the problem by removing and adding some orders, we can now still find a high-quality solution more easily compared to SA. With the V_1-vectors, we trained the GloVe model to find vectors for each order. We will have to retrain the model using new data points found by the solutions generated with the new orders. We can add these new results to the cost function of AdaGrad and use the 'old' vectors as a starting point. In the case where we used the co-occurrences between the Matrix IDs, which we used to find the V_2-vectors, we can add new orders much more easily if the model has already been trained with orders that have the same Matrix ID. We can create instances of the capacitated vehicle routing problem where we use that (parts of a) street or small residential/commercial areas have the same Matrix ID.

One of the great features is that if we have to retrain the model we do not necessarily have to start from the beginning. We can just add the new data point to the dataset and its error to the objective function of the SGD algorithm and do additional iterations starting with the existing vectors.

One of the advantages of using the vectors trained using the Matrix ID co-occurrences is that this approach is frequency-independent. The vectors we are interested in for such a problem are based on how frequently the different locations occur in each other's context.

Remark that in our experiment, we have let the penalties for going over time or picking up too much garbage untouched, we did this to keep the number of different variables within range. The current penalties have been optimized for the 'basic' SA solver. It could be the case that these penalties are not optimal for the SSA algorithm.

Future research As suggested above we would like to point out that the SSA algorithm used currently and its implementation can be optimized. We believe that, especially for longer runs, it could be beneficial to make more pre-computations in order to increase the iterations per second. Also, we have tested many combinations of parameters, but since there are many combinations, there could be a combination of variables that will give better results.

Since we only tested our algorithm with one dataset it would be useful to test the effect of changing some variables. As an example, the capacity of the truck can be changed. Other variables include the driving times and emptying times. We expect that we do not have to retrain the model completely since we can use the co-occurrences of the locations and if the locations don't change we expect that the influence of a larger or smaller capacity on the effect of SSA is small. We noticed that for most good solutions each truck did one or two trips per day. However, if the driving times change considerably, the vectors should be retrained using a new set of good solutions since we do not know whether the solutions are still considered good under the new driving times.

An important area that we did not experiment with is the robustness of our solutions. With our case study, we assumed that the driving times and garbage to be picked up are constant. In real life this is of course far but the truth. We would suggest to run simulations with normally distributed driving times (possibly taking into account time of day) and garbage amounts. With the data found, one could create a robustness function and add this to the objective function in order to maximize robustness.

Concluding remarks In a changing economy, the need for high-quality vehicle scheduling solutions increases. In this thesis, we have presented a way to integrate machine learning into Local Search. With this new algorithm, we were able to reduce the number of iterations needed for comparable solutions by 7 to 17 percent.

With this improvement, we have shown that it is beneficial to integrate machine learning into a vehicle scheduling process. Also, this algorithm is more future proof since new orders can be added more easily. In the approach where we have used *OrderVectors* based on the location of the orders, we can add new orders to existing locations without the need to retrain the model. These vectors also showed to give the best results.

7 Acknowledgements

First and foremost, I would like to thank my supervisor, dr. J.A. Hoogeveen, for his valuable help throughout the process of writing this thesis. During the research phase, he challenged me to get the best possible results. Furthermore, he gave great feedback during the writing process. Without his help, this thesis would not have been possible.

I would also like to thank dr. A.J. Feelders for his help to understand the GloVe model.

In addition, I am grateful to Pieter Knops for his great feedback on earlier versions of this thesis. His feedback was essential to making this thesis as clear as it is now. Last but not least, I would like to thank my fellow students Shashi Chotkan and Xavier de Bondt. During the course 'Optimalisering en Complexiteit' we worked together on the SA Solver and this solver was elemental for this thesis.

Finally, but definitely not less importantly, I am thankful to my family for their moral support.

References

[1] L. Bottou. Large-scale machine learning with stochastic gradient descent. *Proceedings of COMPSTAT'2010*. Springer, (2010), pages 177–186.

[2] P. Bowes. *Pitney Bowes Parcel Shipping Index*. (2020). URL: `https://www.pitneybowes.com/us/shipping-index.html#` (visited on 06/02/2020).

[3] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, (2004).

[4] K. Braekers, K. Ramaekers, and I. V. Nieuwenhuyse. The vehicle routing problem: State of the art classification and review. *Computers & Industrial Engineering* 99 (2016), pages 300–313.

[5] R. Brochier, A. Guille, and J. Velcin. Global Vectors for Node Representations. *The World Wide Web Conference*. WWW '19. San Francisco, CA, USA: Association for Computing Machinery, (2019), pages 2587–2593.

[6] C. Chou, R. Han, S. Li, and T.-K. Lee. Guided simulated annealing method for optimization problems. *Physical Review E* 67 (2003), pages 066704.

[7] A. Connor and K. Shea. A comparison of semi-deterministic and stochastic search techniques. *I.C. Parmee Evolutionary Design and Manufacture (Selected Papers from ACDM'00)*. Springer-Verlag: London, (2000), pages 287–298.

[8] S. Cook. The P versus NP problem. (2006), pages 87–104.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms. MIT press, (2009), pages 1049–1069.

[10] J. Dean, G. Corrado, R. Monga, K. Chen, M. D., M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. Le, and A. Ng. Large Scale Distributed Deep Networks. *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., (2012), pages 1223–1231.

[11] J. Duchi, E. Hazan, and Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research* 12.61 (2011), pages 2121–2159.

[12] B. Eksioglu, A. Vural, and A. Reisman. The vehicle routing problem: A taxonomic review. *Computers & Industrial Engineering* 57.4 (2009), pages 1472–1483.

[13] FedEx. *FedEx Annual Report 2019*. (2020). URL: `https://investors.fedex.com/financial-information/annual-reports/default.aspx` (visited on 05/22/2020).

[14] F. Glover. Tabu search—part I. *ORSA Journal on computing* 1 (1989), pages 190–206.

[15] F. Glover. Tabu search—part II. *ORSA Journal on computing* 2 (1990), pages 4–32.

[16] C. Groër, B. Golden, and E. Wasil. A library of local search heuristics for the vehicle routing problem. *Mathematical Programming Computation* 2.2 (2010), pages 79–101.

[17] K. Hoffman, M. Padberg, and G. Rinaldi. Traveling salesman problem. *Encyclopedia of operations research and management science* 1 (2013), pages 1573–1578.

[18] ImageNet. *About ImageNet*. (2016). URL: `http://image-net.org/about-overview` (visited on 05/19/2020).

[19] ImageNet. *Large Scale Visual Recognition Challenge (ILSVRC)*. (2015). URL: `http://www.image-net.org/challenges/LSVRC/` (visited on 05/19/2020).

[20] J. Lysgaard, A. Letchford, and R. Eglese. A new branch-and-cut algorithm for the capacitated vehicle routing problem. *Mathematical Programming* 100.2 (2004), pages 423–445.

[21] A. Mazidi, M. Fakhrahmad, and M. Sadreddini. A meta-heuristic approach to CVRP problem: local search optimization based on GA and ant colony. *Journal of Advances in Computer Research* 7 (2016), pages 1–22.

[22] A. Nikolaev and S. Jacobson. Simulated annealing. *Handbook of metaheuristics*. Springer, (2010), pages 1–39.

[23] J. Pennington, R. Socher, and C. Manning. Glove: Global vectors for word representation. *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. (2014), pages 1532–1543.

[24] D. Pisinger. Where are the hard knapsack problems? *Computers & Operations Research* 32.9 (2005), pages 2271–2284.

[25] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv : 1609.04747* (2016).

[26] S. Russell and P. Norvig. *Artificial intelligence: A modern approach*. Pearson Education Limited, (2016).

[27] T. Stützle. An ant approach to the flow shop problem. *Proceedings of the 6th European Congress on Intelligent Techniques & Soft Computing (EUFIT'98)*. Volume 3. (1998), pages 1560–1564.

[28] UPS. *UPS Fact Sheet*. (2020). URL: `https://pressroom.ups.com/pressroom/ContentDetailsViewer.page?ConceptType=FactSheets&id=1426321563187-193` (visited on 05/22/2020).

[29] P. Van Laarhoven and E. Aarts. Simulated annealing. *Simulated annealing: Theory and applications*. Springer Netherlands, (1987), pages 7–15.

[30] F. Vermeulen. Combining Stochastic Local Search with Machine Learning for Vehicle Routing. Master's thesis. (2019).

[31] M. Wright. Scheduling fixtures for New Zealand Cricket. *IMA Journal of Management Mathematics* 16.2 (2005), pages 99–112.

[32] M. Wright. Subcost-guided simulated annealing. *Essays and surveys in metaheuristics*. Springer, Boston, (2002), pages 631–639.

[33] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized Stochastic Gradient Descent. *Advances in Neural Information Processing Systems 23*. Curran Associates, Inc., (2010), pages 2595–2603.