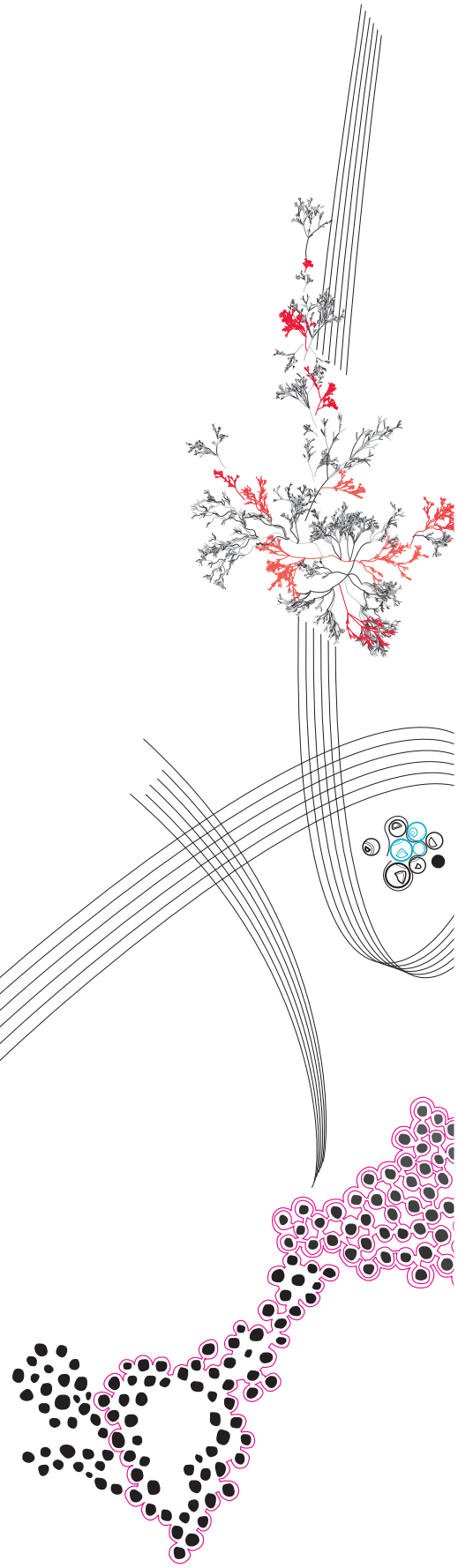

BSc Thesis Applied Mathematics

Improving solutions of Service Engineer Planning using Mixed Integer Programming and Slack Induction by String Removal

Dorian Luchtmeijer

Supervisors: G. Post and T. van Dijk

July, 2022

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Preface

This report is written as part of my Bachelors Assignment Applied Mathematics and Technical Computer Science.

I want to thank my supervisors Gerhard Post and Tom van Dijk for their helpful insights both in implementing the algorithms and in the planning/writing process. I also want to thank them for their flexibility while and after I was unexpectedly not able to work on the project for a week.

Improving solutions of Service Engineer Planning using Mixed Integer Programming and Slack Induction by String Removal

Dorian. L. Luchtmeijer

July, 2022

Abstract

Multi-Depot Capacitated Vehicle Routing Problems with Time Windows are problems where multiple engineers have to visit multiple customers. The challenge is to find routes for the engineers to take. Several aspects must be taken into account: the starting points of the engineers (multi-depot means engineers have different starting points), the capacity of the engineers and the time windows in which customers can be visited. In this report, different ways of improving existing solutions to these problems are analysed. The different methods to find improvements are a mixed integer program, a technique developed by Berghe and Christiaens [1] called Slack Induction be String Removal (SISRs) and a combination of these two. Algorithms were implemented and developed to test these methods on ten instances similar in size, number of engineers and number of customers. Out of the three methods, SISRs seems to be the most promising by far. It was able to improve solutions in little time, and improve solutions significantly given more time.

Keywords: vehicle routing problem, time windows, multi-depot, SISRs, ruin-and-recreate, mixed integer program, improvement

Contents

1 Introduction

Vehicle routing problems are problems in which a number of vehicles must visit a number of nodes. Distances between nodes are given and the goal is to minimize travel costs while optimizing the rewards for visiting certain nodes. For companies whose main expense is travel time or travel distance optimizing solutions for these problems is of high priority. There are multiple kinds of vehicle routing problems. This paper will focus on improving existing solutions for multi-depot capacitated vehicle routing problems with time windows (MDCVRPTW). These problems can be thought of as a number of engineers wanting to visit a number of customers. The top priority is serving as many customers as possible in total, the second priority is to reduce total travel time (excluding possible waiting times for engineers in between visits). Every customer would like to be visited by only one engineer and has indicated a preferred time window in which they would like the engineer to arrive. The engineers start from their home and should return to this home at the end of the schedule. If it is beneficial for the total travel time if some engineers do not leave their home at all then that is allowed. All engineers also have a maximum time they can be working, a capacity.

Solving vehicle routing problems optimally is NP-hard [5], thus solving these problems optimally using exact methods is not viable on a large scale. In this paper, two methods to improve MDCVRPTW solutions are analysed in their ability to improve already 'good' solutions. The first method is using exact methods to solve parts of the problem optimally. This is done by using a mixed integer program where most of the solution is already known. This way, local optima can be calculated and, depending on what part of the given solutions is kept, improvements could be made.

The second method is a heuristic method based on the technique of 'slack induction by string removal' (SISRs) as described by Christiaens and Berghe [1]. They have developed a ruin-and-recreate algorithm that 'destroys' part of the solution and recreates the solution in a possibly different way. The ruin-method of the SISRs technique is based largely on location of customers, a random customer is selected and multiple customers close to this customer will be removed from the existing routes. The recreate method uses 'greedy insertion with blinks': a customer is usually added to the route where the extra travel time because of this customer would be minimal, but such an insertion can be skipped with a certain probability (the blink factor).

Being able to create better and better solutions to vehicle routing problems is of great importance. For delivery companies a decrease in total travel time could almost directly increase their profit, either because they have to work for less time or because they could serve more customers. All instances in which people have to visit multiple places like electricians, plumbers, (home) teachers or gardeners directly benefit from having more efficient routes. The more specific case of multi-depot capacitated VRPs with time windows is also very relevant. For companies scheduling routes for multiple people, multi-depot can be very useful, since this would enable them to create routes starting at everyone's home. Even delivery companies might have multiple stores or storage spaces they could start from. The benefit of taking capacity into account is rather obvious, since workers cannot work forever and vehicles cannot carry an infinite load. With regards to the time windows, customers greatly appreciate having a say in when their delivery arrives or when their new TV will be installed. Including these extra considerations/limitations might increase the

complexity of acquiring a solution, but if done successfully, they are well worth it.

2 Literature review

The research area of this paper is that od Multi Depot Vehicle routing problems, or, more specifically, Multi Dept Capacitated Vehicle Routing Problems with Time Windows. J. R. Montoya-Torres et al. (2015) [7] present hundreds of papers published between 1984 and 2014 on different kinds of MDVRP, including MDCVRPTW. The first well documented heuristic to use for MDVRPTW was Tabu Search by Cordeau et al. (2001) [2], where the main goal was the minimization of vehicles. MDVRPTW with a fixed distribution of vehicles, like in this research, were studied by Polacek, Hartl, Doerner, and Reimann (2005) [8] with a Variable Neighbourhoud Search algorithm. Lim and Wang (2005) [6] researched similar problems, with the addition of having one depot for every engineer, like in this research.

The main article that this research is based on is written by Berghe and Chirstiaens [1] on Slack Induction by String Removal, SISRs for short. They developed a ruin-and-recreate algorithm for several kinds of vehicle routing problems. They described the results of using SISRs on standard VRPs, capacitated VRPs, VRPs with time windows and pickup and delivery problems. SISRs has not been tested and documented in literature on Multi Depot Vehicle Routing Problems with Time Windows.

3 Problem description

The problem consists of n customers and m engineers. The customers would like to be visited by exactly one engineer. All engineers should be home on the same day (meaning within 24 hours of the start of the day). All customers have a preferred time-window to be visited in, and they can only be visited within these time windows.

Provided are the locations (x and y coordinate) and time windows of every customer, as well as a travel time matrix with all travel times between two vertices. Note that, since the travel times are calculated with euclidean distance between two points on a 2D-grid, the graph is complete and undirected: every vertex is connected to every other vertex and the travel time from vertex i to vertex j is the same as the travel time from vertex j to vertex i.

Existing solutions to improve on are also provided, these solutions are already of 'decent' quality. The goal is to improve these solutions. Since this kind of scheduling is a daily occurrence, and one might be required to calculate many of these solutions in a day, the time it takes to create a better solution is taken into account. It is still, however, interesting to see if, with longer running times, solutions could be improved even further. Thus, we have not limited ourselves to running times of only a few minutes per instance, but have also done research into longer running times.

4 Mixed Integer Programming

Using Mixed Integer Linear Programming (MILP) we can solve MDVRPTW optimally (and be sure the resulting solution is optimal). Even though solving the entire problem this way is completely infeasible due to its complexity, it is useful to describe the entire problem mathematically such that it is suitable for MILP. the used variables are defined in Table 1.

TABLE 1: Table of variables

variable	value (if applicable)	description
n	200	number of customers
N	{1, 2, ..., n}	set of customers
m	20	number of engineers
M	{1, 2, ..., m}	set of engineers
V	$N \cup M$	set of vertices
d_{ij}		travel time between vertex i and j.
Q	86400	capacity of engineers
q_i	5400 if $i > m$ else 0	service time of an engineer at a vertex.
a_i		start of time window of vertex i.
b_i		end of time window of vertex i.
s_i		start of service at vertex i.
x_{ijk}	$\in \{0, 1\}$	1 if engineer k travels from i to j, else 0
α	10000	'Value' of visiting a customer

4.1 Goal and Constraints

Next, we need a minimization or maximization goal and several constraints. Our goal is to minimize the total travel time of the engineers while maximizing the number of customers visited. To achieve this we have chosen the following equation that will be minimized, where α, in our case 10000, should be big enough to always make visiting a customer worth it, but small enough to not cause any problems when using computers that might round numbers up or down:

$$-\alpha \sum_{i \in V, j \in N, k \in M} x_{ijk} + \sum_{i,j \in V, k \in M} d_{ij} x_{ijk} \tag{1}$$

The first two constraints ensure that every customer can be visited by at most one engineer. Note that this is only valid when the shortest travel time between every two nodes is known since this makes it impossible for two engineers to use the same customer as a point on the route.

At most one engineer going to a vertex:

$$\sum_{i \in V} \sum_{k \in M} x_{ijk} <= 1 \qquad \forall j \in V \tag{2}$$

At most one engineer going from a vertex:

$$\sum_{j \in V} \sum_{k \in M} x_{ijk} <= 1 \qquad \forall i \in V \tag{3}$$

The following constraint ensures that if an engineer goes to a vertex, they will also go from that vertex to another vertex (or the same). Note that it does not necessarily have to be in this order, since at their depot, the engineer first leaves and later arrives.

$$\sum_{i \in V} x_{ijk} - \sum_{l \in V} x_{jlk} = 0 \qquad \forall j \in V,\ \forall k \in M \tag{4}$$

The next constraints ensure that every engineer departs from and finishes at their own depot. It is possible that for some k, $x_{kkk} = 1$. This is equivalent to an engineer not leaving his/her depot at all.

Depart from own depot:

$$\sum_{j \in N \cup \{k\}} x_{kjk} = 1 \qquad \forall k \in M \tag{5}$$

Ends at own depot:

$$\sum_{j \in N \cup \{k\}} x_{jkk} = 1 \qquad \forall k \in M \tag{6}$$

The last constraints describe the time windows. Constraint (7) states that if $x_{ijk} = 1$ (an engineer travels from i to j), then the start of service of customer j must happen at least the duration of a service plus the travel time from i to j after the start of service at customer i. If $x_ijk = 0$ we need a number big enough such that the equation will always hold. For this we can use Q, the maximum time an engineer can be working. This way, the equation can only not hold when $s_i + q_iQ$, which is not possible since this would mean an engineer would have to work overtime. This constraint also prevents loops not including a depot from forming.

$$s_j + Q \cdot (1 - x_{ijk}) \geq s_i + (x_{ijk} \cdot d_{ij}) + q_i \qquad \forall j \in N,\ \forall i \in V,\ \forall k \in M \tag{7}$$

Constraint (8) simply states that the start of a service for a customer must happen within its time window.

$$a_i \leq s_i \leq b_i \qquad \forall i \in V \tag{8}$$

All engineers start at time 0:

$$s_k = 0 \qquad \forall k \in M \tag{9}$$

The last constraint states that engineers must be back at their depot before the end of their work day.

$$Q \geq s_i + (x_{ikk} \cdot d_{ik}) + q_i \qquad \forall i \in V,\ \forall k \in M \tag{10}$$

4.2 Creating the model

The model was created using CPLEX [4] in python. By adding all aforementioned variables and constraints to a model and using the data from the instances, the mixed integer program can start solving. When creating the model, it becomes clear that adding all constraints described in (7) makes the creation take quite some time: multiple minutes. This is not that surprising since $n \cdot (m+n) \cdot k = 880000$ constraints need to be added here.

Because solving the problem from scratch is impossible with the size of the provided problems, two things are required before we can start solving after the model is created. First we will load the existing solution into the model by adding the constraints $x_{ijk} = 1$ if engineer k travels from customer/depot i to customer/depot j. The next step is destroying a part of the solution.

4.3 Destroying the solution

We have used three different ways to destroy a part of the solution. The first is to remove one route at a time, which ensures that the order of visits within this route will be optimal. It could also transfer customers from the removed route to other routes, but not the other way around. The second option is to remove two routes at a time, which, if doable within respectable time and done for every combination of two routes in an instance, is almost sure to yield improvements. The third way is to use the ruin-method used in SISRs, which removes several customers who are close together from a solution (this is explained in more detail in section 5.1).

5 Slack Induction by String Removal

Slack Induction by String Removal (SISRs) is a ruin and recreate technique. This means that, from an existing solution, a part is removed. After this ruin-phase, the solution will be recreated. In SISRs, this is repeated a number of times. Whether a new solution will be used in the next iteration depends on the quality of the solution and on the 'temperature' of the system. In the earlier iterations the program will allow new solutions that are significantly worse than the previous solution because this turns out to be beneficial for the final solution. With the use of a starting temperature and a cooling down constant, the program will allow the new solution to be less and less worse than the old solution. Before explaining the algorithm some variables need to be defined:

TABLE 2: Table of variables (SISRs)

variable	**standard value (if applicable)**	**description**
T_0	100	starting temperature of the system
T_f	1	final temperature of the system
f		number of iterations
c	$\left(\frac{T_f}{T_0}\right)^{\frac{1}{f}}$	cooling down constant
$\bar{c}$	10	average number of removed customers
s_0		starting solution
γ	0.01	blink factor
L^{max}	10	maximum number of customers to remove from a route
l_s^{max}	$min\{L^{max}, \overline{\lvert t \in T \rvert}\}$	maximum number of customers to remove from a route given solution s
l_t^{max}	$min\{\lvert t \rvert, l_s^{max}\}$	maximum number of customers to remove from route t
l_t	$\lfloor U(1, l_t^{max} + 1) \rfloor$	number of customers to remove from route t
k_s^{max}	$\frac{4 \cdot \bar{c}}{1 + l_s^{max}} - 1$	maximum number of routes to remove customers from given solution s
k_s	$\lfloor U(1, k_s^{max} + 1) \rfloor$	number of routes to remove customers from given solution s

This local search algorithm is well described by Berghe and Christiaens [1] in figure 1.

We can implement the Local Search exactly as described in Figure 1. In line 5 we run the ruin phase to destroy a part of the soluiton. After that we run the recreate-phase to recreate the solution in a possibly different way. In line 6 and 7, it is decided whether the new solution gets used in the next iteration: it is if the solution quality is 'close enough' to the original solution quality. If the new solution is not good enough, the previous solution is used again. Because of randomness in the ruin and recreate phases, using the same solution multiple times can produce different results. Lines 8 and 9 make sure the best solution is stored. In line 10 the system is cooled down to allow less worse solutions.

Algorithm 1 (Local Search Metaheuristic)
Input: Initial solution $s = \{T, A\}$, where T is the set of tours and A the set of absent customers

procedure LOCAL-SEARCH(s)
  $s^{best} \leftarrow s$
  $\mathcal{T} \leftarrow \mathcal{T}_0$
  for f iterations **do**
    $s^* \leftarrow$ SISRs-RUIN-RECREATE(s)
    if $dist(s^*) < dist(s) - \mathcal{T} \ln(U(0,1))$ **then**
      $s \leftarrow s^*$
    if $dist(s^*) < dist(s^{best})$ **then**
      $s^{best} \leftarrow s^*$
    $\mathcal{T} \leftarrow c\,\mathcal{T}$
end procedure

FIGURE 1: Local search algorithm. [1]

5.1 Ruin-phase

Berghe and Christiaens formulated the ruin-phase of the algorithm as follows:

Algorithm 2 (SISRs' Ruin Method)
Input: Solution $s = \{T, A\}$, where T is the set of tours and A the set of absent customers

procedure RUIN(s)
  $l_s^{max}, k_s^{max}, k_s \leftarrow calculate(Equations(5),(6),\text{and }(7))$.
  $c_s^{seed} \leftarrow randomCustomer(s)$
  $R \leftarrow \emptyset$
  for $c \in adj(c_s^{seed})$ **and** $|R| < k_s$ **do**
    if $c \notin A$ **and** $t \notin R$ **then**
      $c_t^* \leftarrow c$
      $l_t^{max},\ l_t \leftarrow calculate(Equations\ (8)\text{ and }(9))$.
      $A \leftarrow A \cup removeSelected(t,\ l_t,\ c_t^*)$
      $R \leftarrow R \cup \{t\}$
end procedure

FIGURE 2: ruin algorithm. [1]

Equations (5), (6), (7), (8) and (9) in Figure 2 reference to the following equations:

$$l_s^{max} = min\{L^{max}, \overline{|t \in T|}\} \tag{5}$$

$$k_s^{max} = \frac{4 \cdot \bar{c}}{1 + l_s^{max}} - 1 \tag{6}$$

$$k_s = \lfloor U(1, k_s^{max} + 1) \rfloor \tag{7}$$

$$l_t^{max} = min\{|t|, l_s^{max}\} \tag{8}$$

$$l_t = \lfloor U(1, l_t^{max} + 1) \rfloor \tag{9}$$

In our implementation'randomCustomer(s)' simply selects a random customer who is visited in the existing solution. 'removeSelected' removes l_t customers from route t, starting from customer c_t^*. It does this in such a way that, given the starting customer, a customer (with equal probability) before or after (in the route) the starting customer is removed. This continues until the number of customers to remove is reached. $adj(c_s^{seed})$ is a list of all customers orders from being the closest to customer c_s^{seed} to being the furthest from customer c_s^{seed}.

5.2 Recreate-phase

Berghe and Christiaens formulated the ruin-phase of the algorithm as follows:

Algorithm 3 (SISRs' Recreate Method)
Input: Ruined solution $s = \{T, A\}$ where T is the set of tours and A the set of absent customers

```
1: procedure RECREATE(s)
2:     sort(A)
3:     for c ∈ A do
4:         P ← NULL
5:         for t ∈ T (which can serve c) do
6:             for P_t in t do
7:                 if U(0,1) <1 − γ then
8:                     if P = NULL or costAt(P_t)<costAt(P)
                           then
9:                         P ← P_t
10:        if P = NULL then
11:            T ← T ∪ {new tour t}
12:            P ← position in t
13:        insert c at P
14:        A ← A \ {c}
15: end procedure
```

FIGURE 3: recreate algorithm. [1]

Berghe and Christiaens suggested a sort algorithm based on random, distance (close to depot first), demand and distance (furthest from depot first). They do this with respective weights 4, 4, 2 and 1. Since in our case there are multiple depots instead, we have chosen to sort on distance of closest depot instead. In our case demand is the same for every customer, thus we remove this from the algorithm.

Another small difference in our program is that creating new routes is not possible, there are always 20 routes. Lines 10 to 12 in figure 3 are therefore removed. Other than this we implemented the recreate-phase in the same way as in figure 3.

6 Results

All implementations of SISRs and Mixed Integer Programming used in this research can be found on GitHub [3].

6.1 Experiment 1

We used 10 instances of 20 engineers and 200 customers. For each of these 10 instances, we have three different existing solutions. The first is the solution after only a construction algorithm is used: routes are created but not optimized in any way. Obviously this solution is the worst quality out of the three. The second solution has been created with a local search algorithm after the construction phase, making the solutions of better quality. The third solutions are created using a collection of algorithms to improve the solution. These kind of solutions are the best quality of the three and are considered of 'good' quality. For the first experiment we ran SISRs on all instances with these three kinds of solutions as starting point.

6.1.1 starting temperature

The temperature of the system is what determines whether worse solutions are momentarily allowed to find improvements in later iterations. Berghe and Christiaens used $T_0 = 100$ as the starting temperature. In our case, however, this seems rather low at first sight because the total travel time in our solutions is measured in seconds and thus of order 10^5. A starting temperature would allow new solutions to be at most 100 seconds worse, an almost negligible number compared to 10^5. Because of this we did Experiment 1 with two different starting temperatures: 100 and 10000.

TABLE 3: Improvements by SISRs on solutions after construction, 10000 iterations, $T_0 = 100$

Instance	**Original**	**Improved**	**percentage decrease**	**running time (minutes)**
1	113725.0	95692.0	15.8567	3
2	142628.0	109854.0	22.9787	2
3	168127.0	-	-	1
4	170528.0	-	-	1
5	126505.0	94867.0	25.0093	3
6	128816.0	99667.0	22.6284	3
7	129762.0	97132.0	25.146	3
8	124629.0	98181.0	21.2214	3
9	135474.0	103528.0	23.5809	2
10	149203.0	101888.0	31.7118	2

6.1.2 findings

In Tables 3 to 8 the results of Experiment 1 are presented. The first thing we notice is that for instance 3 and 4, the one where not every customer could be visited, no improvements were found. Looking at the other instances, for the after construction and after local search starting solutions SISRs is able to find improvements of around 25 and 10 percent respectively. In both of these kinds of starting solutions, we observe that a starting temperature of 10000 is more beneficial (though there could still be exceptions, such as Instance 1 in the solutions after construction). In the good solutions however, this difference is not so clear as we find that $T_0 = 10000$ yields fewer improvements, but two out of the three found improvements are of higher quality than all improvements found when using $T_0 = 100$.
For instance 1, 6 and 7, running SISRs on the solutions after local search yields improved solutions of better quality than the original 'good' solutions. For instance 6 and 7, these are also better than the best solutions SISRs was able to find using the good solutions as starting point.

TABLE 4: Improvements by SISRs on solutions after construction, 10000 iterations, $T_0 = 10000$

Instance	**Original**	**Improved**	**percentage decrease**	**running time (minutes)**
1	113725.0	99652.0	12.3746	3
2	142628.0	108549.0	23.8936	2
3	168127.0	-	-	1
4	170528.0	-	-	1
5	126505.0	89450.0	29.2913	3
6	128816.0	91627.0	28.8699	3
7	129762.0	95641.0	26.2951	3
8	124629.0	91232.0	26.7971	3
9	135474.0	99674.0	26.4257	2
10	149203.0	100896.0	32.3767	2

TABLE 5: Improvements by SISRs on solutions after local search, 10000 iterations, $T_0 = 100$

Instance	**Original**	**Improved**	**percentage decrease**	**running time (minutes)**
1	103849.0	96387.0	7.18543	3
2	126835.0	110033.0	13.2471	2
3	161373.0	-	-	1
4	160378.0	-	-	1
5	105783.0	96201.0	9.05817	2
6	110151.0	101172.0	8.15154	3
7	103383.0	98763.0	4.46882	3
8	104432.0	98524.0	5.65727	3
9	105439.0	104395.0	0.990146	2
10	104904.0	102524.0	2.26874	2

TABLE 6: Improvements by SISRs on solutions after local search, 10000 iterations, $T_0 = 10000$

Instance	Original	Improved	percentage decrease	running time (minutes)
1	103849.0	91088.0	12.288	3
2	126835.0	114937.0	9.38069	2
3	161373.0	-	-	1
4	160378.0	-	-	1
5	105783.0	91579.0	13.4275	2
6	110151.0	90565.0	17.781	3
7	103383.0	90270.0	12.6839	3
8	104432.0	96265.0	7.8204	3
9	105439.0	94536.0	10.3406	2
10	104904.0	97979.0	6.60127	2

TABLE 7: Improvements by SISRs on 'good' solutions, 10000 iterations, $T_0 = 100$

Instance	Original	Improved	percentage decrease	running time (minutes)
1	92348.0	92050.0	0.322692	3
2	100121.0	-	-	2
3	136755.0	-	-	1
4	119205.0	-	-	1
5	89053.0	-	-	2
6	92218.0	91468.0	0.81329	3
7	90639.0	-	-	3
8	90860.0	90619.0	0.265243	3
9	92184.0	91926.0	0.279875	2
10	97554.0	97385.0	0.173237	2

TABLE 8: Improvements by SISRs on 'good' solutions, 10000 iterations, $T_0 = 10000$

Instance	Original	Improved	percentage decrease	running time (minutes)
1	92348.0	90138.0	2.39312	3
2	100121.0	-	-	2
3	136755.0	-	-	1
4	119205.0	-	-	1
5	89053.0	-	-	2
6	92218.0	92099.0	0.129042	3
7	90639.0	-	-	3
8	90860.0	-	-	3
9	92184.0	90977.0	1.30934	2
10	97554.0	-	-	2

6.2 Experiment 2

In Experiment 2, we are interested in the effect of the number of iterations on the performance of SISRs, as well as the ability of SISRs to improve on good solutions. We used both 1000 and 100000 iterations, again both with $T_0 = 100$ and $T_0 = 10000$. Table 9 shows the results of the instances where improvements were found.

Table 9: Improvements by SISRs

Instance	**Original**	**Improved**	**percentage decrease**	**iterations**	**time**	T_0
1	92348.0	91505.0	0.912851	100000	35	10000
1	92348.0	91220.0	1.22147	100000	32	100
1	92348.0	92055.0	0.3172781	1000	<1	100
6	92218.0	91546.0	1.28391	100000	23	10000
6	92218.0	91513.0	1.20801	100000	25	100
6	92218.0	92208.0	0.010843	1000	<1	100
6	92218.0	91821.0	0.43050	1000	<1	10000
8	90860.0	90619.0	0.265243	100000	28	100
8	90860.0	90619.0	0.265243	1000	<1	100
9	92184.0	91457.0	0.78864	100000	22	10000
9	92184.0	91926.0	0.279875	100000	22	100
10	97554.0	94928.0	2.69184	100000	19	10000
10	97554.0	97361.0	0.197839	100000	21	100

We observe that the SISRs technique can consistently improve several instances (1, 6) and can occasionally find an improvement for others (8, 9, 10). More iterations definitely seems to yield better solutions, averaging around 1 percent while 1000 iterations has an average of around 0.3 percent. Using 1000 iterations we also find less improvements. Using 100000 iterations does, however, take a lot of time, and using 10000 iterations like in Experiment 1 might be more beneficial. We also observe that again, instances 3 and 4, the ones were not every customer could be visited, do not have any improvements. It seems highly unlikely that this is purely a coincidence. It seems logical to conclude that SISRs does not work on problems were a significant part of the customers are not planned to get a visit in the existing solution, or at least not in the way we implemented it.

6.3 Experiment 3

In experiment 3, we used Mixed Integer Programming on the good solutions. Destroying one route at a time did not yield any improvements, which means all routes were already optimal in terms of order of visits within the route. Next, we tried removing two routes at a time. This however, required us to limit the computation time of the program, resulting in possibly not-optimal solutions. We ran every combination of two routes of instance 1 with a time limit of 20 seconds, which turned out to be way too short. Next we tried some combinations with a limit of 200 seconds, but this was still insufficient. Finally, we tried removing two routes that we were sure were not optimal because of improvements obtained by SISRs: routes 7 and 12 of instance 1. Even running this for over 5000 seconds did not yield an improvement. Finally, we combined the ruin-method of SISRs with the Mixed Integer Programming. Even this way, with around 10 customers removed at a time, time limits were necessary. Time limits of 200 seconds did not yield any improvements.

7 Conclusions

7.1 Mixed Integer Programming

Neither the mixed integer program, nor a combination of the ruin method of SISRs and the calculations of the mixed integer program resulted in any improvements. This was mostly because of the time it took the mixed integer program to solve even relatively small parts of the problem like 2 out of 20 routes. Running the program for hours on, for example, two routes that we know could be improved because SISRs found improvements, would surely have resulted in an improvement eventually. The problem is that, while SISRs found these improvements often within minutes, even with some foresight as to where the improvements might be it can take the mixed integer program hours to calculate a solution. We conclude that the time it takes to use mixed integer programming, even when used in combination with the ruin phase if the SISRs algorithm, makes this method highly impractical. Even just creating the model takes a few minutes because of the constraints related to the time windows, and that is nothing compared to the time it takes to solve small parts of the problem.

7.2 SISRs

7.2.1 good starting solutions

For creating multiple solutions for different instances, SISRs is a useful algorithm. Within minutes it can improve on existing solutions of good quality. When limited to less than a minute, and thus relatively few iterations (around 1000), the improvements will likely be less than 0.5 percent with existing solutions of similar quality as the ones used in this research. With such a short computation time running SISRs once or multiple times could definitely be worth it. When running times are increased, SISRs produces solutions that improve the old ones by more than 1 percent, sometimes even more than 2. Running the algorithm a few times using 10000 iterations, improvements of 1 to 2 percent can be expected. In our case this would correspond to around 1500 seconds, or 25 minutes a day. This could already save around 10 hours per month. When the goal is not to quickly produce a good solution, but rather to study problems for longer, trying to increase the solutions for longer periods of time, SISRs is very useful. SISRs can produce significant improvements within half an hour and, although not tested within this research, could possibly produce greater results with more iterations (and thus longer running times). A possible disadvantage of SISRs is that its randomness makes it so that running the algorithm multiple times could (and most likely will) produce different results. This could be solved by using random seeds and possibly using a couple different seeds for each instance, but this was not done in this research.

7.2.2 worse starting solutions

SISRs works very well on starting solutions with worse quality. Within minutes it was able to improve very basic solutions with more than 25 percent. What is even more interesting, is that in two of the ten instances, the overall best solution was not found by improving on the best known solutions, but on solutions after just a local search algorithm. This suggests that it might be worth it to also run SISRs on solutions worse than the best. A logical reason behind this occurance, would be that with the algorithms used to improve on the solution after local search, the solution might have gone 'too far down one path'. There might be a better solution that would require too much changes for SISRs to find.

7.3 Future research

Theoretically it should be possible to find improvements using mixed integer programming, and with more research into finding efficient ruin-methods such that the program can solve small parts of the problem optimally this method could be more effective. Using different mixed integer program solvers than CPLEX or different programming languages than python might speed up the computation. Still, exact methods like this will most likely currently be outperformed by heuristic methods, and more research into exact methods does not seem worth it if the goal is to improve the solutions as much as possible. More research into SISRs however, could be very interesting. Problems of different sizes are covered by Berghe and Christiaens [1], but could still be taken into account with regards to time windows or solution quality. It would be worth doing more research into the different variables to find out which values work best specifically for multi depot vehicle routing problems with time windows, and whether these values depend on things like the size of the problem. Finally, looking into a way to adapt SISRs to make it more suitable for instances in which not every customer can be visited could be researched. One simple approach might be to just remove the not-visited customers from the model, but this would mean that if a better solution could include more customers, this would not be found.

References

[1] Berghe G. V. Christiaens, J. Slack induction by string removals for vehicle routing problems. *Transportation Science,*, pages 417–433, 2020.

[2] Jean-François Cordeau, Gilbert Laporte, and Anne Mercier. A unified tabu search heuristic for vehicle routing problems with time windows. *Journal of the Operational research society*, 52(8):928–936, 2001.

[3] DorianIV. Mdcvrp_with_sisrs_and_mip, https://github.com/dorianiv/mdcvrp_with_sisrs_and_mip.

[4] IBM. Ibm ilog cplex optimization studio, https://www.ibm.com/nl-en/products/ilog-cplex-optimization-studio?mhsrc=ibmsearch_amhq=cplex.

[5] J. K. Lenstra and A. H. G. R. Kan. A hybrid swarm intelligence algorithm for vehicle routing problem with time windows,. 8:221–227, 1981.

[6] Andrew Lim and Fan Wang. Multi-depot vehicle routing problem: A one-stage approach. *IEEE transactions on Automation Science and Engineering*, 2(4):397–402, 2005.

[7] Jairo R. Montoya-Torres, Julián López Franco, Santiago Nieto Isaza, Heriberto Felizzola Jiménez, and Nilson Herazo-Padilla. A literature review on the vehicle routing problem with multiple depots. *Computers Industrial Engineering*, 79:115–129, 2015.

[8] Michael Polacek, Richard F Hartl, Karl Doerner, and Marc Reimann. A variable neighborhood search for the multi depot vehicle routing problem with time windows. *Journal of heuristics*, 10(6):613–627, 2004.