Utrecht University

Master Thesis Computing Science

Combining Stochastic Local Search with Machine Learning for Vehicle Routing

Author:
F. M. Vermeulen

First Supervisor:
Dr. J.A. Hoogeveen

Second Supervisor:
Dr. A.J. Feelders

30th June 2019

ABSTRACT

In this work a hybrid approach to the Capacitated, Periodic, Vehicle Routing Problem with Multiple Trips (and Optional Customers) (CPVRPMT) is presented: First, Simulated Annealing is used to generate a moderately sized batch of sufficiently good solutions. Second, an unsupervised machine learning algorithm is trained on these solutions to obtain vector representations for all customers/orders. The vector representations contain meaningful information about the relation between orders and enable us to calculate the Cosine Similarity between them. The Cosine Similarity we interpret as a measure indicating if two orders should generally be close together in a route and if so, how close? Third, the Simulated Annealing algorithm from the first step (SA) is adjusted so that it can take the Cosine Similarity into account, forming new algorithm SACos.

Both SACos and SA are tested on a simplification of a real case and compared by looking at the average score of their generated solutions under several conditions. It is found that although SACos is slightly slower, it more than significantly outperforms SA.

ACKNOWLEDGEMENTS

Writing a thesis can be not only an intellectual challenge, but also a psychological one: It tests determination, discipline, motivation and resilience.

It is with the second aspect that one person in particular has been of enormous help to me and whom I cannot but mention here. She supported, advised and pushed when needed, and altogether did much more than I could have asked for. I therefore want to say special thanks to:

Monique Pronk

CONTENTS

GENERAL INTRODUCTION

1.1 Introduction

The Vehicle Routing Problem(VRP) concerns itself with the question: "What is the optimal set of routes for a fleet of vehicles to traverse in order to deliver to a given set of customers?"[23]. Apart from it being an interesting problem from a theoretical point of view, answering this question has many practical applications. For many businesses transportation can take up a big part of the budget and for large companies a better set of routes could even save millions.

The VRP is a combinatorial optimisation problem, which means it tries to find the optimal combination of a finite set of objects. It is a generalised version of the well-known Travelling Salesman Problem(TSP), in which merely the shortest route to visit all cities is sought. See Figure 1 for a visualisation of these two problems. Figure b shows the depot in the middle, from where all vehicles must leave and to which they must return.

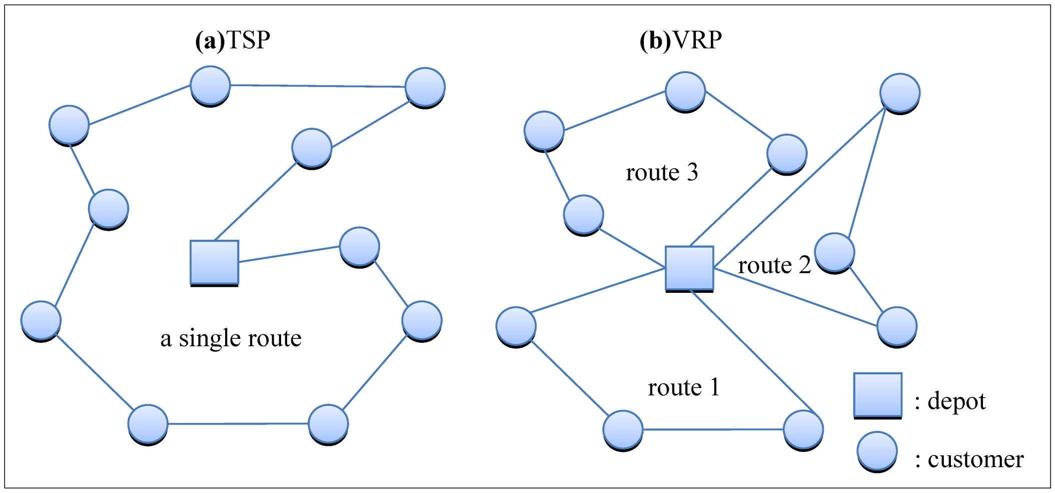


Figure 1: Vehicle Routing Problem

Of course in many real life situations there are a lot more aspects that possibly have to be taken into consideration. This resulted in many variants of the VRP, of which some examples are:

– the Multiple Depot Vehicle Routing Problem: In which there are multiple depots and every vehicle has to end at the same depot it started.

– the Capacitated Vehicle Routing Problem: In which each customer has a certain demand (e.g. the size of their delivery) and each vehicle has a maximum capacity at which it is full.

– the Periodic VRP: In which the schedule is not just one day, but it stretches out over a certain period, and for each customer there is a frequency stating how often they must be serviced in this period. .

– the Stochastic Vehicle Routing Problem: A variant in which one or several elements of the problem are not fixed numbers, but are randomised. Common examples would be the uncertainty of the amount of customers, how much they want delivered or the influence of traffic on how long it takes to go from customer to customer.

Finding the optimal solution to each of these problems would be very useful, but unfortunately they and their entire class of problems are NP-hard. NP-hardness is a term from computational complexity theory that means that for such a problem no polynomial algorithm is likely to exist that solves the problem exactly. It indicates that for larger problem sizes it is infeasible in terms of computation time or computation space to find the optimal solution. They are therefore widely studied problems and a large variety of inventive methods have been devised in search of the best approximations. Our research concerns itself with a special case of the CPVRPMT, which stands for the Capacitated, Periodic Vehicle Routing Problem with Multiple Trips (and Optional Customers). In the next chapter the details of our case will be explained.

1.2 Hybridisation of Metaheuristics and Frequent Pattern Mining

Over the last couple of years, a certain approach to the VRP and other combinatorial optimisation problems has received increasing attention: the hybridisation of metaheuristics and frequent pattern mining techniques [2, 5, 11–17, 24].

Metaheuristics

Metaheuristics are higher-level procedures designed to provide a sufficiently good solution to a problem, to which an exact solution cannot be found, that guide lower-level search procedures. They employ strategies that usually entail some kind of stochastic component and can be used to a wide range of problems. Examples hereof are: genetic algorithms, ant colony optimisation, tabu search and Simulated Annealing.

Frequent pattern mining

Frequent pattern mining or *association rule mining* is a form of data mining that concerns itself with discovering hidden patterns and rules in a larger database. These patterns could be itemsets, substructures, or subsequences. Suppose you have a database containing every purchase made in a supermarket per customer, then examples of these patterns would be:

- A frequent itemset: **{cheese,ham}**, indicating that these are often bought together
- An association rule: **{cheese,ham}** $\Longrightarrow$ **{white bread}**, indicating that íf cheese and ham are bought, white bread is bought as well.

Or suppose you have a set of routes that people often take that is a sequence of locations in the order that they are driven.

- A subsequence: **{Bremen, Hamburg, Berlin}**, indicating that this part of a route is frequently visited.

The general approach to a hybridisation of these two would go as follows:

▷ Run preferred metaheuristic algorithm a number of times until enough solutions are found, among which there are enough good quality solutions.

▷ Run a frequent pattern mining algorithm on the obtained solutions.

▷ Introduce the gathered information back into the metaheuristic algorithm and run it again

1.2.1 Hybridisations in Prior Work

The first of such a hybrid that was proposed was DM-GRASP [15], designed to solve the Set Packing Problem, where a combination of sets has to be found of size k, but in which none of the elements in these sets can overlap. It combined Greedy Randomised Adaptive Search Procedure (GRASP) with frequent itemset mining. A frequent itemset is a combination of elements that frequently occurred together in good solutions.

DM-GRASP has since then been applied to a lot of other problems from combinatorial optimisation, such as the maximum diversity problem[16], the reliable multicast problem[12] and the p-median problem[11]. Others have tried a combination of frequent pattern mining with evolutionary algorithms [13, 14](for the Weighted Constraint Satisfaction Problem), iterated local search [2](for the Multicast Congestion Problem) and breakout local search [24](for the Quadratic Assignment Problem). The last one proposes a general-purpose approach named Frequent Pattern Based Search, which should be applicable to any combinatorial search problem, frequent pattern mining algorithm and optimisation procedure of your choosing. It keeps an elite set of good solutions, from which it mines the frequent patterns. These patterns are used to build solutions that go back into the optimisation procedure. If better solutions are found, they are added to the elite set. When the elite set stagnates, the mining procedure is restarted. This repeats until a stopping criterion is reached.

In 2011, Kammer, Van den Akker, and Hoogeveen [5], used a hybrid approach to solve the Job Shop Scheduling Problem. In Job Shop Scheduling one tries to minimise the makespan of a number of jobs with a certain processing time that have to be carried out on a number of machines with a certain processing power and potential additional constraints. The jobs consist of several operations, which must be executed in a certain order. It ran Simulated Annealing 100 times (Simulated Annealing is an optimisation algorithm further explained in detail below). During run-time, data structures were continuously updated in order to identify *commonalities* -a term coined by Schilham [17]-, who defined commonalities to be the building blocks of good solutions. Two of the commonalities that Kammer et al. [5] mined for were: pairs of operations that would be carried out in a certain order contiguously and operations that were being carried out on a certain machine either first or last. After the commonalities had been mined for, Simulated Annealing was run again, but this time solutions that violated the commonalities were penalised.

1.3 Our Approach: a Brief Outline

In this research we will be exploring a novel hybridisation, namely that of Simulated Annealing[6] with the GloVe algorithm[5] and specifically its applicability to the CPVRPMT.

Simulated Annealing is a randomised optimisation algorithm that starts with one solution, and then iteratively moves towards other solutions that only slightly differ from the current, in search for the optimal solution.

In more detail: Assume the algorithm starts with solution s and tries a move to a neighbouring solution $n(s)$ that is very similar to s. If $n(s)$ is a better solution, it is accepted: $n(s)$ becomes s and a new move can be tried. Occasionally however, it will accept a worse solution, with a probability of p. This enables the algorithm to get out of local optima, and move towards the neighbourhood of the global optimum. p is a variable that is dependent on two things: Firstly, on how much lower the quality of $n(s)$ is than s, where a worse quality means a lower p. Secondly, on the state the process is currently in: The temperature T. The lower T, the lower p is.
T is gradually decreased until p nears 0 and stop criteria are met. Now Simulated Annealing resembles a hill-climbing algorithm, and it can get to the local optimum of the neighbourhood it ends up in. In Chapter 3 Simulated Annealing will be laid out in-depth, along with our implementation.

GloVe is an unsupervised learning algorithm that obtains vector representations for items in a dataset. Said representations can then be used to calculate the items' Cosine Similarity to each other. The Cosine Similarity in this case is essentially a number that conveys how well two items go together. GloVe's workings will be expanded upon in Chapter 4.

The aim of this research is to develop an algorithm that combines these two methods by taking the information gathered with GloVe and reincorporating it into Simulated Annealing, such that significantly better solutions can be found.

We will first let our Simulated Annealing implementation perform a certain amount of runs to obtain good solutions. We will then let GloVe train on the obtained data so that for every combination of orders we can estimate whether they will impact the quality of a solution positively or negatively. This information will be incorporated in a new version of the Simulated Annealing algorithm by affecting the acceptance probability. The resulting hybridisation will be introduced in Chapter 5.

We will use a simplification of a real live case from a garbage collecting company that translates to a CPVRPMT as a test case, described in Chapter 2. In Chapter 6 we will compare the adapted version to the original Simulated Annealing implementation and see whether it is a significant improvement.

CASE DESCRIPTION

2.1 Case in a Nutshell

The van Gansewinkel Group, a company that collects and recycles waste, has to find the most cost-effective schedule to collect waste in a region in the South of the Netherlands. They have a number of garbage collection vehicles that have to service a list of customers on their locations with a fixed amount of garbage. They know the amount of time it costs to travel from one customer to the next and how much time it costs to pick up their garbage. If a certain customer would take a lot of time to serve, it is possible to skip him against a penalty. Some customers want their garbage collected multiple times per week, with naturally the appointments spread evenly over the week. In this case it is only possible to service this customer all requested times or not at all.

The vehicles have to start and end their day at the waste disposal location. They have limited capacity, which means they have to return to the waste disposal location if they are full and take some time to dispose of the waste, after which they can resume their route. We have simplified this assignment, among other things, by only looking at a weekly schedule and not taking into account continuity conditions where customers want their garbage picked up every day around the same time.

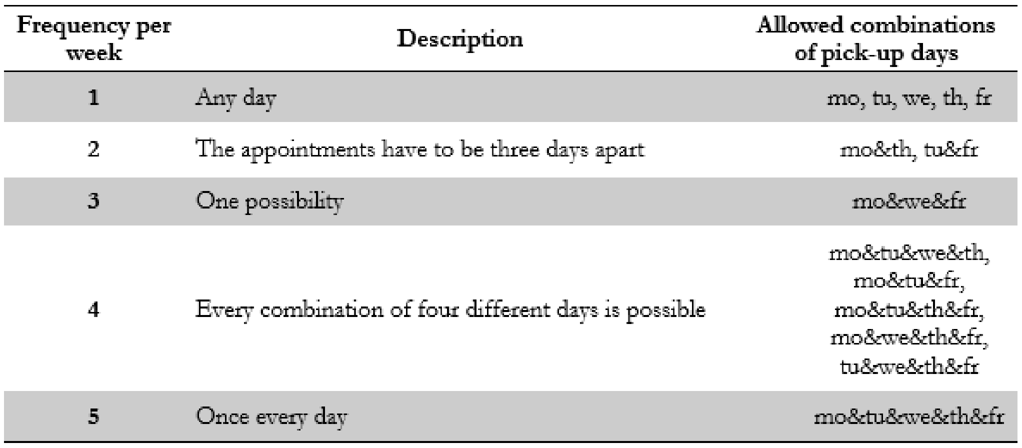
This makes this case, as mentioned before, an instance of the Capacitated Periodic Vehicle Routing Problem with Multiple Trips, the CPVRPMT, with Optional Customers. Although in the classic VRP each customer receives a delivery, whereas in our case something has to be picked-up, these problems are formally the same: In both cases whether or not the capacity constraint is violated is verified by seeing if the sum of all orders on a route exceeds the capacity.

2.2 Detailed Problem Description

The exact specifications of the problem instance:

With regard to the vehicles

– There are 2 identical vehicles with a limited capacity of 100,000 liters



Frequency per week	Description	Allowed combinations of pick-up days
1	Any day	mo, tu, we, th, fr
2	The appointments have to be three days apart	mo&th, tu&fr
3	One possibility	mo&we&fr
4	Every combination of four different days is possible	mo&tu&we&th, mo&tu&fr, mo&tu&th&fr, mo&we&th&fr, tu&we&th&fr
5	Once every day	mo&tu&we&th&fr

Figure 2: Frequency constraints

- The vehicles are available from Monday to Friday from 6:00 to 18:00
- Outside these times the vehicles have to be empty and present at the waste disposal location
- The vehicles can go for another trip if there is still time
- Disposing of waste always takes 30 minutes, regardless of how full the vehicle is at that time, which in effect means that they have to be at the disposal by 17:30
- The two vehicles can dispose of waste simultaneously

With regard to the frequency

The customers want to have their garbage collected between 1 and 5 times per week. Per frequency different combinations of days are allowed. See Figure 2.

With regard to the score

The score of a schedule is defined as the total time in minutes that the two vehicles are occupied plus penalties, entailing that a lower score is a better score. The penalties are calculated as follows: For every customer that is not serviced a penalty of three times the total emptying time is added. E.g.: An order that takes 10 minutes to empty the container and has a frequency of 2, will give you a penalty of 60 minutes if not fulfilled.

The total score then comes down to a summation of:

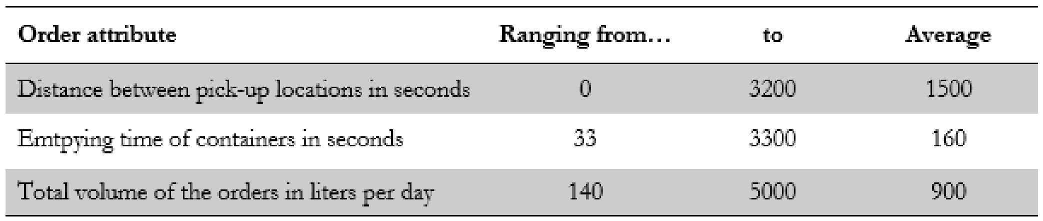
- The total travel time
- The total time of emptying the containers
- The time of disposing of the waste
- The penalties

Supplied input

A file with all orders, their frequency, their daily garbage volume and the time it took to empty this garbage into the truck. Another file with a matrix with the distances between all orders expressed in time.

Noteworthy aspects of this problem

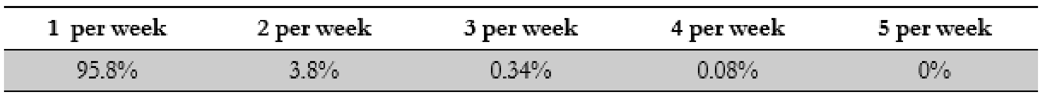
The general size and characteristics of the problem input will be shown to give an idea. Most numbers are rounded for legibility. There were 1177 orders, and their attributes were in the following ranges:



Order attribute	Ranging from...	to	Average
Distance between pick-up locations in seconds	0	3200	1500
Emtpying time of containers in seconds	33	3300	160
Total volume of the orders in liters per day	140	5000	900

Figure 3: Rough size and variation of input data

Of these 1177 orders, the frequencies were distributed as follows:



1 per week	2 per week	3 per week	4 per week	5 per week
95.8%	3.8%	0.34%	0.08%	0%

Figure 4: Percentage of total orders per frequency

Because this specific problem instance had previously already been solved by others, it was known that 14 trips tended to give the best answers. Making the number of trips a variable for Simulated Annealing to be optimised would slow down the algorithm severely and was unnecessary, so we chose to keep this number fixed. 14 Trips meant 1 trip for both vehicles on every day plus 4 extra. Since the frequency 1 orders could be fulfilled on any day, and the amount of frequency 3 and 4 orders was negligible, the 4 extra trips were added on Monday, Tuesday, Thursday and Friday. (Recall Figure 2). It was also known that a solution could be called good if the score got below 5600 minutes.

SIMULATED ANNEALING

3.1 Simulated Annealing

3.1.1 Introduction

Simulated Annealing is a metaheuristic that was inspired by the process of metals annealing [6], a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. By increasing the temperature of the material, there is enough thermodynamic free energy for atomic bonds to break, permitting atoms more freedom to redistribute themselves in their equilibrium state. The more slowly the metal is cooled, the more opportunities the atoms get to redistribute, and so the greater the effect of the annealing.

The Simulated Annealing algorithm correspondingly searches through the solution space, always accepting better solutions and accepting worse solutions with a certain probability. This probability p, is dependent on the quality of the solution and the temperature parameter, according to the following formula:

$$p = e^{\frac{-\Delta}{T}}$$

where Δ = the quality of the current solution - the quality of the candidate solution.

A higher temperature and a higher quality bring about a higher p. T decreases slowly over time, which means that in the first iterations, bad solutions have a big chance of getting accepted, but as the algorithm progresses, this chance slowly diminishes.
The higher initial temperature enables the algorithm to break free from local minima, analogous to the increased thermodynamic energy enabling atoms to break free from their bonds. A slower decrease in temperature in Simulated Annealing will result in better average quality solutions, which is analogous to a slower decrease in metal annealing resulting in metals with fewer defects.

3.1.2 Use in this Research

In this research a schedule represents a solution and the score of the schedule represents the quality of the solution.

3.1.3 More Formally

Below you will find the basic standard pseudocode for Simulated Annealing.

Algorithm 1: Simulated Annealing for the VRP

```
input  : temperature $T$, cooling constant $\alpha$, max no iterations: $i_{max}$, $Q$
output: solution s
Function Simulated Annealing
    ▶ Create start solution $s$;
    ▶ $i = 0$;
    while $i < i_{max}$ do
        for iteration 0 to $Q$ do
            ▶ Pick a random neighbour $s_{new}$ from all neighbours of $s$;
            if acceptance(score($s_{new}$), score(s)) then
                ▶ $s = s_{new}$;
            ▶ $i$++;
        ▶ T = T*$\alpha$;
    return $s$
Function Acceptance(score($s_{new}$), score(s))
    begin
        if score($s_{new}$) < score(s) then
            return true
        ▶ $ScoreDiff$ = score($s_{new}$) - score($s$);
        if $e^{-ScoreDiff/T}$ > a random number between 0 and 1 then
            return true
        else
            return false
```

What defines a neighbouring solution is of course problem-specific, as are the input parameters T and α . The maximum number of iterations is dependent on how fast our algorithm will be, and how much time there is available.

3.2 Applying Simulated Annealing to the van Gansewinkel Case

3.2.1 The Workings

In our implementation of Simulated Annealing SA we represent a solution by one long sequence of orders in the schedule. The orders in the schedule carry with them the information on which day they are filled and by which vehicle.
With the sequence and the information about the day and vehicle established, there is only one way to do them as quickly as possible, namely without waiting in between. Therefore this information is enough to differentiate solutions.

Some terms will be defined below:

– A *trip* is a route a garbage vehicle drives that begins when it is empty at the waste disposal location and ends when it returns there to dispose of its waste.
– A *day schedule* is the entire route for a certain vehicle on a certain day, so one or more consecutive trips.
– The *schedule* is the combination of all day schedules together.
– The *denied list*is a list of all orders that are currently not in the schedule.
– *High frequency* indicates a frequency higher than one.
– *Copies* are the multiple instantiations of an order with a frequency higher than 1.

Neighbourhood of a solution

We define eight possible *moves* to find a neighbouring solution.

1 *Adding an order and inserting it at the best position on a certain trip:*
Taking an order from the denied list and inserting it at the best position (the position that results in the best score) of a randomly chosen trip. If this order has a high frequency, all copies are inserted at the best position possible in a randomly chosen trip that does not violate any frequency constraints.

2 *Adding an order and inserting it at a random position:*
Taking an order from the denied list and inserting it at a random position of a randomly chosen trip. If this order has a high frequency, all copies are inserted at a randomly chosen position in a randomly chosen trip that does not violate any frequency constraints.

3 *Randomly removing an Order:*
Taking a random order from the schedule and adding it to the denied list. If the order has a high frequency, all copies are removed from the schedule.

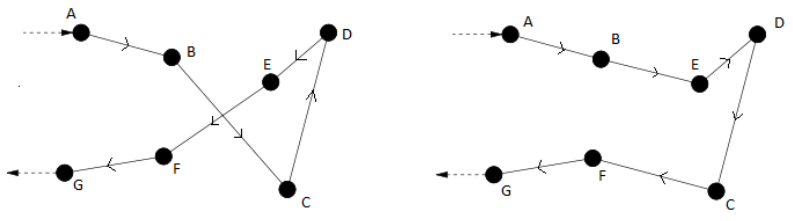
4 *Transferring an Order within the same trip to the best position possible on that trip:*
Taking a random order from the schedule and transferring it to the best position possible on that trip. Because it stays in the same trip, no constraints can be violated. (The volume stays the same, the total driving time can only go down and the frequency constraint is of no issue here.)

5 *Transferring an Order within the same day to a random position on that day:*
Taking a random order from the schedule and transferring it to a random position within the same trip. Again frequency is of no importance here.

6 *Transferring an Order to a different day to the best position possible on a randomly chosen trip on that day:*
Taking a random order from the schedule and transferring it to the best position in a trip on a day that is different from the one that it is in now. For further explanation about satisfying the frequency constraint, see paragraph *"Transferring copies"*. For the rest it works the same as **1**.

7 *Transferring an Order to a different day to a random position on that day:*
Taking a random order from the schedule and transferring it to a random position on a day that is different from the one that it is in now. Again, see paragraph *"Transferring copies"*. For the rest it works the same as **2**.

8 *Applying 2-OPT*
2-OPT is a well-known local search algorithm that discovers if a route crosses itself and reorganises it so it does not. A crossing route can never be shorter than a non-crossing one. See Figure 5.



(a) Crossed route (b) Uncrossed route

Figure 5: 2OPT [20]

In route $A \Rightarrow B \Rightarrow C \Rightarrow D \Rightarrow E \Rightarrow F \Rightarrow G$, uncrossing effectively means reversing subroute $C \Rightarrow D \Rightarrow E$. Therefore 2-OPT tries the reversal of all possible subroutes by calculating the new total distance they yield and comparing it to the existing route. For the example in Figure 5 this calculation requires deleting edges $B \Rightarrow C$ and $E \Rightarrow F$ and adding edges $B \Rightarrow E$ and $C \Rightarrow F$. Our garbage collection case, however, is a directed graph, implying that the distance of $A \Rightarrow B$ is not necessarily the same as the distance of $B \Rightarrow A$. This means that the difference of reversing the subroute to $C \Rightarrow D \Rightarrow E$ has to be calculated as well.

For a 2-OPT move, first a random trip is chosen. Then for orders o_0 to o_n in this trip, the reversal of all possible subroutes is tried. A lot of subroutes are contained in other subroutes. We use this fact in our calculations. For a specification see Algorithm 2. Recall that the score is measured in time, as is the distance.

All calculations can be done in $O(1)$. They are executed $O(n)$ times by the inner loop in line 6, which is itself executed $O(n)$ times by the outer loop in line 5. This makes the total time complexity of this move $O(n^2)$

Algorithm 2: 2-OPT move

```
Function 2opt(trip t, tripSize n, score)
    begin
        ▶ newScore = score ;
        ▶ reverseDifference = 0 ;
        for i=0 to i=n-2 do
            for j=i+2 to j=n-1 do
                ▶ newScore -= distance $o_i \Rightarrow o_{i+1}$ ;                // $O(1)$
                ▶ newScore -= distance $o_j \Rightarrow o_{j+1}$ ;                // $O(1)$
                ▶ newScore += distance $o_i \Rightarrow o_j$ ;                    // $O(1)$
                ▶ newScore += distance $o_{i+1} \Rightarrow o_{j+1}$ ;            // $O(1)$
                ▶ reverseDifference -= distance $o_j \Rightarrow o_{j+1}$ ;       // $O(1)$
                ▶ reverseDifference += distance $o_{j+1} \Rightarrow o_j$ ;       // $O(1)$
                ▶ newScore += reverseDifference ;                               // $O(1)$
                if newScore < score then
                    ▶ A better solution has been found ;
                else
                    newScore = score ;
```

When a new potential position for an order is found, it has to satisfy constraints: Neither the maximum time of the DaySchedule nor the capacity of the garbage truck can be exceeded. If constraints are violated at any time during the move, the neighbour will be rejected and the algorithm will search for another candidate. The acceptance function will only be called with a valid schedule.

Some motivation

For higher frequency orders, if all copies always have to be transferred at the same time, the probability of producing a valid solution will be rather low. This will mean a lot of failed moves that waste time. If an order is transferred within the same day, all other copies can stay where they are. Another benefit is that inserting an order in a trip might exceed the time limit, whereas transferring it within the same trip has a smaller chance of doing so. By splitting the transferring moves up in a same day variant and a different day variant, the probability of higher frequency orders changing position at all increases.

Transferring copies

When a randomly obtained higher frequency order has to be transferred to a different day, first the day is chosen and afterwards randomly vehicle 1 or 2. Satisfying the frequency constraints can always be achieved by transferring at most two copies, in the following way:

– *Frequency 2* orders only have two possible day-combinations. If there has to be switched between Monday and Tuesday, the other one has to switch between Thursday and Friday and vice versa.

– *Frequency 3* orders are only possible on fixed days. As all copies are equal, letting two copies switch days would be the same as transferring within the same day. The only reason the different day schedule move is used for frequency 3 orders was to ensure the other vehicle is also attempted.

– *Frequency 4* customers are visited on all days except one. The random order is simply transferred to this one day.

3.2.2 Implementation Details

We have implemented the Simulated Annealing algorithm in java 1.8. The experiments were carried out on a computer equipped with an Intel(R) Core(TM) i7-7820HQ Processor with 2.90 GHz and 8GB RAM on Windows 10.

Incremental updating

Considering that a higher number of iterations makes a better score more likely and that a shorter time per iteration means more iterations in the same time span, it is important that our algorithm be implemented as efficiently as possible. A solution is a rather large object, which means that both calculating the score of the entire object and copying the object are costly operations. This is why the scores of candidate solutions are calculated by looking at the difference from the current solution's score and "new solutions" by making modifications to the same solution object.

When an order o is removed we simply deduct the travelling time from the previous order o_{prev} to o, the emptying time of o and the travelling time from o to the next order o_{next}, and add the travelling time from o_{prev} to o_{next} plus 3 times the emptying time of o to account for the penalty. Inserting an order is the reverse and transferring an order is then merely a combination of removing and inserting an order. When the difference in score is calculated, the neighbour can be either rejected or accepted. Only when it is accepted, are the changes applied to the solution.

Choice of data structures a solution

The data structure for a solution has to be able, among other things, to perform the following actions in constant time:

1. randomly choose an order from the schedule
2. insert an order at or delete an order from any position in the schedule
3. randomly choose an order from a certain day schedule

To achieve this, something resembling a combination of an ArrayList and a LinkedList was designed. Every trip is represented by an ArrayList of orders that begins and ends with a schedule entry for the waste disposal. The ArrayList.get(index i) method can be carried out in constant time, so by choosing a random integer between 1 and the size of the ArrayList minus 1 and retrieving the order with that integer as an index, action 1 and 3 on the list are achieved. The ArrayList.add(index i) and ArrayList.remove(index i) are very costly, however. To solve this all orders that had a position in the schedule are wrapped in *schedule entries.*

A schedule entry is an order plus information about its position in the trip, such as the day, the vehicle and a reference to the previous and the next schedule entry. As a result the order of succession in the LinkedList is completely independent from the order of succession in the ArrayList. Inserting or removing a schedule entry can now be reduced to updating the references of the preceding, the succeeding and said schedule entry. Updating references is a constant time operation, so also item 2 on the list can be checked off.

Having different ArrayLists for different trips means .add and .remove methods had to be used nonetheless. Fortunately, because the order of the schedule entries in the ArrayList is irrelevant, adding can be done at the end. Removing a schedule entry is done by putting the last schedule entry of the ArrayList at the index of the former and removing the last slot. All these actions can again be performed in constant time.

Choice of data structure for the denied list

The denied list works as a queue; orders that are candidates to be added to the schedule are taken from the front of the list and if they are rejected they are appended to the end. Orders that are removed from the schedule are appended to the end of the list as well. The only requirement for the denied list is that these actions could be performed in constant time. This is easily achieved with a standard LinkedList.

3.3 Optimising SA

3.3.1 Begin Solutions

At the beginning of a run, the algorithm does a couple of iterations to create a begin solution. These initial iterations have slightly different parameters from the rest of the iterations. The α and temperature are set so that the acceptance probability of worse solutions remains high, and there are only add- and remove moves performed. This produces a rather full schedule, albeit not with a very good score, but it produces it quickly. Starting from a fuller schedule improves the final schedule significantly, while only costing a small amount of iterations. From here on out, when the parameters of a solution are being mentioned, it is always the parameters of the last part of the iterations that are referenced.

3.3.2 Finding the Best Parameter Settings

After setting the program up as sensibly as possible, the parameters still have to be set. This was done by hand. These parameters consist of:

The initial temperature, Q and α

The ideal initial temperature should be such that the acceptance ratio of bad to good neighbours is equal to a certain value ρ_{init}. It should then slowly decrease by a factor of α, a number just below 1, with every cycle of Q until another acceptance ratio ρ_{end} is reached in the final iteration:

i_{max}. This ρ_{end} should, as mentioned before, be nearing 0 to cause the algorithm to be greedy. This would entail that a higher i_{max} requires α to be bigger, to cause the temperature to decrease more slowly.

How often each move was attempted

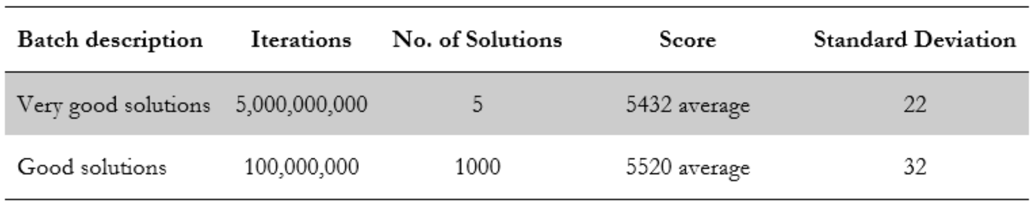
There are eight different moves possible to obtain a neighbour, but not all of these moves should be attempted with the same frequency. Things to take into consideration are:

Not all moves take the same amount of time: all random moves are constant time, all moves looking for the best position are linear in the size of the trip and 2-OPT even is quadratic.
The ratio between add and remove moves influences how full the schedule will end up being. The number of calls to moves that add an order has to be several times bigger than those to the removing one for it to be filled at all. Also: the fuller the schedule becomes, the harder it will be for a new order to be added to a certain trip whilst remaining a valid solution.

The i_{max} for the thousand "good solutions" was set at 100,000,000, which took about 2 minutes. None of the moves that searched for the best position in a trip turned out to improve the score in the end even though they increased the run-time considerably, so their frequency was set to 0.

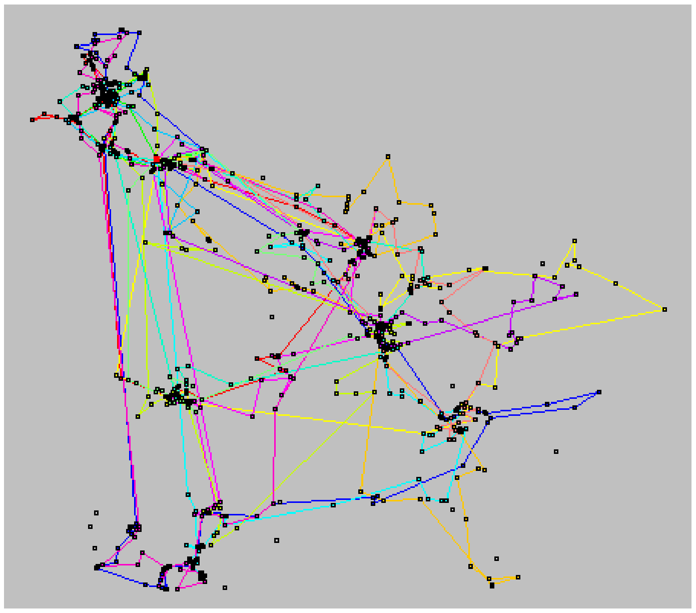
3.4 Results

1005 solutions were generated. See Figure 6 for an overview of these solutions. See Figure 7 for visualisations of solutions and for what the difference between a rather bad and a very good solution might look like.

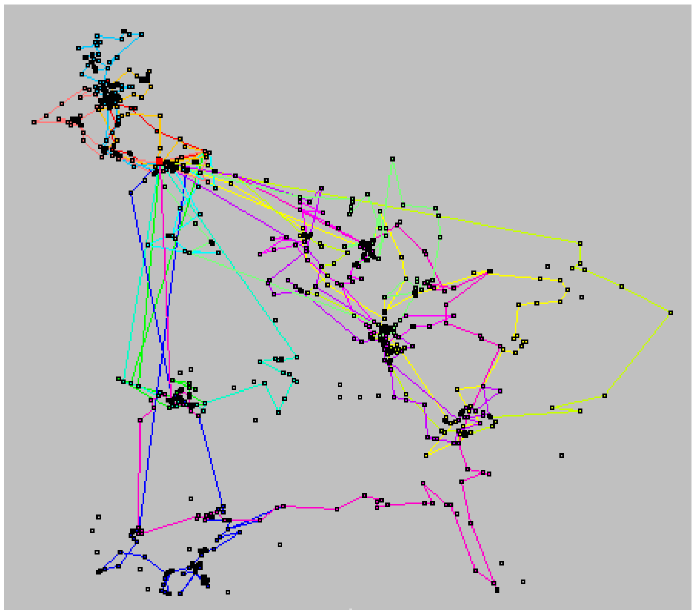


Batch description	Iterations	No. of Solutions	Score	Standard Deviation
Very good solutions	5,000,000,000	5	5432 average	22
Good solutions	100,000,000	1000	5520 average	32

Figure 6: Summary of obtained solutions



(a) Score: 6168 min



(b) Score: 5399 min

Every colour represents a Trip. The red square on the left upper side represents the waste disposal location

Figure 7: Visualisations of some solutions

The solutions were all generated with the same parameters. The very good solutions were generated to see how well our algorithm would do within a reasonable amount of time.

GLOVE

In this chapter the GloVe learning algorithm [10] will be explained in addition to how it can be used to gain information about good solutions.

4.1 The Objective

In Chapter 3 circa 1000 solutions with a relatively good score were generated. Next, we require a way to extract from these sequences information on what constitutes a good solution that can aid our Simulated Annealing algorithm when it is searching for better solutions. SA tries moves by attempting to insert an order into -or delete it from- an existing solution. We would like to have a measure to express how well that order generally fits with the orders currently in that part of the route. This information could guide the algorithm to where the principal objective (a better score) might be best achieved. To this end we apply GloVe, which is both a model describing the similarities between data entities and an algorithm to obtain them.

4.2 Introduction

GloVe is an algorithm originally designed for language processing purposes. It tries to find vector representations for words such that they best capture their meaning. These vectors can then be used as features in various applications, such as information retrieval [18], document classification [19] and question answering [21]. The vectors GloVe finds, that can consist of hundreds of components, have been proven to contain meaningful substructures: Not only can they contain the information that *queen* is similar to *woman*, they can also convey more complicated relationships such as: "*king* is to *queen* as *man* is to *woman*.".

Another interesting possibility vector representations offer is an easy way to calculate the similarity between two words, via the Cosine Similarity. The Cosine Similarity is a measure originating from the geometric interpretation of vectors. It measures the cosine of the angle between two vectors of any number of components. In the field of Information Retrieval, however, it has also successfully been used as a measure of linguistic or semantic similarity between word vectors for decades [7].

The model is founded on the idea that how often words co-occur (or not) should tell you something about the semantic relation between them. Given a corpus of text, GloVe counts the occurrence of every pair of words, where both words are within a predefined distance of each other. It has defined the mathematical relationships that are expected to hold between the occurrences and tries to find vector representations for all words such that they adhere to these relationships as much as possible. The vectors are found by training the model with stochastic gradient descent.

There are other methods for learning vector space representations of words, but GloVe is rather recent and renowned in its field. As a result many implementations were readily available online, including one in Java. This meant that the two different parts of the research, Simulated Annealing and GloVe, could be combined smoothly.

4.2.1 GloVe for the Vehicle Routing Problem

Even though GloVe usually concerns itself with words, the vector representation is obtained by training on the co-occurrence matrix only. As this is merely a matrix of co-occurrences of items, it can easily be translated to our problem instance, where orders represent words and trips represent documents. The Cosine Similarity translates to an appropriate way to measure how well two orders match. With GloVe we have a sophisticated way both to generate meaningful information out of our data and to compare pairs of orders efficiently.

4.3 Model Definition

GloVe bases its vectors on relationships that hold between the co-occurrence probabilities of words in the $Vocabulary$ of a corpus. The Vocabulary (with size V) is defined as the collection of all distinct words occurring in a corpus. GloVe will use an unsupervised learning algorithm to try and find the vectors for all words 1 to V that best satisfy these relationships. It must therefore first of all obtain the co-occurrence matrix.

4.3.1 The co-occurrence matrix

The co-occurrence matrix X is a 2-dimensional matrix where element X_{kl} counts how often word l occurs in the window of word k. See Figure 8a. The window is limited by both the document and the *window size*, which is a user-specified parameter that indicates the maximum number of words there can be between k and l for them to contribute to X_{kl}. The difference in position in the text between k and l is called the *distance* d. To account for the fact that very distant word pairs are expected to be less relevant than close ones, every co-occurrence is multiplied by a weight of $1/d$ before it is added to X.

Symmetry

For every entry X_{kl} counting how often word l occurs in the window of word k, there is a value X_{lk} counting how often k occurs in the window of l, making X symmetrical.

The value for vector u_k will depend on values X_{k1} to X_{kV}. From this -together with the fact that X is symmetrical- it follows that it would also be possible to define a second vector v_k, equal to u_k, that depends on entries X_{1k} to X_{Vk}. This concept is illustrated by Figure 8b. Vectors u_k and v_k should in theory be the same, but because the learning algorithm is initialised randomly, its resulting vectors can differ. Both vectors are used and combined so as to reduce overfitting [1].

$$\begin{array}{c|ccccccc} \boldsymbol{X} & 1 & 2 & \cdots & k & l & \cdots & V \\ \hline 1 & 1 & 0.1 & \cdots & 0.64 & 0.34 & \cdots & 0.23 \\ 2 & 0.1 & 1 & \cdots & 0.05 & 0.1 & \cdots & 0.57 \\ \vdots & \vdots & \vdots & 1 & \vdots & \cdots & \iddots & \vdots \\ k & 0.64 & 0.05 & \cdots & 1 & 0.78 & \cdots & 0.13 \\ l & 0.34 & 0.1 & \cdots & 0.78 & 1 & \cdots & 0.1 \\ \vdots & \vdots & \iddots & \cdots & \vdots & \cdots & 1 & \vdots \\ V & 0.23 & 0.57 & \cdots & 0.13 & 0.1 & \cdots & 1 \end{array}$$

(a)

$$\begin{array}{c|ccccccc} \boldsymbol{X} & 1 & 2 & \cdots & k & l & \cdots & V \\ \hline 1 & 1 & 0.1 & \cdots & 0.64 & 0.34 & \cdots & 0.23 \\ 2 & 0.1 & 1 & \cdots & 0.05 & 0.1 & \cdots & 0.57 \\ \vdots & \vdots & \vdots & 1 & \vdots & \cdots & \iddots & \vdots \\ k & 0.64 & 0.05 & \cdots & 1 & \boxed{0.78} & \cdots & 0.13 \\ l & 0.34 & 0.1 & \cdots & \boxed{0.78} & 1 & \cdots & 0.1 \\ \vdots & \vdots & \iddots & \cdots & \vdots & \cdots & 1 & \vdots \\ V & 0.23 & 0.57 & \cdots & 0.13 & 0.1 & \cdots & 1 \end{array}$$

(b)

Figure 8: Co-occurrence matrix X

4.3.2 Vector Definitions

The exact relationships between the co-occurrence probabilities that GloVe proposes will not be discussed here, but for a more in-depth explanation see Pennington et al. [10]. It suffices to relay that after rewriting the equations from the co-occurrence probabilities, for words k and l, we are looking for vectors u_k and v_l such that:

$$u_k^\intercal v_l = \log X_{kl} \tag{4.1}$$

Definitions of the terms:

- u_k is the first vector representing word k.
- $u_k^\intercal$ is the transpose of the vector.
- v_l is the second vector of word l, as explained in Subsection 4.3.1.
- X_{kl} is how often l occurs in the context of k, weighted s.t. all numbers lie between 0 and 1.

4.3.3 Objective function

We can now define the following objective function F_o that has to be minimised by finding the appropriate values for all vectors u_k and v_l:

$$Minimise\ F_o = \sum_{k=1}^{V}\sum_{l=1}^{V} f(X_{kl})\big(u_k^\intercal v_l - \log X_{kl}\big)^2 \tag{4.2}$$

The number of components for each vector is a user-specified parameter. The more components, the richer the information that can be held. A higher number of components does unfortunately mean a more complex program, with the added processing time that comes along.

If the only thing the objective function considered was Equation 4.1, all co-occurrences would weigh equally, even very rare and very frequent ones. A standard solution to address such issues is employing a *weighted least squares* regression model, resulting in the squaring of the expression and the addition of the *co-occurrence weighting function* $f(X_{kl})$.

The co-occurrence weighting function is defined as follows:

$$f(x) = \begin{cases} (x/x_{max})^{\beta} & \text{if } x < x_{max} \\ 1 & \text{otherwise} \end{cases} \tag{4.3}$$

where both x_{max} and β are user-specified parameters:

- x_{max} determines the cut-off above which a higher co-occurrence count of k and l does not matter
- β determines how heavy frequent co-occurrences are weighted

4.4 Training the Model: Stochastic Gradient Descent

To find the vectors that minimise the objective function, GloVe employs AdaGrad, a stochastic gradient descent optimisation algorithm. Such algorithms and AdaGrad in particular will be explained in this section.

4.4.1 Gradient Descent

When attempting to model the relationship between predictor variables x_i, y_i and their response variable z_i, a common approach is regression analysis. In regression analysis one tries to find the function $f(x, y)$ that best predicts the actual responses z. In order to do this the *error function* F is derived, which is a function describing how much the current model (function $f(x, y)$) differs from the actual response values. For an explanation of the derivation of the error function see [22].

For a linear function $f(x, y)$ the error function $F(a, b, c)$ will typically look like this:

$$F(a, b, c) = \frac{1}{N} \sum_{i=1}^{N} (z_i - (ax_i + by_i + c))^2$$

We are looking for a way to find parameters a, b and c that minimise error function F. Gradient descent is an example of an algorithm that can do exactly that.

The idea behind gradient descent is the following: The slope (or gradient) of a function $F(a)$ is always 0 at a minimum. At all other points in a graph following $F(a)$ in a direction such that the slope is negative, will lead towards a local minimum. So, by constantly updating parameter a at step τ with the following rule, gradient descent closes in on a minimum of $F(a)$:

$$a_{\tau+1} = a_{\tau} - (\gamma_a * \frac{\partial}{\partial a_{\tau}})$$

where $\frac{\partial}{\partial a_\tau}$ is the slope of $F(a)$ in point a_τ and γ_a is the step size. A larger step size means it will take gradient descent less steps to reach the minimum, but makes it less precise. To combine the best of both worlds, γ is usually dependent on the step number, where it decreases with every step. As such, big steps towards the general vicinity of the minimum can be taken in the beginning and smaller steps at the end to approximate it as closely as possible.

For a function with multiple parameters such as error function $F(a, b, c)$, calculating the slope is not as straightforward. First, the partial derivatives will have to be computed. The partial derivative of e.g. parameter a is defined as the derivative of function $F(a) = F(a, b, c)$ where b and c can be treated as constants, because they are considered per time step. Now it is possible to define the update rules for each parameter:

$$a_{\tau+1} = a_\tau - (\gamma_a * \frac{\partial}{\partial a_\tau}) = a_\tau + \gamma_a * \left[\frac{2}{N}\sum_{i=1}^{N} x_i(z_i - (a_\tau x_i + b_\tau y_i + c_\tau))\right] \tag{4.4}$$

$$b_{\tau+1} = b_\tau - (\gamma_b * \frac{\partial}{\partial b_\tau}) = b_\tau + \gamma_b * \left[\frac{2}{N}\sum_{i=1}^{N} y_i(z_i - (a_\tau x_i + b_\tau y_i + c_\tau))\right] \tag{4.5}$$

$$c_{\tau+1} = c_\tau - (\gamma_c * \frac{\partial}{\partial c_\tau}) = c_\tau + \gamma_c * \left[\frac{2}{N}\sum_{i=1}^{N} z_i - (a_\tau x_i + b_\tau y_i + c_\tau)\right] \tag{4.6}$$

Coming back to our initial problem of finding the best values for vectors u_k and v_l: Instead of function $F(a, b, c)$ above with merely 3 parameters, it is possible to optimise any kind of function, for instance objective function F_o:

$$\textit{Minimise } F_o(u_1 \dots u_V, v_1 \dots v_V) = \sum_{k=1}^{V}\sum_{l=1}^{V} f(X_{kl})\big(u_k^\intercal v_l - \log X_{kl}\big)^2 \tag{4.7}$$

4.4.2 Stochastic Gradient Descent

For regression analysis with multiple variables, the standard gradient descent algorithm will have to calculate the sum of many terms at every step for each parameter. For a function with a high number of variables such as F_o, this can become very costly.

Stochastic gradient descent is an adaptation on gradient descent devised to advance towards the minimum more quickly. Before training starts, all data points $(X_{1,1} \dots X_{V,V})$ are shuffled so they are in random order. With every step it only calculates the gradient for *one* data point, going through the data points in the new random order. If the stop criterion has not yet been met by the time all data points have been used, they can be reshuffled and training can resume on the new list.

4.4.3 AdaGrad

AdaGrad[3] is an extension of stochastic gradient descent in that it has a separate learning rate for each parameter (for instance γ_a for parameter a in Equation 4.4). This fine-tuning often improves convergence performance for parameters with few data points. It nonetheless requires a base learning rate η that is applied to all variable-specific learning rates.

AdaGrad expects two user-specified parameters in total:

– The *base learning rate η:* How large the step size with every iteration will be

– The number of *iterations:* The number of steps/iterations the algorithm can perform before it halts.

4.5 Cosine Similarity

Once AdaGrad has halted and all vector representations are acquired, we can use the Cosine Similarity to perform meaningful computations on them. As explained before, the Cosine Similarity between two vectors measures the cosine of the angle between the two. Between two non-zero vectors $\boldsymbol{A}$ and $\boldsymbol{B}$ it is defined as

$$\text{cosineSimilarity}(\boldsymbol{A}, \boldsymbol{B}) = \frac{\boldsymbol{A} \cdot \boldsymbol{B}}{\|\boldsymbol{A}\| \, \|\boldsymbol{B}\|} \tag{4.8}$$

This entails that two vectors of the exact same orientation have a Cosine Similarity of 1. For two vectors orthogonal to each other the Cosine Similarity is 0 and when their direction is exactly opposite it is -1. This allows us to express the similarity of any two words in our vocabulary with a simple number. One could for instance say that the words *king* and *man* have a similarity of 0.7.

With the Cosine Similarity the final step has been reached toward defining our new algorithm SACos that will be presented in the following chapter.

SIMULATED ANNEALING WITH AGREEABILITY

5.1 Introducing SACos

After running GloVe, we have a vector representation for every order from the problem instance and so the Cosine Similarity between every combination of two orders can be looked up. We will utilise the similarity by introducing it back into our previous Simulated Annealing algorithm (SA), creating a modified version we name SACos.

Whenever a move in SACos is tried, an order with variable name *kim* has to be inserted at (or removed from) a certain position within a sequence of orders. The variable name *kim* is chosen as such, both to emphasise that this a named order gathered from user input and to distinguish it more clearly from other auxiliary variables that will be introduced below. The Cosine Similarities between *kim* and each of the orders in the sequence together should give some idea as to how well *kim* would fit around there.

We combine the Cosine Similarities relevant to a move and convert them into a number that we can use in our algorithm. This number we call the *agreeability.* The agreeability will influence the probability of accepting lower quality solution positively, when this solution complies with it better than the current one.

5.2 The Theory

5.2.1 Introducing Agreeability into the Algorithm

Recall the acceptance function from Algorithm 1 that decides whether or not a move towards a neighbouring solution is accepted:

Algorithm 3: Old acceptance function

```
1 Function acceptance score($s_{new}$), score($s$)
2   begin
3     if score($s_{new}$) < score($s$) then
4       return true

5     $ScoreDiff$ = score($s_{new}$) - score($s$);
6     if $e^{\frac{-ScoreDiff}{T}}$ > a random number between 0 and 1 then
7       return true
8     else
9       return false
```

The only thing that changes in SACos is an alteration in the probability of accepting a neighbouring solution. The *weighted agreeability* $w_a(a)$ of the move is inserted at line 6. This term consists of agreeability a, weighted by the agreeability weighting function w_a with a subscript to discern it from the distance weighting function w_d introduced below. Line 6 now looks as follows:

if $e^{\frac{-ScoreDiff}{w_a(a)*T}}$ > *a random number between 0 and 1* **then**

In other words, all moves that improve the solution are still always accepted and the alteration will only influence the probability of accepting moves that are worse.

5.2.2 Converting Cosine Similarities to Agreeabilities

The Cosine Similarity itself is a number between -1 and 1 and only dependent on two orders. It needs to be converted into the agreeability such that it can be applied to a move and such that it causes the desired behaviour in SACos.

In the following definitions we will employ orders denoted by variables *kim* and *leo*, to clearly distinguish between orders by name and orders denoted by their position.

The agreeability $a(kim, leo)$ between two orders *kim* and *leo* is defined as an upward shift from the Cosine Similarity, adjusted by user-specified parameter *cap*:

$$a(kim, leo) = \frac{(\text{cosineSimilarity}(kim, leo) + 1)^{1+cap}}{2^{cap}}$$

The parameter *cap* indicates from which point on a Cosine Similarity is deemed to be good, i.e. from which point within range [-1,1] a Cosine Similarity will result in an agreeability ≥ 1. The rest of the formula is to ensure all possible Cosine Similarity values will still be in [0,2], for any value of *cap*.

Some explanation of terms:

- o_i*:* The order at position i in a trip of length n with $1 \leq i \leq n$
- d*:* The distance d is the difference in position between two orders.

- *d_{max}:* A user-specified parameter that indicates the maximum distance that contributes to the agreeability.
- *i_{min}:* The larger of $i - d_{max}$ and 0 (because the beginning of the trip might have been reached)
- *i_{max}:* The smaller of $i + d_{max}$ and n (because the end of the trip might have been reached)
- *$w_d(i,j)$:* A distance weighting function determining the influence on the agreeability of the distance between position i and position j

The agreeability of *inserting* one copy of order *kim* into an existing schedule at the position before i is defined as the mean of the agreeabilities between *kim* and each of the orders that are within distance d_{max} of point i, weighted by the distance weighting function $w_d(i,j)$. Weather or not *kim* fits at that position is dependent on the current solution, but for simplicity reasons we assume the current solution given and will therefore omit it from all formulas.

$$a_{\text{INSERT}(kim,i)} = \frac{\sum_{j=i_{min}}^{i_{max}} w_d(i,j)a(kim, o_j)}{\sum_{j=i_{min}}^{i_{max}} w_d(i,j)}$$

The agreeability of removing one copy of order *kim* currently at position i is the inverse of inserting it there:

$$a_{\text{REMOVE}(kim)} = 2 - a_{\text{INSERT}(kim,i)}$$

The agreeability of transferring one copy of order *kim* to the position before i is the mean of removing it and adding it before i:

$$a_{\text{TRANSFER}(kim,i)} = \left(a_{\text{REMOVE}(kim)} + a_{\text{INSERT}(kim,i)}\right)/2$$

The agreeability of any *move* (INSERT,REMOVE or TRANSFER) concerning order *kim* with copies kim_1 to kim_m, is simply the average of the agreeabilities of all copies:

$$a_{move(kim)} = \left(\sum_{l=1}^{m} a_{move(kim_l)}\right)/m$$

Now the only remaining thing that we need is a way to control how much effect the agreeability has on SACOS, so we introduce a weighting function on the agreeability. The weighted agreeability $w_a(a_{move})$ of a move is the agreeability corrected by the user-specified weight for that move:

$$w_a(a_{move}) = (a_{move})^{w_{move}}$$

In the acceptance function the term $e^{\frac{-ScoreDiff}{w_a(a)*T}}$ decides if a move gets accepted or not. This means that a $w_a(a)$ greater than 1 results in a higher probability of being accepted and a $w_a(a)$ less than 1 in a lower probability. The unweighted agreeability is a number between 0 and 2 where how much higher it is than 1 indicates how good it is. By this definition of the weight these properties are preserved.

5.3 Implementation

The implementation used for the Simulated Annealing part of SACos is the same as the one from Chapter 3, with of course above described additions. For the GloVe part the implementation we use is a Java implementation by Papaoikonomou [9], with a small adaptation such that it conforms to the model Pennington et al. describe. For the precise difference see Appendix 1.1.

5.3.1 Complexity considerations

Storing the Cosine Similarities

The vectors are learned by training on the trips from all generated good solutions. After the training phase, all Cosine Similarities between all pairs of orders are calculated and saved in a matrix to ensure looking them up can be done in $O(1)$ time when running SACos.

Calculating the agreeability

Simulated Annealing will give a better score by letting it run for more iterations. This means that introducing agreeability would become useless if the new version of the algorithm would yield better solutions with the same amount of iterations, but increasing the computing time to such a degree, that it would have been just as well to let the original algorithm run longer. It is therefore important that introducing agreeability will not increase the time complexity too much.

To keep this effect as small as possible, the agreeability is only calculated when it is actually used. A lot of moves will actually fail, because they violate the constraints of a valid solution. From the remaining moves a percentage would be immediately accepted because they improve the score. Only when the score of the new solution is worse than the current score, is it required to calculate the agreeability. In practice the calculation of the agreeability is not carried out all that often, reducing the increase in time complexity.

A remark

During implementation and testing it was noted that the end score of a run is correlated with the score of the begin solution. In the original version of the algorithm, the first couple of iterations are run with slightly different settings to quickly create full solutions that could afterwards be fine-tuned more slowly. The quality of a begin solution actually has a small but significant impact on the score of the final solution. For that reason 200 begin solutions were saved and these exact begin solutions were loaded into the algorithm for every trial. Hence, it was possible to do a pairwise sampled t-test instead of an independent samples T-test[8], vastly decreasing the amount of runs necessary to get significant results.

5.4 Parameter Optimisation

With the modification of our algorithm, a batch of new parameters were introduced. In the original algorithm from Chapter 3 that was used to generate the solutions on which to perform data mining, all parameters involved were optimised by hand. There are, however, many automated techniques available for parameter tuning. These can be preferable over manual tuning for among other several reasons: Most evidently, the algorithm designer will be spared from having to put in countless hours of testing and tuning. Furthermore, a well-devised algorithm might find better parameter configurations that a human has missed. Lastly, using an automated optimisation when comparing two different algorithms is less prone to bias. When two algorithm whose performance depends heavily on its parameters are compared, it would be all too easy to put in a little more effort into the tuning of the favoured one, and a little less in the other. This research has employed Halton Sequences to automate the process [4].

5.4.1 Halton Sequences

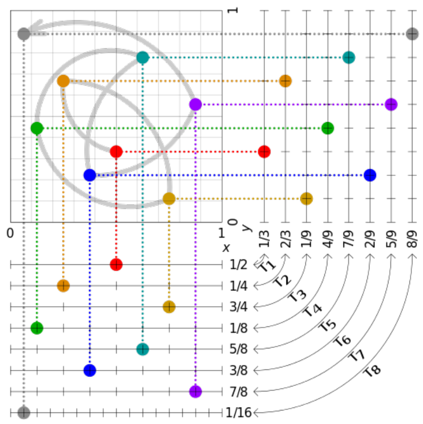
If we want to evenly divide up an n-dimensional space with h points, we can use n Halton sequences to do so. As an example we will use the first 7 numbers of the Halton sequence with base 2 to divide up the interval $(0, 1)$. We begin by dividing 1 by the base, so in halves, then by $2 * 2$, so in quarters, then in eights, et cetera. Points that have already been visited because they can be rewritten as a simpler fraction are of course skipped:

$$\frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{5}{8}, \frac{3}{8}, \frac{7}{8}$$

To divide up the 2-dimensional plane $\big(0, 1\big) \times \big(0, 1\big)$ we need another base for the second dimension. Each dimension requires a base number that is *co-prime* with all other bases, i.e., their only common divisor is 1. A good choice for the second base would be 3 as this is co-prime with 2. With $h = 8$ this would result in the following coordinates:

$$\left(\frac{1}{2}, \frac{1}{3}\right) \quad \left(\frac{1}{4}, \frac{2}{3}\right) \quad \left(\frac{3}{4}, \frac{1}{9}\right) \quad \left(\frac{1}{8}, \frac{4}{9}\right) \quad \left(\frac{5}{8}, \frac{7}{9}\right) \quad \left(\frac{3}{8}, \frac{2}{9}\right) \quad \left(\frac{7}{8}, \frac{5}{9}\right) \quad \left(\frac{1}{16}, \frac{8}{9}\right)$$

A visual representation of these coordinates you can see in Figure 9. In our examples all intervals are between 0 and 1, but these are easily scaled to any interval.



(a) Visualisation of Halton sequences with base 2 and 3

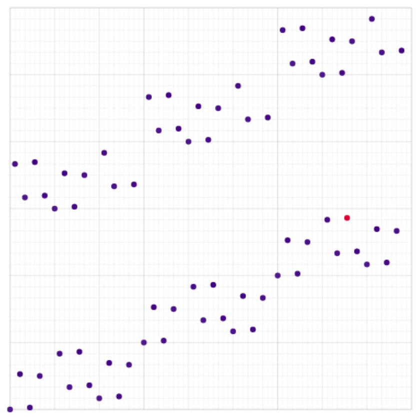

(b) Visualisation of Halton sequences with base 3 and 6

Figure 9: Visualisations of Halton Sequences

Translating this back to parameter optimisation: All n dimensions with their respective intervals represent n parameters with their respective range of possible values and every coordinate represents a parameter configuration. Dividing up the n-dimensional space evenly translates to having the most diverse combinations of parameter values. The benefits of using Halton Sequences over generating random numbers is that they need less points to cover the space more evenly. What is being prevented by correctly using Halton Sequences with prime bases instead of some other less thought-through method is illustrated by the example in Figure 9b that combines two Halton Sequences with ill-combining base numbers.

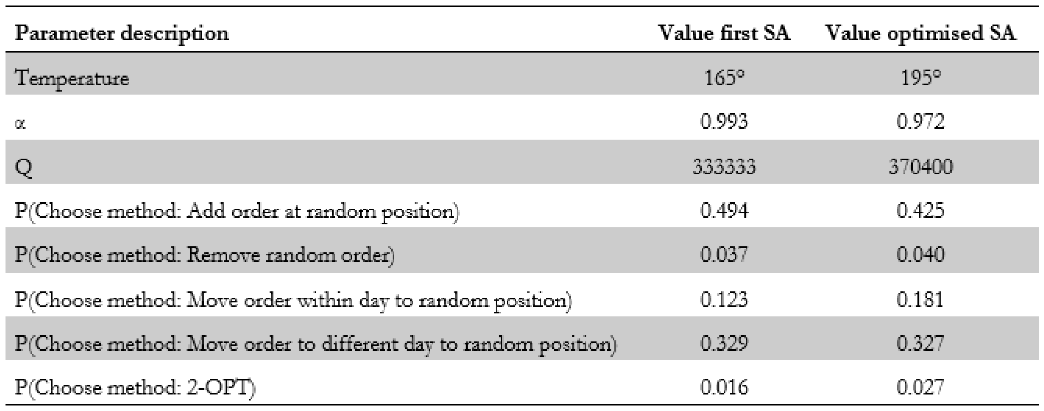
When all h parameter configurations have been tested, we take the configuration with the best average score as our optimal configuration.

5.4.2 Optimising the Old Parameters from SA

In our case the original algorithm SA with which we will compare our new algorithm SACos is directly incorporated in its competitor, which means that SACos shares the first part of its parameter with SA. In choosing the configurations of their *shared parameters* two things were considered

1. In order to achieve the best possible results with SACos, the configuration was optimised by employing Halton Sequences.
2. To make the comparison as significant as possible, we use the optimised configuration in both algorithms.

The parameters were slightly tweaked as can be seen in Figure 10.

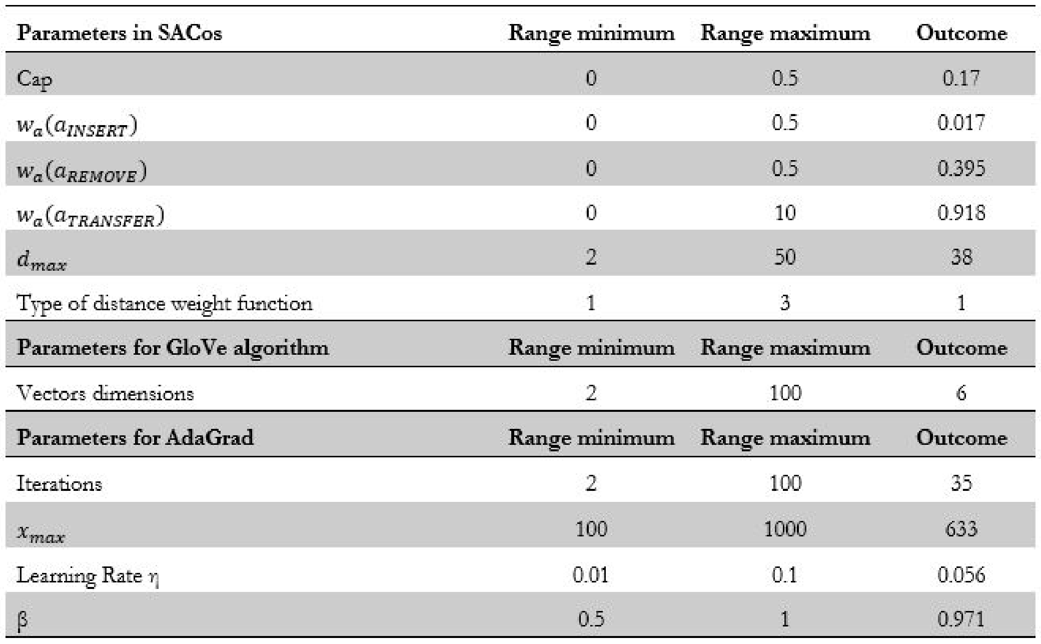


Parameter description	Value first SA	Value optimised SA
Temperature	165°	195°
α	0.993	0.972
Q	333333	370400
P(Choose method: Add order at random position)	0.494	0.425
P(Choose method: Remove random order)	0.037	0.040
P(Choose method: Move order within day to random position)	0.123	0.181
P(Choose method: Move order to different day to random position)	0.329	0.327
P(Choose method: 2-OPT)	0.016	0.027

Figure 10: Parameters used for Simulated Annealing old and fine-tuned

5.4.3 Optimising the New Parameters for SACos

The other part of the parameters consists of the ones that SACos introduces into the Simulated Annealing algorithm in Subsection 5.2.1 and Subsection 5.2.2 plus the ones introduced by GloVe, coming to a total of 11 extra parameters. Their ranges to construe the Halton sequences can be seen in Figure 11.



Parameters in SACos	Range minimum	Range maximum	Outcome
Cap	0	0.5	0.17
$w_a(a_{INSERT})$	0	0.5	0.017
$w_a(a_{REMOVE})$	0	0.5	0.395
$w_a(a_{TRANSFER})$	0	10	0.918
d_{max}	2	50	38
Type of distance weight function	1	3	1
Parameters for GloVe algorithm	**Range minimum**	**Range maximum**	**Outcome**
Vectors dimensions	2	100	6
Parameters for AdaGrad	**Range minimum**	**Range maximum**	**Outcome**
Iterations	2	100	35
x_{max}	100	1000	633
Learning Rate η	0.01	0.1	0.056
β	0.5	1	0.971

Figure 11: Parameters used for Simulated Annealing old and fine-tuned

The parameters and their ranges are explained below.

The agreeability weight $w_a(a)$
The weight represents how much influence the agreeability should have on the algorithm. The agreeability affects the percentage of moves being accepted. This means that it can potentially significantly affect the acceptance probability of the move as a whole. The different moves did not have the same occurrence probability and therefore the agreeability might affect their acceptance probability to different degrees. Therefore the weighting function is different for the three moves:

1. $\boldsymbol{w_a(a_{INSERT})}$,
2. $\boldsymbol{w_a(a_{REMOVE})}$ and
3. $\boldsymbol{w_a(a_{TRANSFER})}$.
4. **The maximum distance d_{max}**
The maximum distance represented how many neighbouring orders contributed to the agreeability in both directions. The minimum is naturally 1, the maximum is 50 as even the fullest trips hardly consist of over 100 orders.
5. **Type of distance weight function $w_d(i, j)$**
The distance weight function is used on two locations in the algorithm: in the Simulated Annealing part of SACos and in the GloVe part. The authors of GloVe used a fixed function as the distance weighting function, namely option 2. We wanted to see if other functions might improve upon this.
This parameter is an encoding for one of 3 possible functions.

    1: 1

    2: $1/d$

    3: $1/d^2$

6. **The capping point *cap*:**
The capping point at which a Cosine Similarity is deemed to be good. This value cannot reasonably be below 0.5 or above 1.5

A summary of all GloVe parameters:

7. **Vector components**
The number of components of the vectors built from the corpus. Based on Pennington et al. [10]: They used a maximum of 300 components when testing their algorithm. As our data is a lot smaller in size and in richness the maximum for this research was decreased to 100.

* **Type of distance weight function $w_d(i, j)$, for every configuration equal to the parameter at 5**
This refers to the distance weighting function when constructing the co-occurrence matrix.

* **Window size, for every configuration equal to d_{max} the parameter at 4**
The window size is the distance at which the GloVe algorithm looks when counting for the co-occurrence matrix, parallel to the distance SACos when calculating the agreeability.

For AdaGrad:

8. **Iterations**
   The number of iterations/steps gradient descent can train when construing the vectors. Based on the ratio of iterations to number of vector components from the authors of Pennington et al. [10].
9. **Learning Rate η**
   The learning rate it uses. The authors used 0.5, so we used a range around that.

For the co-occurrence weighting function:

$$f(x) = \begin{cases} (x/x_{max})^{\beta} & \text{if } x < x_{max} \\ 1 & \text{otherwise} \end{cases}$$

10. ***x_max***
    There are $\pm$ 1000 solutions to learn from so that is the maximum. 100 is the minimum.
11. $\boldsymbol{\beta}$
    A parameter in the weighting function of GloVe indicating how heavy frequent co-occurrences must count. The authors used 0.75, which they say was a modest improvement over the linear 1. Anything above 1 would achieve the opposite effect and below 0.5 would quickly diminish the influence of the value of X_{kl}.

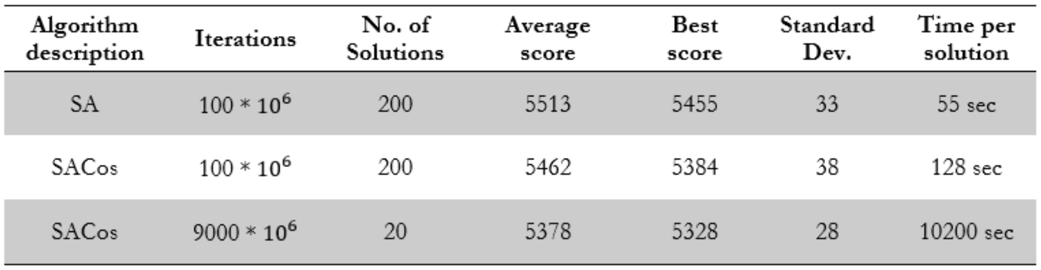
Every parameter configuration was run 20 times, once with each of the first 20 begin solutions of the 200 mentioned in paragraph *A remark* in Subsection 5.3.1. The resulting parameters can be seen in the column *Outcome* in Figure 11.

EXPERIMENTS

6.1 Results

Initially SA and SACos were both run for 100,000,000 iterations, once on each of the stated 200 begin solutions. See Figure 12 for the results. Apart from that, simply to see how good of a score could be achieved, SACos was run 20 times for 9 billion iterations.

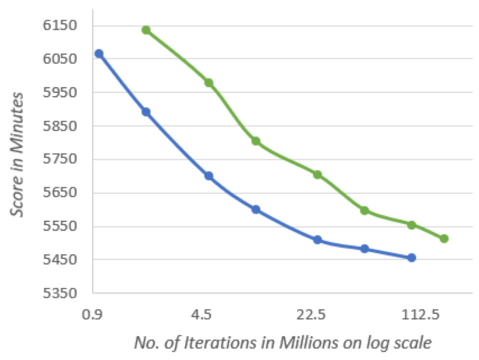
SACos scored on average 51 minutes below the basic version, with a standard deviation of 38, this makes this result more than significant (p-value $3.14 * 10^{-45}$). The best solution overall scored 5328.



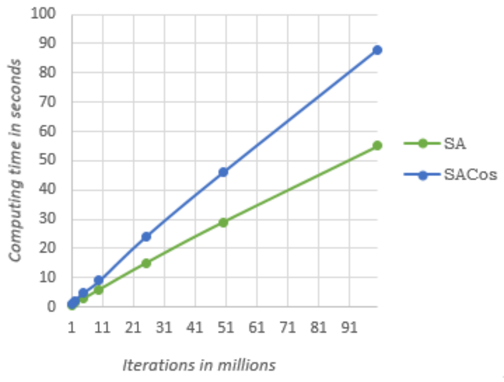
Algorithm description	Iterations	No. of Solutions	Average score	Best score	Standard Dev.	Time per solution
SA	$100 * 10^6$	200	5513	5455	33	55 sec
SACos	$100 * 10^6$	200	5462	5384	38	128 sec
SACos	$9000 * 10^6$	20	5378	5328	28	10200 sec

Figure 12: Initial Results

We let both the algorithms run for a varying number of iterations, while adjusting parameter Q accordingly. See Figure 13.



(a)



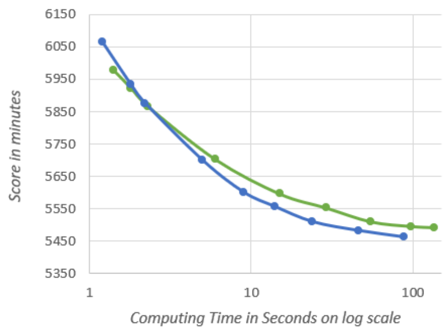
(b)

Figure 13: Scores of the different algorithms

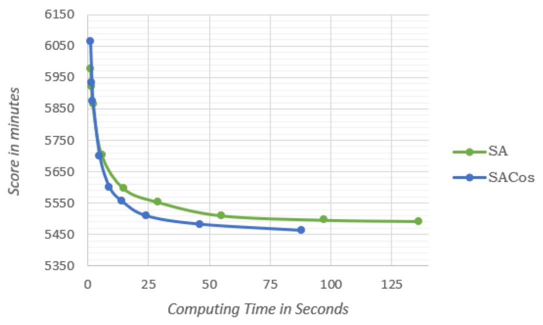
SACOS took a little longer per solution, as you can see in Figure 13b. A Simulated Annealing algorithm will, up to a certain point, perform better the more iterations you let it run. Therefore the question we actually need to pose is how much faster SACOS will yield solutions with the same score.

As can be seen in Figure 14a, SACOS essentially outperforms SA. SA starts out better, but already at a computing time of 1.8 seconds SACOS surpasses it and afterwards only seems to increase its betterment. Times from even better solutions were available, but those runs took so long to finish, that the computers on which they were executed could not be used solely for our purposes. Therefore these times were judged to be inadmissible.

Figure 14b is the same as 14a, but with a linear scale. It shows more clearly that (at least within our chosen computing time domain) the longer both algorithms are executed, the better SACOS performs in comparison.



(a)



(b)

Figure 14: Scores of the different algorithms

6.2 Sensitivity Analysis

We subjected all SACos parameters plus the quality of the start solutions to regression analysis with the data gathered in Subsection 5.4.3.

The most suitable model we found to be a polynomial model with a maximum degree of 2, i.e., a model described by all predictors, their squares, all interactions between pairs of predictors and the intercept, where regression determines coefficients $b...z$:

$$\text{score} = b + c * w_a(a_{INSERT}) + d * w_a(a_{INSERT})^2 + e * w_a(a_{REMOVE}) + f * w_a(a_{REMOVE})^2 +$$
$$g * w_a(a_{INSERT}) * w_a(a_{REMOVE}) + ... + z * x_{max} * \beta$$

The quality of fit of this model, measured by R^2, is 13.89% and unfortunately there are no terms with a significance $p < 0.05$. The most significant term is the interaction between the quality of the start solution and $w_a(a_{REMOVE})$ with $p < 0.15$

See Figure 15 for the scores of the solutions per configuration number (on the X-axis). The configuration number is the Halton number and therefore is a purely categorical value in this graph. The qualities produced by the configuration ranges employed in this research range between 5462 and 5600 minutes, with a standard deviation between 38 and 63 respectively. The figure shows clearly that the standard deviation of the scores of the solution is rather large compared to the difference in scores between configurations. The response plots with different features on the X-axis each look rather similar. Consequently, a poor model was to be expected. There are nevertheless parameter configurations that are consistently better and parameter configurations that are consistently worse.

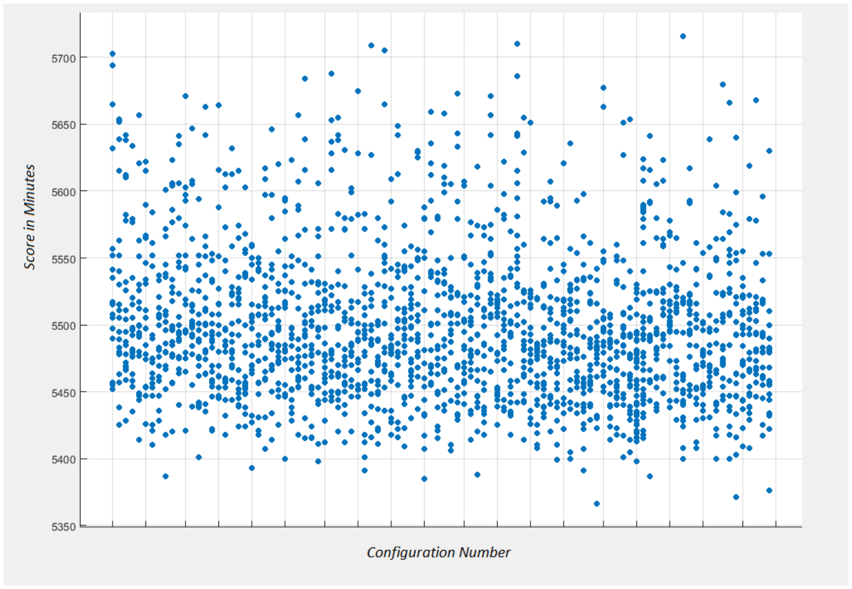


Figure 15: Response plot of solution score per configuration

CONCLUSION

7.1 Conclusion

7.1.1 Summary

In this research an adaptation of Simulated Annealing with the GloVe algorithm, coined SACos, was constructed. Its aim was finding high-quality solutions to a specific instance of the Vehicle Routing Problem, where a solution consisted of a weekly schedule for a garbage truck and the orders it had to fill.

First, a Simulated Annealing implementation (SA) was created in order to generate a thousand reasonably good solutions. Subsequently, GloVe (an unsupervised learning algorithm) trained on these solutions to attain a vector representation for each order. From these vector representations it was possible to calculate the cosine similarity for each pair of orders. A function was defined to translate the cosine similarity into the agreeability. This term could then be inserted into Simulated Annealing to influence the acceptance probability of a new neighbouring solution.

The performances of SA and SACos were initially compared when they had the same stop criterion of 100 million iterations. In this condition SACos beat SA by 61 minutes in score. Because SACos did need a longer running time to achieve this result, for both of the algorithms their solution scores were plotted against their running times. SACos was nevertheless the winner.

7.1.2 Discussion and Future Work

If SACos were to be run longer than has been done here, Figure 14 seems to predict that it would perform even better relatively to SA. A logical explanation for this would be the fact that the agreeability redirects the Simulated Annealing algorithm towards a different area of the search space only slightly, which is an effect that only becomes noticeable when repeated a vast amount of times.

In this research the agreeability was inserted into Simulated Annealing, as a weight manipulating the acceptance probability. One could however easily imagine using it in all kinds of algorithms, in all kinds of ways, as long as solutions resemble sequences. The number itself simply gives some indication as to how well two items in a sequence fit together. It is not dependent on problem-specific requirements. No characteristics or building blocks have to be defined that conform to them. The underlying algorithm or method already takes care of that. This has potential benefits within the same problem instance as well. Take for instance a schedule that has to be recalculated often, with only minor differences such as adding or removing an order. In such cases it would be no problem to run GloVe only once and reuse the obtained cosine similarities for multiple schedules: orders can easily be removed and the new orders get a cosine similarity of 0. The gained improvement would presumably be slightly less, but it would still yield better solutions.

Because of the wide range of possibilities and significant positive results, it would be interesting to see agreeability implemented into more algorithms. Because of its unobtrusiveness, perhaps even in combination with methods that are hybrids already.

REFERENCES

[1] D. Ciresan, A. Giusti, L. M. Gambardella, and J. Schmidhuber. Deep neural networks segment neuronal membranes in electron microscopy images. In *Advances in neural information processing systems*, pages 2843–2851, 2012.

[2] S. de Lima Martins, I. Rosseti, and A. Plastino. Data mining in stochastic local search. *Handbook of Heuristics*, pages 39–87, 2018.

[3] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[4] J. H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Communications of the ACM*, 7(12):701–702, 1964.

[5] M. Kammer, M. Van den Akker, and H. Hoogeveen. Identifying and exploiting commonalities for the job-shop scheduling problem. *Computers & Operations Research*, 38(11):1556–1561, 2011.

[6] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[7] R. Mihalcea, C. Corley, C. Strapparava, et al. Corpus-based and knowledge-based measures of text semantic similarity. In *AAAI*, volume 6, pages 775–780, 2006.

[8] M. J. Norušis. *SPSS 14.0 guide to data analysis*. Prentice Hall Upper Saddle River, NJ, 2006.

[9] T. Papaoikonomou. Jglove. https://github.com/erwtokritos/JGloVe/blob/master/src/glove/GloVe.java, 2015.

[10] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. URL http://www.aclweb.org/anthology/D14-1162.

[11] A. Plastino, R. Fuchshuber, S. d. L. Martins, A. A. Freitas, and S. Salhi. A hybrid data mining metaheuristic for the p-median problem. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 4(3):313–335, 2011.

[12] A. Plastino, H. Barbalho, L. F. M. Santos, R. Fuchshuber, and S. L. Martins. Adaptive and multi-mining versions of the dm-grasp hybrid metaheuristic. *Journal of Heuristics*, 20(1): 39–74, 2014.

[13] M. Raschip, C. Croitoru, and K. Stoffel. Guiding evolutionary search with association rules for solving weighted csps. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 481–488. ACM, 2015.

[14] M. Raschip, C. Croitoru, and K. Stoffel. Using association rules to guide evolutionary search in solving constraint satisfaction. In *Evolutionary Computation (CEC), 2015 IEEE Congress on*, pages 744–750. IEEE, 2015.

[15] M. H. Ribeiro, A. Plastino, and S. L. Martins. Hybridization of grasp metaheuristic with data mining techniques. *Journal of Mathematical Modelling and Algorithms*, 5(1):23–41, 2006.

[16] L. F. Santos, M. H. Ribeiro, A. Plastino, and S. L. Martins. A hybrid grasp with data mining for the maximum diversity problem. In *International Workshop on Hybrid Metaheuristics*, pages 116–127. Springer, 2005.

[17] R. M. Schilham. *Commonalities in local search*. PhD thesis, Technische Universiteit Eindhoven, 2001.

[18] H. Schütze, C. D. Manning, and P. Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press, 2008.

[19] F. Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[20] P. Selim. 2-opt [digital image]. *Wikipedia*, 2009. URL `https://upload.wikimedia.org/wikipedia/commons/b/b8/2-opt_wiki.svg`.

[21] S. Tellex, B. Katz, J. Lin, A. Fernandes, and G. Marton. Quantitative evaluation of passage retrieval algorithms for question answering. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 41–47. ACM, 2003.

[22] Wikipedia - Mean squared error. Mean squared error - wikipedia, 2019. URL `https://en.wikipedia.org/wiki/Mean_squared_error`.

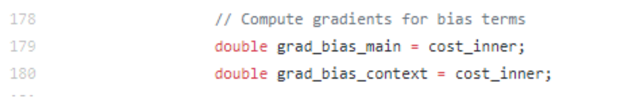
[23] Wikipedia - Vehicle Routing Problem. Vehicle routing problem - wikipedia, 2008. URL `https://en.wikipedia.org/wiki/Vehicle_routing_problem`.

[24] Y. Zhou, J.-K. Hao, and B. Duval. When data mining meets optimization: A case study on the quadratic assignment problem. *arXiv preprint arXiv:1708.05214*, 2017.

APPENDICES

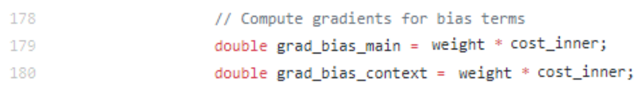
1.1 Adaptation to JGloVe implementation

Adapatation to https://github.com/erwtokritos/JGloVe/blob/master/src/glove/GloVe.java, lines 179 and 180.



```
178                 // Compute gradients for bias terms
179                 double grad_bias_main = cost_inner;
180                 double grad_bias_context = cost_inner;
```

(a) Papaoikonomou [9]



```
178                 // Compute gradients for bias terms
179                 double grad_bias_main =  weight * cost_inner;
180                 double grad_bias_context =  weight * cost_inner;
```

(b) Version from this research

Figure 16: Code excerpts from [9]