B.Com. Dissertation

Multi-Trip Vehicle Routing Problem with Time Windows

By

Nguyen Trong Truong Thanh

Department of Computer Science

School of Computing

National University of Singapore

AY 2020/2021

B.Com. Dissertation

Multi-Trip Vehicle Routing Problem with Time Windows

By

Nguyen Trong Truong Thanh

Department of Computer Science

School of Computing

National University of Singapore

AY 2020/2021

Project No: H0201400

Advisor: Assoc Prof Ooi Wei Tsang

Deliverables:

Report: 1 Volumne

Abstract

This project introduces a new extension of the vehicle routing problem by allowing multiple trips per vehicle and considering the allowable time window at each customer. We coin it multi-trip vehicle routing problem with time windows (MTVPRTW). Comparing to the classical vehicle routing problem, this extension models more precisely real-life operations like postal services, food delivery, school bus routing, etc. In this paper, we aim to study the MTVRPTW in details. We experiment with existing heuristics originally designed to solve related problems in the literature, then come up with a solution algorithm for MTVRPTW. We analyze the proposed solution algorithm with 3 different benchmarks and the results are very promising.

Subject Descriptors:

G.2.1 [Discrete Mathematics]: Combinatorics - Combinatorial algorithms

G.2.2 [Discrete Mathematics]: Graph Theory - Graph algorithms

Keywords:

Travelling salesman problem, Multi-trip vehicle routing problem with time windows, algorithm, heuristic, meta-heuristic, iterated local search

Implementation Software and Hardware:

MacOS 11.0.1, Java 11 SE

Contents

Chapter 1

Introduction

1.1 Contribution

What I have achieved in this project:

- Researched and experimented with existing algorithms for related problems in the literature.
- Proposed new algorithms to solve the Multi-Trip Vehicle Routing Problem with Time Windows.
- Designed a meta-heuristic that outperforms existing benchmarks.
- Analyzed different design choices for the proposed solution algorithm.
- Evaluated the performance of the solution algorithm with regard to different parameters.

1.2 Organization of the report

This report first gives a formal definition of the Multi-Trip Vehicle Routing Problem with Time Windows (MTVRPTW) (chapter 2). Chapter 3 introduces various

algorithms for related problems in the literature like Travelling Salesman Problem (section 3.1), Vehicle Routing Problem (section 3.2) and Vehicle Routing Problem with Time Windows (section 3.3).

Then, the report proposes different algorithms to solve the MTVRPTW (chapter 4). These algorithms are our attempts to tackle MTVRPTW by extending and improving existing heuristics in the literature. After that, chapter 5 describes a meta-heuristic for MTVRPTW. Many algorithms previously discussed in the report are the building blocks for this meta-heuristic.

Finally, computational results are reported for Solomon (1987) [11] benchmark and Gehring & Homberger (1999) [20] benchmark (chapter 6). We also analyze the performance of the meta-heuristic with regard to different parameters (section 6.3).

Chapter 2

Problem definition

The Multi-Trip Vehicle Routing Problem with Time Windows (MTVRPTW) is defined as follows: let $G = (V, E)$ be a weighted complete graph, where V is the set of nodes and E is the set of edges between the nodes. The node set consists of a single depot and a set of customers, all nodes are distributed in a 2D Euclidean space. The edge set represents the travel time between nodes, where each edge (v_i, v_j) is associated with the travel time from v_i to v_j.

Each customer is associated with a demand, a service time and a time window. For the depot, the service time represents the vehicle loading and unloading time, and a time window is also introduced to model a "scheduling horizon" - all routes must start and end within this window. The time window of each customer is hard, meaning that if the vehicle arrives early, it must wait until the time window starts to serve the customer, and the vehicle is not permitted to arrive after the time window (ends). Moreover, each customer must be served by exactly one vehicle, which will prevent a customer being visited multiple time by a vehicle (or different vehicles). By modelling the vehicles as uniform fleet of trucks (vehicles with same parameters like capacity, speed), we can use the same metric for the travelled distance and time. We will only concern with the symmetric version of the problem where $(v_i, v_j) = (v_j, v_i)$,

some modifications are needed if one wants to tackle the asymmetric case.

A vehicle starts from the depot, visits some customers, and comes back to the depot, which constitutes a trip. Each trip must satisfy (1) the vehicle capacity constraint, where the total demand of all customers must not exceed the vehicle capacity, and (2) the time window constraint, where each customer must be served within his time window. In practice, it is often more cost-effective to re-use existing vehicles by making multiple trips rather than introducing new vehicles. Therefore, for the MTVRPTW, we allow each route to consist of multiple trips (the vehicle may come back to the depot and start a new trip many times). This is the distinct characteristic of MTVRPTW compared to other problems in the literature.

The primary objective of MTVRPTW is to minimize the total number of vehicles, and the secondary objective is to minimize the total travelled distance of all vehicles.

In this paper, we represent a solution as a set of vehicles (or routes), each route corresponds to a list of customers (vertices) that the vehicle will visit, including the depot. The solution must satisfy all the capacity and time constraints.

Chapter 3

Literature review

TSP is NP-hard, thus, its extensions including VRP, VRPTW and MTVRPTW are NP-hard, this means that there are no known polynomial time algorithm to solve these problems optimally. Held-Karp algorithm (Held & Karp, 1961) [1] solves the TSP in time $O(|V|^2 * 2^{|V|})$. The VRP can also be solved optimally (Baldacci et al, 2007) [2] with integer linear programming (ILP). However, the exponential runtime (and space) complexities of such algorithms make them unsuitable for large-sized problems. Apart from finding the optimal answer, one could try the approximation approach, with algorithm like Christofides algorithm [3] that gives a 1.5-approximation ratio in $O(|V|^3)$ time for metric TSP. Although better approximation ratio exists (Karlin et al, 2020) [4], such algorithms have more theoretical rather than practical significance. In practice, the most effective line of attack for hard combinatorial optimization problems is search heuristic. Although heuristics might not have a bound to the approximation ratio or runtime, they could find near-optimal solutions in feasible running time. This report would focus mainly on those search heuristics targeted at MTVRPTW.

TSP and VRP are some of the most intensive-studied problems in combinatorial optimization. The multi-trip vehicle routing problem (MTVRP) has also been

actively researched, especially in the last decade (Cattaruzza et al, 2016) [5]. The MTVRPTW is a special case of VRP that incorporates both multi-trip and time window. Campbell & Savelsbergh (2004) [6] propose insertion heuristics to handle vehicle routing and scheduling problems with different types of constraints including time windows and multiple trips, however, the report does not contain any computational results. Due to our knowledge, there are few published papers dealing with the MTVRPTW.

In this section, we will examine possible heuristics to tackle the MTVRPTW. Some heuristics discussed below are derived from papers dealing with TSP, VRP and VRPTW. However, with some modifications, it is possible to utilize those heuristics to solve the MTVRPTW.

3.1 Heuristics for Travelling Salesman Problem (TSP)

TSP is about finding the shortest Hamiltonian cycle in a weighted graph. It could also be understood as a restricted case of MTVRPTW where we are limited to 1 vehicle with unlimited capacity, making single trip, and all customers have really large time windows.

3.1.1 Nearest neighbor

Nearest neighbor is perhaps the simplest and most straightforward heuristic to construct a solution: starting with one customer and stopping only when all customers have been visited. It is greedy in nature: at each step it chooses the un-visited customer that is closest to the current location. Because of its greedy nature, it will always go for immediate gains and miss out on opportunities that will pay out in a

longer term. Nevertheless, due to its simplicity, nearest neighbor can be useful to find an initial solution quickly.

3.1.2 2-opt

One of the simple yet effective heuristics for TSP is 2-opt. This route improvement heuristic is a local search algorithm first proposed by Croes in 1958 [9]. The main idea is to take a route that crosses over itself and reorder it so that it does not (see Figure 3.1).

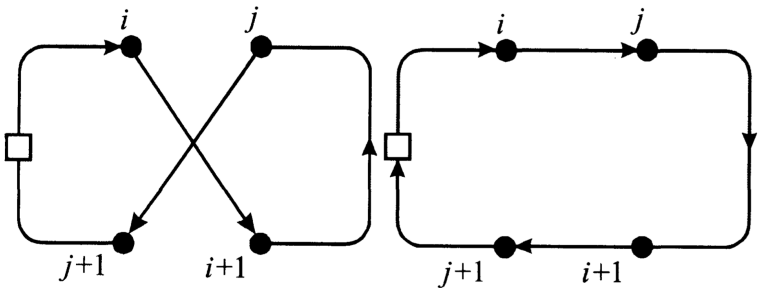

Figure 3.1: 2-opt Operator

Note. The 2-opt operator replaces 2 edges $(i, i+1)$ and $(i, j+1)$ with 2 new edges (i, j) and $(i+1, j+1)$, it also reverses the route from $i+1$ to j.

The 2-opt algorithm works as follows: take 2 arcs from the route, reconnect these arcs with each other and calculate new travelled distance. If this modification has led to a shorter travelled distance then the route is updated. We refer to this process as a *move* or an *operator*. The algorithm continues to build on the improved route, this procedure is repeated until no more improvements are found or until a pre-specified number of iterations is completed (see Algorithm 1).

The 2-opt algorithm is a classic example of local search algorithms, where the algorithm gradually improves an initially given, feasible solution until it reaches a

Algorithm 1 2-opt algorithm

```
1: procedure 2-Opt-Algorithm(r)                          ▷ Input route r
2:     d* ← Compute-Distance(r)
3: For_loop:                          ▷ Iterate all possible exchange positions
4:     for all pair of customers (i, j) do
5:         r' ← 2-Opt-Operator(r, i, j)                  ▷ See Figure 3.1
6:         d ← Compute-Distance(r')
7:         if d < d* then
8:             r ← r'
9:             d* ← d
10:            go to For_loop
11:        end if
12:    end for
13:    return r                                  ▷ Directly modify and return r
14: end procedure
```

local optimum and no more improvements can be made. A complete 2-opt local search will examine every possible valid combination of the swapping mechanism. This technique can be applied to the TSP as well as many related problems. In practice, we can calculate the difference in distance before and after swapping in $O(1)$. Thus, the runtime of a single 2-opt pass is $O(n^2)$, which makes it a fast and effective heuristic that can be used as a sub-procedure in other heuristics.

3.1.3 3-opt and k-opt

The 3-opt operator works in a similar fashion as 2-opt operator, but instead of removing 2 arcs, we remove 3. A 3-opt move can actually be seen as multiple 2-opt moves. Generally speaking, 3-opt algorithm will generate better solution compared to 2-opt, however, with much higher runtime of $O(n^3)$ for each move. We do not necessarily have to stop at 3-opt, we can continue with 4-opt and so on, but each of these will take more and more time and will yield a small improvement on the 2-opt and 3-opt heuristics. Best value k for k-opt could be different depending on the problem instance, Lin & Kernighan (1973) [8] constructed an algorithm that decides which k is the most suitable at each iteration step.

3.1.4 Or-opt

Or-opt is an algorithm proposed by Or (1976) [16]: a chain of consecutive vertices is shifted to other position in the route. This is done by removing 3 edges in the original route with 3 new ones (see Figure 3.2).

We can see that or-opt is a restricted version of 3-opt. Therefore it is faster, but may produce worse routes than 3-opt algorithm. However, or-opt move preserves the route orientation, such characteristic is powerful for problems with time window. Note that here we describe only a single or-opt move, the reader can construct a full

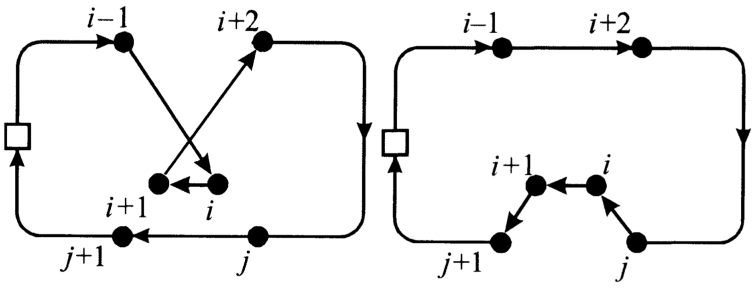

Figure 3.2: Or-opt Operator

Note. The or-opt operator replaces 3 edges $(i-1,\ i)$, $(i+1,\ i+2)$, and $(j,\ j+1)$ with 3 new edges $(i-1,\ i+2)$, $(j,\ i)$, and $(i+1,\ j+1)$, while preserves the route orientation.

or-opt local search algorithm with the technique shown in section 3.1.2.

3.1.5 GENIUS

Gendreau et al. (1992) [10] introduced a route construction heuristic that build the route iteratively, and also a route improvement routine named GENeralized Insertion procedure and Unstringing and Stringing (GENIUS). The route construction procedure (GENI) and the improvement procedure (US) are briefly described below:

Generalized Insertion procedure (GENI)

This route construction procedure starts with a trivial route of 3 vertices and proceeds to add vertex one by one to the route. The main difference between GENI and other procedures is that a new vertex is inserted into the route by incorporating 3-opt and 4-opt optimization moves (see Figure 3.3 and Figure 3.4).

Clearly, running such 3-opt and 4-opt moves with all possible combination takes $O(n^4)$ time, which is undesirable in most cases. To improve the runtime, the GENI procedure only considers p-neighborhood $N_p(v)$ of v, as the set of p vertices on the route closest to the new vertex v. The GENI procedure may now be described:

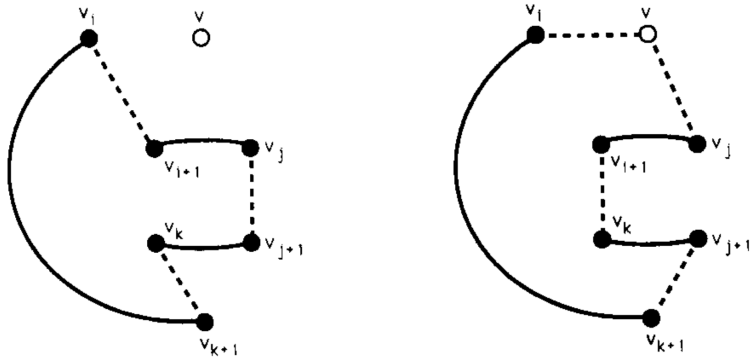

Figure 3.3: GENI insertion procedure with 3-opt move

Note. Customer v is inserted into the route between customers v_i and v_j. 3-opt move is incorporated in choosing v_i, v_j and v_k. Note that 2 paths $(v_{i+1}, ..., v_j)$ and $(v_{j+1}, ..., v_k)$ are reversed.

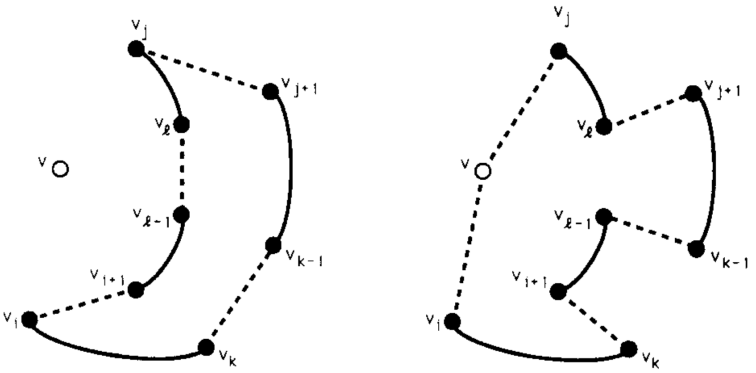

Figure 3.4: GENI insertion procedure with 4-opt move

Note. Customer v is inserted into the route between customers v_i and v_j. 4-opt move is incorporated in choosing v_i, v_j, v_k and v_l. Note that 2 paths $(v_{i+1}, ..., v_{l-1})$ and $(v_l, ..., v_j)$ are reversed.

Step 1. Create an arbitrary starting route of 3 vertices, initialize p-neighbor of all vertices.

Step 2. Arbitrarily select a vertex v not yet on the route and find the best insertion place using 3-opt and 4-opt moves. Update p-neighbor of all vertices.

Step 3. If there is vertex not on the route, go to Step 2

Unstringing and Stringing (US)

In addition to GENI, Gendreau et al. (1992) also proposed a post-optimization procedure to remove and re-insert a vertex in order to improve the route called Unstringing and Stringing (US) (see Figure 3.5). The stringing is identical to the general insertion procedure introduced in GENI, and the unstring is just the reverse.

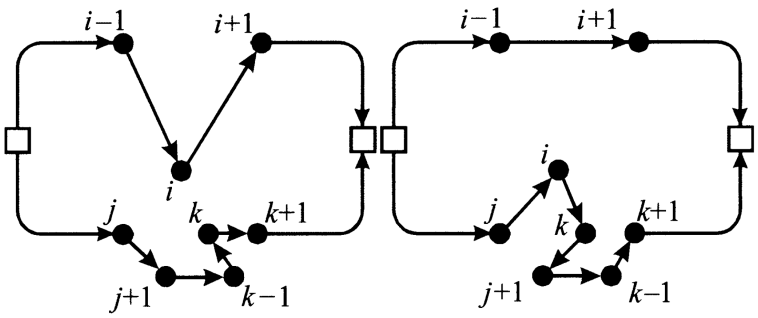

Figure 3.5: US operator

Note. Customer i on the upper route is relocated to the lower route between customers j and k of its closest neighborhood by adding the edges (i, j) and (i, k). Since j and k are not consecutive, the lower route must be reordered.

It's worthwhile to note that the US operator can be used independently of the GENI procedure. To author's empirical testing on TSP instances, applying US process to routes constructed by nearest neighbor actually generated better results comparing to using routes constructed by GENI (GENI then US).

3.2 Heuristics for Vehicle Routing Problem (VRP)

Vehicle Routing Problem (VRP) was first introduced by Dantzig & Ramser (1959) [17]. It extends TSP by allowing multiple vehicles to serve the set of customers. There are 2 main classes of solution improvement heuristics for VRP and its extensions: intra-route and inter-route. Intra-route improvement heuristics are those concern with optimizing the solution by reordering customers within a single route. Most heuristics for TSP can be classified as intra-route heuristics. Inter-route improvement heuristics optimize the solution by considering different routes at the same time.

Savelsbergh (1992) [14] proposed 2 inter-route operators for the classical VRP: relocate and exchange. The relocate operator tries to insert a vertex from one route (*origin*) into another (*destination*) (see Figure 3.6). The exchange heuristic swaps 2 vertices in 2 different routes (see Figure 3.7). Note that here, we only introduce these heuristics as individual operators, details on when to perform such operations depend on the framework that such operators are applied to.

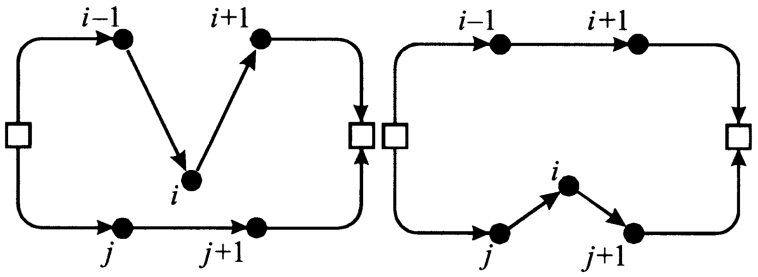

Figure 3.6: Relocate operator

Note. Customer i from the origin route is placed into the destination route between customer j and customer $j+1$. The orientation of both routes remains unchanged.

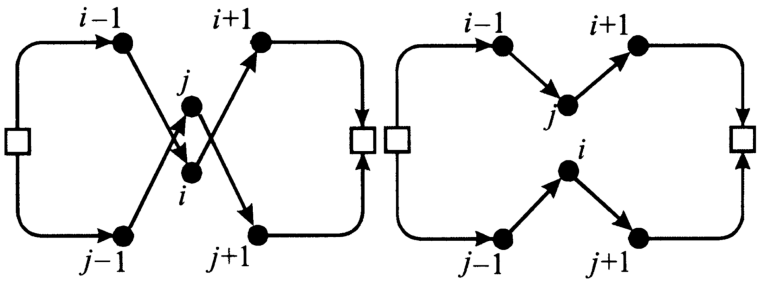

Figure 3.7: Exchange operator

Note. The edges $(i-1,i),(i,i+1),(j-1,j),(j,j+1)$ are replaced by $(i-1,j),(j,i+1),(j-1,i),(i,j+1)$.

3.3 Heuristics for Vehicle Routing Problem with Time Windows (VRP + TW)

Comparing to VRP, VRPTW introduces an important constraint: customer's time window and service time. Actually, VRPTW is a slightly restricted version of MTVRPTW, since MTVRPTW can be reduced to VRPTW by limiting the number of trips for each vehicle to one.

3.3.1 Feasibility check

When we consider the VRPTW, there is an additional level of complexity of allowable delivery time windows. Any modifications should be checked for feasibility, which includes of both vehicle capacity and time window constraints of all customers. Although the feasibility test for vehicle capacity is straightforward if we maintain a list of the current vehicle load up to each customer in the route, checking the time window constraints for all customers in a route can be an expensive $O(n)$ operation. Solomon (1987) [11] proposed the *push-forward* concept to speed this up.

Denote $b_{i_p}^{new}$ as the new time when the service at customer i_p begins, given the insertion of customer u. Let w_{i_r} be the waiting time at i_r for $p \leq r \leq m$. Then the *push-forward* in the schedule at i_p would be:

$$PF_{i_p} = b_{i_p}^{new} - b_{i_p}$$

$$PF_{i_{r+1}} = max\{0,\ PF_{i_r}\} - w_{i_{r+1}}$$

Lemma: The necessary and sufficient conditions for time feasibility when inserting a customer, say u, between i_{p-1} and i_p, $1 \leq p \leq m$, on a partially constructed feasible route $(i_0, i_1, i_2, ..., i_m),\ \ i_0 = i_m = depot$ are:

$$b_u \leq l_u \ \text{ and } \ b_{i_r} + PF_{i_r} \leq l_{i_r},\ p \leq r \leq m$$

where b_u, l_u is the new service time and the latest time window at u, respectively, and PF_{i_r} is the push-forward time at i_r. The detailed proof for this lemma can be found in Solomon (1987) [11].

With the above lemma, when inserting a new customer, we can terminate the time feasibility check early if the either one of the 2 conditions is violated. We can also stop the checking if $PF_{i_r} \leq 0$, since this means there is no push-forward in time at customer i_r and the following customers $i_{r+1},\ i_{r+2}, ...$ are not affected. The details for the feasibility check of inserting a new customer are shown in Algorithm 2.

3.3.2 Solomon's insertion heuristic

Solomon (1987) [11] also proposed a sequential route construction heuristic that iteratively inserts a new customer into the best position in the route. The paper introduced 3 implementations of the insertion heuristic, of which the first (named $I1$) gave the most promising result. This heuristic incorporates both additional time incurred by visiting a new customer and the push-forward in service time of the following customers. The $I1$ insertion heuristic is described below:

Algorithm 2 Feasibility check

1: **procedure** CHECK-INSERTION(u, p, r) ▷ Input customer & position
2: $c \leftarrow$ vehicle capacity
3: $l \leftarrow$ vehicle load in current trip
4: $a_u \leftarrow$ arrival time at u
5: **if** $l + u.demand \leq c \ \vee \ a_u > u.dueTime$ **then**
6: **return** false
7: **end if**
8: $b_{i_p} \leftarrow$ (starting) service time at (customer) i_p
9: $b'_{i_p} \leftarrow$ new service time at i_p
10: $PF_{i_p} \leftarrow b'_{i_p} - b_{i_p}$
11: **for** $r \leftarrow p + 1$ to $r.length - 1$ **do**
12: **if** $PF_{i_{r-1}} = 0$ **then** ▷ Can early terminate
13: **return** true
14: **end if**
15: $a_{i_r} \leftarrow$ arrival time at i_r
16: $b_{i_r} \leftarrow$ service time at i_r ▷ $b_{i_r} = max\{a_{i_r}, \ i_r.readyTime\}$
17: $w_{i_r} \leftarrow max\{0, \ i_r.readyTime - a_{i_r}\}$
18: $PF_{i_r} \leftarrow max\{0, \ PF_{i_{r-1}}\} - w_{i_r}$
19: **if** $b_{i_r} + PF_{i_r} > i_r.dueTime$ **then**
20: **return** false ▷ Violate time window constraint
21: **end if**
22: **end for**
23: **return** true
24: **end procedure**

Step 1. Initialize a route with a "seed" customer s. The seed customer is selected by finding either the geographically farthest un-routed customer in relation to the depot d or the un-routed customer with the lowest allowed starting time for service. The initial route is (d, s, d).

Step 2. While there are customers that can be inserted into the route, insert the new customer u into the route based on 2 subsequently defined criteria.

Step 3. If there are remaining customers not served, add a new vehicle and go to *Step 1.*

Let $(i_0, i_1, i_2, ..., i_m)$ be the current route. For each un-routed customer u, we find the best feasible insertion position $i(u)$, $j(u)$ on the route (for this particular customer) as:

$$c_1(i(u),\ u,\ j(u)) = \min_{\rho=1,...,m}\{c_1(i_{\rho-1},\ u,\ j_\rho)\}$$

The cost function c_1 of inserting u between i and j is calculated as:

$$c_1(i,u,j) = \alpha_1 c_{11}(i,u,j) + \alpha_2 c_{12}(i,u,j),\ \alpha_1 + \alpha_2 = 1,\ \alpha_1 \geq 0,\ \alpha_2 \geq 0$$

$$c_{11}(i,u,j) = d_{iu} + d_{ui} - \mu d_{ij},\ \mu \geq 0,\ d_{kl} \text{ is the travel time between } k \text{ and } l$$

$c_{12}(i,u,j) = b_{ju} - b_j$, b_j and b_{ju} is the old and new time for service at j, respectively

Then, we select the best un-routed customer u^* to be inserted in the route by:

$$c_2(i(u^*), u, j(u^*)) = \max_u\{c_2(i(u), u, j(u))\} \text{ where the route is feasible}$$

$$c_2(i(u), u, j(u)) = \lambda d_{0u} - c_1(i,u,j),\ \lambda \geq 0,\ d_{0u} \text{ is the distance/time to depot}$$

Parameter λ is used to define how much the best insertion place for an un-routed customer depends on its distance from the depot and on the other hand, how much the best place depends on the extra distance required to visit the customer by the current vehicle.

The algorithm runs different sets of configurations based on parameters and initialization criterion, then outputs the best found solution. Details for the algorithm can be found in Solomon (1987) [11].

3.3.3 2-opt*

Potvin & Rousseau (1995) [15] proposed the 2-opt* exchange heuristic for solving VRPTW. The idea of 2-opt* is to replace two arcs by two new arcs, so as to break each route into two different sub-routes, then merge them in order to save travelled distance (see figure 3.8).

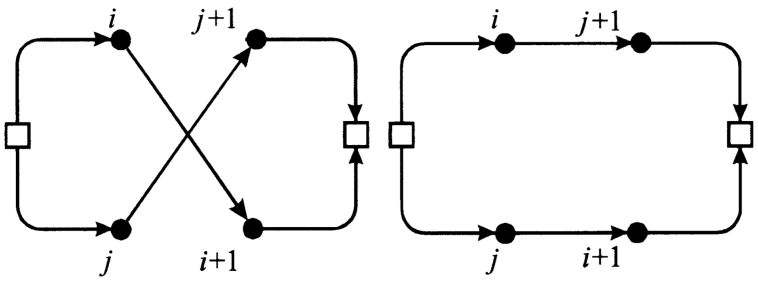

Figure 3.8: 2-opt* Operator

Note. The customers served after customer i on the upper route will be served after customer j on the lower route, and vice versa.

This 2-opt* exchange operator is powerful for problems with time window, because it preserves the orientation of the routes, and introduces the last customers of a given route (i.e. those with late time windows) at the end of the first customers of another route (i.e. those with early time windows). Hence, the operator has high probability of satisfying the time constraint for all customers.

Chapter 4

Heuristics for Multi-Trip Vehicle Routing Problem with Time Windows

Paradiso et al (2020) [7] proposed an exact solution for a restricted version of MTVRPTW where we limit the number of vehicles and try to optimize the travelled distance. The author utilized a new mathematical model with exponential number of variables to formalize MTVRPTW as an Integer Linear Programming (ILP) problem. The computational results show that this ILP algorithm is capable of solving small instances of MTVRPTW. However, such algorithm is not suitable for problem with more than 50 customers. In this section, we will present prospective heuristics for tackling large-scale problem instances of the MTVRPTW.

4.1 Solution construction heuristics

4.1.1 Greedy earliest neighbor

This is a route construction method inspired by the nearest neighbor algorithm for TSP (section 3.1.1). In this case, a new customer is inserted into the route based on the ready time of the vehicle after serving the customer. Note that we need to check if the customer can be inserted into the route without violating any capacity or time constraints before making the insertion. If no feasible customer exists, the vehicle will go back to the depot, then try to find a new customer for the new trip. If there is still no feasible customer, initialize a new vehicle (a new route). This simple and straightforward heuristic could be used to construct initial solution or as a baseline to benchmark other heuristics.

4.1.2 MT-Solomon

Due to the similarity between VRPTW and MTVRPTW, we can extend the $I1$ insertion heuristic by Solomon (1987) [11] to adapt with the multi-trip relaxation of MTVRPTW. We coin it MT-Solomon. The algorithm adds a *Step 2.1* after the original *Step 2* (see section 3.3.2) as below:

Step 2. While there are customers that can be inserted into the route, insert the new customer u into the route based on 2 subsequently defined criteria.

Step 2.1. If no customer can be inserted, append a new depot to the end of the route and got to *Step 2*.

Step 3. If there are remaining customers not served, add a new vehicle and go to *Step 1*.

Appending a dummy depot in *Step 2.1* creates the opportunity for the vehicle to make multiple trips as we reset the vehicle capacity in case the algorithm fails to find feasible customer due to vehicle capacity constraint. Essentially, we exhaust the vehicle by forcing it to make multiple trips before adding a new one.

4.1.3 Cluster - route - merge

Cluster - route - merge is a popular technique in solving the VRP. Here, we tailored it for the routing problems that involve both multiple trips and time windows. The idea is based on an unpublished manuscript from Chang & Wang (2020) [21].

For each cluster, we might select multiple departure times from depot. Thus, the resulting process can be considered as *cluster-route-merge* in *depth-first-search* manner. Details of the algorithm are presented below.

Cluster

The customers are first divided into clusters. A possible way of partition is based on the polar-coordinate angle with the depot (Gillett & Miller, 1974) [12]. Here, we propose clustering customers by their latest service times in order to account for the time window constraint. This increases the chance of a vehicle making multiple trips by serving customers in different clusters, since the vehicle can serve customers in a cluster and come back to the depot before the latest service times of customers in the next cluster.

Our empirical tests show that the number of clusters greatly affects the algorithm's performance, however, the best value depends greatly on the nature of the problem instance. Thus, we propose considering different number of clusters to achieve the best result.

Route

Route construction heuristics can be categorized into sequential and parallel methods. Here, we propose candidate algorithms for both methods:

Sequential route construction

Since each cluster can be treated as an instance of the MTVRPTW, we could re-use previous approaches to solve the sub-MTVRPTW. Both Greedy earliest neighbor and MT-Solomon are applicable in this case.

Parallel route construction

We initialize k routes simultaneously, each route to consist of only the depot and gradually add un-routed customers to the best feasible position in the list of routes. We utilize Solomon (1987) [11] push-forward concept and $I1$ insertion heuristic to check for route feasibility and best position, respectively.

In the parallel route construction, some customers may remain un-routed if we set k too low, or we may introduce more vehicles than needed if k is too high. Therefore, we utilize sequential route construction as a pre-computation step to arrive at a reasonably good value for k (set $k' \leftarrow k - 1$ with k being the value obtained from sequential route construction).

Moreover, in order for a vehicle to make multiple trips, the arrival time at depot of the first trip needs to be earlier than the latest time the vehicle can leave the depot in the second trip (this could be calculated with the latest time window of the first customer in the second trip). Therefore, from the second cluster onward, instead of letting all vehicle leave the depot at time 0, we select some m latest arrival times at depot of the previous cluster as the candidate departure times for the current cluster. As a result, we will need to solve m different instances of the MTVRPTW, in which each instance corresponds to a different departure time. We select multiple

departure times to act as a multi-start strategy to avoid local optima, but limit to only m departure times to save computational time, since the recursive depth-first-search nature of this approach might generate up to $O(m^{n-1})$ MTVRPTW instances, with n being the number of clusters.

Merge

Since we have constructed the routes in the time-ordered hierarchical clusters, we will try to merge the clusters iteratively so that a single vehicle can be re-use in as many clusters as possible. The merging process can be described as follow:

Step 1. Input the route set $\{S_1,\ S_2, ...,\ S_k\}$, where S_i is the set of routes constructed in cluster i.

Step 2. For i in range $(1,\ k)$, merge route sets S_i and S_{i+1} as follow:

Step 2.1. Order the routes in S_i in increasing order of latest arrival time at the depot (since each route could consist of multiple trips, we only concern with the last arrival at the depot). Order the routes in S_{i+1} in increasing order of earliest time to leave depot.

Step 2.2. For each route r_j in S_i, merge r_j with the first *feasible* route in S_{i+1}. If such a feasible merge exists, remove both routes from S_i and S_{i+1} and add the merged route to T_{i+1}. Else add r_j to T_{i+1}.

Step 2.3. Add the remaining routes in S_i and S_{i+1} to T_{i+1}.

Step 2.4. Set S_{i+1} to T_{i+1}

Step 3. Output S_k.

We perform the merge procedure after each route procedure to merge 2 route sets S_i and S_{i+1}. This means that at the second layer of cluster-route-merge in depth-first-search manner, we will obtain m different merged sub-solutions, each containing all customers in the first 2 clusters.

4.2 Local search heuristics for MTVRPTW

In previous chapters, we introduce local search heuristics as individual operators, in other words, how to exchange some vertices, or edges. We now specify when to apply those operations by further describing the algorithms in details. The following 2 local search algorithms for MTVRPTW can be applied to any solutions constructed by the algorithms in section 4.1.

4.2.1 Relocate Algorithm

The primary objective of MTVRPTW is to minimize the total number of vehicles. Based on our observation, reducing the number of vehicles by 1 is already a big improvement step. Indeed, the average vehicle number for Solomon's 56 test cases (Solomon, 1987) [11] differs not more than 1 in many papers [11] [15] [13]. Such reduction in vehicle number can only occur if all customers in 1 route are relocated or inserted into other routes. With that observation, we can design a local search algorithm based on the relocate operator (see Figure 3.6) that directly targets vehicle number reduction. Details are presented in Algorithm 3.

The idea of this algorithm is to select a route r_1 in the solution, then try to relocate all customers of r_1 to other routes. For each customer u in route r_1, we try to find the best feasible insertion position among all other routes. This *best-feasible* criterion allows more customers from route r_1 to be able to insert into route r_2 even if route r_2 has received some customers from r_1. Note that if a customer u is relocated to the end of route r_2 (after the depot), we need to add a depot after u to make the trip legal.

For the cost function, we use the push-forward time because when the time is

Algorithm 3 Relocate algorithm

```
1: procedure RELOCATE-ALGORITHM(s)                          ▷ Input initial solution s
2:     S ← {}                                               ▷ Neighborhood solutions
3:     for r_1 in s do                                      ▷ Iterate all routes in s
4:         s' ← s                                           ▷ Keep the original solution intact
5:         for u in r_1 do                                  ▷ Find best-feasible insertion position for u
6:             c* ← ∞, pos* ← (−∞, −∞)
7:             for r_2 in s', r_1 ≠ r_2 do
8:                 for p_2 ← 0 to r_2.length − 1 do
9:                     if CHECK-INSERTION(u, p_2, r_2) then     ▷ See Algorithm 2
10:                        c ← PUSH-FORWARD(u, p_2, r_2)
11:                        if c < c* then
12:                            c* ← c, pos* ← (p_2, r_2)
13:                        end if
14:                    end if
15:                end for
16:            end for
17:            if c* ≠ ∞ then
18:                RELOCATE-OPERATOR(u, r_1, pos*[0], pos*[1])   ▷ See Figure 3.6
19:            end if
20:        end for
21:        if r_1 is empty then
22:            s' ← s \ {r_1}                               ▷ Remove r_1 from s
23:        end if
24:        S ← S ∪ {s'}
25:    end for
26:    return best solution s* in S
27: end procedure
```

tight (less waiting time at customers or customers are served near deadline), it is difficult to insert new customers (since each new customer will incur both travel time and service time). We do not incorporate a distance component into the cost function because there is no clear relationship in distance between other nodes of route r_1 and nodes of route r_2: inserting/relocating a customer u of r_1 into a "closer" position in route r_2 does not facilitate future insertion/relocation to route r_2, since the new nodes of route r_1 to be examined may lie anywhere in the Euclidean space.

An initial solution with k vehicles will generate k different neighborhood solutions corresponding to each of the k routes being chosen. We only select the best solution in this neighborhood list (*best-acceptance* criterion). Details on how to find the best solution will be discussed later (section 5.3).

4.2.2 Or-opt Algorithm

Or-opt algorithm (Or, 1976) [16] is an intra-route improvement heuristic that optimizes the travelled distance of a route. Or-opt move focuses on a small subset of 3-opt moves that does not reverse any parts of the route (see Figure 3.2). Therefore, or-opt is faster than 3-opt, while still being effective for problems with time constraint. We can implement the or-opt algorithm based on 2 different schemes: *first-feasible* or *best-feasible*. Combining both schemes of the algorithm could help escape local optima. The or-opt algorithm is described in Algorithm 4, 5 and 6.

Or (1976) [16] originally proposed this algorithm to solve TSP, thus, the gain/-cost function is a function of distance. We keep it that way and do not introduce a time component such as *push-forward* time due to its high computational cost. Our empirical tests show that adding the *push-forward* time into the gain/cost function does not improve the final result while incurring significant computational time. We

Algorithm 4 Or-opt algorithm

```
1: procedure OR-OPT-ALGORITHM(s)                    ▷ Input solution s
2:     for all route r in s do
3:         OR-OPT-FIRST-FEASIBLE(r) or OR-OPT-BEST-FEASIBLE(r)
4:     end for
5: end procedure
```

hypothesise that such introduction of time component is ineffective for or-opt algorithm because the or-opt operator changes location of multiple customers, while *push-forward* is a local property, which is more suitable for evaluating insertion/deletion of a single customer.

Note that for both relocate algorithm and or-opt algorithm, we must perform a feasibility check before making a move. This feasibility check includes checking the vehicle capacity and time constraints for all customers in the related routes. The capacity check is an $O(1)$ operation if we maintain a list of current vehicle load in each route, and the time constraint check, although being an $O(n)$ operation, can be speed-up using Solomon's push-forward concept (Solomon, 1987) [11].

For or-opt algorithm, we further optimize the runtime by executing the feasibility check only if gain $g > \varepsilon$ or $g > g^*$. Our empirical tests show that this trick reduces the runtime of the or-opt algorithm by $30 - 70\%$, depending on test cases.

Algorithm 5 Or-opt algorithm (part 2)

```
6: procedure OR-OPT-FIRST-FEASIBLE(r)                          ▷ Input route r
7:     n ← r.length
8:     localOptimal ← false
9: While_loop:
10:    while !localOptimal do
11:        localOptimal ← true
12:        for k ← 1 to 3 do                                    ▷ Segment length
13:            for i ← 0 to n − k − 1 do
14:                x1 ← r[i], x2 ← r[i + 1]
15:                y1 ← r[i + k], y2 ← r[i + k + 1]
16:                for k ← 0 to n − 2 do
17:                    if k ≥ i ∧ k ≤ j then                    ▷
18:                        continue
19:                    end if
20:                    z1 ← r[k], z2 ← r[k + 1]
21:                    g ← ((x1, x2) + (y1, y2) + (z1, z2)) − ((x1, y2) + (z1, x2) + (y1, z2))
22:                    if g > 0 ∧ OR-OPT-FEASIBLE(x1, y1, z1, r) then
23:                        localOptimal ← false
24:                        r ← OR-OPT(x1, y1, z1, r)            ▷ See section 4.2.2
25:                        continue While_loop
26:                    end if
27:                end for
28:            end for
29:        end for
30:    end while
31: end procedure
```

Algorithm 6 Or-opt algorithm (part 3)

32: **procedure** OR-OPT-BEST-FEASIBLE(r) ▷ Input route r
33: $n \leftarrow r.length$
34: $localOptimal \leftarrow true$
35: **for** $k \leftarrow 1$ to 3 **do** ▷ Segment length
36: $g^* \leftarrow -\infty$ ▷ Max gain
37: $r^* \leftarrow r$ ▷ Best route
38: **for** $i \leftarrow 0$ to $n - k - 1$ **do**
39: $x_1 \leftarrow r[i]$, $x_2 \leftarrow r[i + 1]$
40: $y_1 \leftarrow r[i + k]$, $y_2 \leftarrow r[i + k + 1]$
41: **for** $k \leftarrow 0$ to $n - 2$ **do**
42: **if** $k \geq i \wedge k \leq j$ **then** ▷
43: **continue**
44: **end if**
45: $z_1 \leftarrow r[k]$, $z_2 \leftarrow r[k + 1]$
46: $g \leftarrow ((x_1, x_2) + (y_1, y_2) + (z_1, z_2)) - ((x_1, y_2) + (z_1, x_2) + (y_1, z_2))$
47: **if** $g > g^* \wedge$ OR-OPT-FEASIBLE(x_1, y_1, z_1, r) **then**
48: $r' \leftarrow$ OR-OPT(x_1, y_1, z_1, r) ▷ See section 4.2.2
49: $g^* \leftarrow g$
50: $r^* \leftarrow r'$
51: **end if**
52: **end for**
53: **end for**
54: $r \leftarrow r^*$
55: **end for**
56: **end procedure**

Chapter 5

The solution algorithm for Multi-Trip Vehicle Routing Problem with Time Windows

In this chapter, we will present a complete solution algorithm for MTVRPTW. The core of this algorithm is an iterated local search meta-heuristic, after that, a post-optimization procedure is applied (section 5.5).

5.1 A general framework for Iterated Local Search

So far, the algorithms that we have discussed have a common trait: start from a candidate solution obtained from solution construction algorithm, locally optimize it based on some neighbourhood relations, stop after reaching local optima with regard to that neighborhood. Some algorithms are more sophisticated and have larger search space than others, however, they will all stuck at local optima.

A technique to escape local optima for combinatorial optimization problems is iterative improvement of a set of randomly selected feasible solutions (Lin &

Kernighan, 1973) [8]:

Step 1. Generate a random feasible solution, that is, a set S that satisfies C, where C is the set of constraints.

Step 2. Attempt to find an improved solution S' by some transformation of S.

Step 3. If an improved solution is found, i.e., $f(S') < f(S)$, then replace S by S' and repeat from *Step 2*.

Step 4. If no improved solution is found, S is a locally optimum solution. Repeat from *Step 1* until computation time runs out, or the answers are satisfactory.

In this framework, step 2 and 3 can be viewed as intensification step, where we may apply different local search algorithms to transform a solution to a neighbor solution; step 4 is the diversification step, where we simply restarting from a random candidate solution to escape local optimum.

However, restarting from a random solution is not the best diversification technique because the algorithm is throwing away the entire search history and it will take many iterations to reach another local optimum from a randomly constructed solution. In reality, it is good to keep some form of "memory" to guide the future search iterations. This is the general idea of meta-heuristic - heuristic to guide other heuristics.

Iterated Local Search (ILS) is a popular meta-heuristic that has been applied successfully to solve VRP (Penna et al., 2011) [19]. We will use ILS to tackle the MTVRPTW. The general framework for an ILS algorithm can be described as follow:

Step 1. Construct initial solution

Step 2. While termination criterion is not satisfied:

Step 2.1. Intensification: perform subsidiary local search

Step 2.2. Diversification: perturbing the current local optimum

We choose to tackle MTVRPTW with ILS due to its modularity. As shown in previous chapters, there are many candidates for initial solution construction and local search algorithm. Using ILS allows us to write modular code and experiment with a variety of algorithms.

For meta-heuristics, intensification and diversification are the two forces that largely determine the behavior of the algorithm (Blum & Roli, 2003) [18]. ILS is not an exception, therefore, finding balance between intensification and diversification is crucial in designing an effective ILS algorithm.

We will present the choice of initial solution construction heuristic, subsidiary local search, perturbation move as well as how we balance between intensification and diversification for our iterated local search. This ILS meta-heuristic is specifically designed for minimizing the number of vehicles used. At the end of this chapter, we will discuss the technique to optimize the total travelled distance (section 5.5) and give the full algorithm in details (section 5.6).

5.2 Solution construction

We implemented all 3 proposed solution construction heuristics (section 4.1). For the cluster-route-merge heuristic, we set the threshold for number of clusters to 20 and use the sequential route construction for the "route" phase.

While the greedy earliest neighbor is the fastest and most simple heuristic, it produces solutions with mediocre quality (see section 6.3). MT-Solomon and cluster-route-merge have comparable results. Our empirical tests show that using either of

the 2 heuristics yields similar final results for the iterated local search algorithm. Therefore, we decided to use MT-Solomon for initial solution construction due to its simplicity and good performance.

5.3 Subsidiary local search

We employ a combination of or-opt algorithm and relocate algorithm as our subsidiary local search.

We choose the relocate algorithm (section 4.2.1) because it can effectively reduce vehicle number. Each execution of the relocate algorithm will generate multiple solutions. Let the number of vehicles of a solution S be h and the length of the shortest route in S be l, we select the solution with $min(h,\ l)$ as the output. Clearly, a solution with fewer vehicles is better. In case 2 solutions have the same h value, we use l as the tie-breaker. We choose the shortest route length instead of a function of time or travelled distance in order to support subsequent execution of the relocate algorithm. The reason being: in the perturbation step (weak-perturbation, see section 5.4.1), we may modify the set of customers in each route, however, the number of customers in each route remains unchanged. Therefore, the shortest route when selected by subsequent execution of the relocate algorithm is more likely to be "removed completely". Our empirical tests verify this argument - using shortest route length as tie-breaker in relocate algorithm outperforms other methods including total travelled distance and total waiting time.

We also run or-opt algorithm with *best-feasible* criterion before each execution of the relocate algorithm. The reason why we use or-opt - an intra-route improvement heuristic is to optimize individual routes, thus, maximize the possibility of feasible customer insertion into those routes and eventually improve the success rate of sub-

sequent relocate operations.

Due to the nature of the MTVRPTW, the primary objective of reducing the number of vehicles, even if only by 1 is usually difficult to achieve. Thus, we consider such reduction in vehicle number to be a big step, and reset the ILS immediately (set the termination condition to initial state).

5.4 Perturbation

The perturbation step needs to be carefully chosen so that its effect cannot be easily undone by subsequent local search iterations. In other words, the perturbation must be sufficiently strong to guide the search to a different search space and subsequently lead to a different local optimum. However, it should not be too strong, otherwise we would basically obtain a random restart. We address this problem by introducing 2 levels of perturbation: *weak-perturbation* and *strong-perturbation*. The structure for our ILS algorithm for MTVRPTW is as follow:

Step 1. Construct initial solution

Step 2. While termination criterion is not satisfied:

- *Step 2.1.* While strong-perturbation condition not satisfied:
  - *Step 2.1.1.* Perform subsidiary local search
  - *Step 2.1.2.* Perform weak-perturbation
- *Step 2.2.* Perform strong-perturbation

A popular perturbation move for TSP is double-bridge - a special case of 4-opt heuristic. This move is effective for TSP because it cannot be directly reversed by a sequence of 2-opt or 3-opt moves. However, the double-bridge move is not suitable for MTVRPTW because it is an intra-route move. In the intensification step, we

try to relocate all customers from a route to other routes with the relocate algorithm. Using intra-route move as the perturbation means that in the next iteration, we would run the relocate algorithm with the same set of customers in each route, thus, likely to obtain the same local optimum. Moreover, we have already used or-opt algorithm as a pre-processing step to optimize individual routes of the solution before running the relocate algorithm, thus, such intra-route perturbation move is redundant and does not serve the purpose of the perturbation step.

We also experimented with the GENIUS algorithm (the Un-stringing and Stringing procedure - US operator) as an inter-route move for the perturbation step (see figure 3.5). However, this yields minimal improvements. Further analyzing the operator, we can see that it is in fact an extension of the relocate operator using in our subsidiary local search. Indeed, the US operator also tries to insert a customer from a source route to a destination route, the difference is that it inserts the customer (of the source route) between two customers on the destination route that are nearest to it, even if they are not consecutive. Due to such similarity between the US operator and our subsidiary local search, using US operator as the perturbation move may not enable the algorithm to effectively jump to another neighborhood (and therefore not able to escape the local optimum).

5.4.1 Weak-perturbation

After some unsuccessful attempts, we decide to use a sequence of exchange moves as our perturbation step. There are 2 main reasons for this choice: (1) exchange operator cannot be easily undone by the or-opt algorithm or relocate algorithm (the subsidiary local search); (2) exchange operator is likely to be feasible for problem with time windows, thus, we can easily run multiple random exchange operators to move a solution to a different neighborhood. See Algorithm 7 for more details.

Algorithm 7 Weak-perturbation

```
1: procedure WEAK-PERTURBATION(s, t_e)        ▷ t_e is number of exchange moves
2:     c_e ← 0                                 ▷ Count number of exchange moves
3:     c_i ← 0                                 ▷ Count number of iterations
4:     t_i ← 0                                 ▷ Threshold for number of iterations
5:     while c_e < t_e ∧ c_i < t_i do
6:         c_i ← c_i + 1
7:         select 2 random routes r_1, r_2 from s
8:         select 2 random positions p_1, p_2 from r_1, r_2
9:         if EXCHANGE-FEASIBLE(r_1, p_1, r_2, p_2) then    ▷ See Algorithm 8
10:            EXCHANGE(r_1, p_1, r_2, p_2)                 ▷ See Figure 3.7
11:            c_e ← c_e + 1
12:        end if
13:    end while
14: end procedure
```

The function takes an initial solution and the number of exchange moves to be performed as parameters. Because MTVRPTW involves multiple vehicles, each of which contains multiple customers, traditional single exchange move that involves only 2 customers in 2 routes is not strong enough. Therefore, we employ a sequence of exchange moves. The best number of exchange moves may vary depending on the problem's nature. Thus, we use different number of exchange moves t_e values in the ILS algorithm and take the best found solution.

We also introduce the threshold for number of iterations t_i to avoid infinite loop in case a solution does have any feasible exchange moves.

Algorithm 8 Weak-Perturbation (part 2)

15: **procedure** EXCHANGE-FEASIBLE($r_1,\ p_1,\ r_2,\ p_2$)

16: $u_1 \leftarrow r_1[p_1]$

17: $u_2 \leftarrow r_2[p_2]$

18: ${r_1}' \leftarrow$ REMOVE-CUSTOMER($p_1,\ r_1$)

19: ${r_2}' \leftarrow$ REMOVE-CUSTOMER($p_2,\ r_2$)

20: **return** CHECK-INSERTION($u_1,\ p_2,\ r_2$) $\wedge$ CHECK-INSERTION($u_2,\ p_1,\ r_1$)

21: **end procedure**

5.4.2 Strong-perturbation

The purpose of the *strong-perturbation* step is to jump the search to a completely different neighborhood. To do that, we aim to "shuffle" each route as much as possible in order to make each route's set of customers change enough so that the possibility of encountering "promising" routes is high. The 2-opt* move is a good candidate for this purpose because it can change up to 50% the customer set of each route (if each route is broken into equal halves). Moreover, this move is suitable for MTVRPTW because the last customers of a route are appended after the first customers of another route and vice versa, thus it is more likely to pass the time feasibility check. The procedure is illustrated in Algorithm 9.

Note that in Algorithm 9, we use the sum of push-forward time in 2 routes as the cost function in order to make each route as "relaxed" as possible (higher push-forward time at a customer normally means that the following customers will be served nearer to their deadlines). This is to facilitate the execution of the relocate algorithm in subsequent local search iterations.

Algorithm 9 Strong perturbation

```
1: procedure STRONG-PERTURBATION(s)
2:     for routes r_1, r_2 in s, r_1 ≠ r_2 do
3:         c* ← ∞                                              ▷ Min cost
4:         p_1* ← −∞, p_2* ← −∞                                ▷ Best 2-opt* exchange position
5:         for p_1 ← 0 to r_1.length − 1 do
6:             for p_2 ← 0 to r_2.length − 1 do
7:                 if 2-OPT*-FEASIBLE(r_1, p_1, r_2, p_2) then
8:                     c ← PUSH-FORWARD-2-OPT*(r_1, p_1, r_2, p_2)
9:                     if c < c* then
10:                        c* ← c
11:                        p_1* ← p_1
12:                        p_2* ← p_2
13:                    end if
14:                end if
15:            end for
16:        end for
17:        r_1, r_2 ←2-OPT*(r_1, p_1, r_2, p_2)                ▷ See section 3.3.3
18:    end for
19: end procedure
```

5.5 Distance optimization

The previously discussed iterated local search algorithm primarily aims for minimizing the number of vehicles needed. In this section, we will focus on the secondary objective of MTVRPTW: minimizing total travelled distance. As pointed out by Marti (2003) [22], multi-start strategy has been recognized as efficient ways for heuristic search procedures to find global optima to hard combinatorial optimization problems. We employ a multi-start strategy by selecting the local optimal solutions obtained from the previous vehicle number optimization phase with the number of vehicles equivalent to the best found solution. We do this to save computational time, because the or-opt algorithm (first-feasible criterion) with cost function of distance (see Algorithm 5) has been used as part of the subsidiary local search in the iterated local search, thus, the travelled distance of these solutions can be considered partially optimized.

For each initial solution, we perform a combination of or-opt and exchange moves until local optimum is found. We employ both or-opt and exchange algorithms with the aim of incorporating both intra-route and inter-route improvement heuristics, thus, obtaining the best results. The outline of this procedure is illustrated in Algorithm 10.

First, we use or-opt with *best-feasible* criterion (see section 4.2.2). We purposely use a different version of the or-opt algorithm compared to the subsidiary local search in order to avoid going to the same local optima found in the previous phase.

After that, we perform an exchange local search with *first-feasible* criterion, where we accept a move immediately if it is a feasible and improving move. The exchange algorithm continues until reaching local optimum. Our empirical tests show that

Algorithm 10 Distance optimization

```
1: procedure DISTANCE-OPTIMIZATION(s)                    ▷ Input initial solution s
2:     localOptimal ← false
3:     while !localOptimal do
4:         d ← DISTANCE(s)                               ▷ Total travelled distance
5:         s ← OR-OPT-BEST-FEASIBLE(s)                   ▷ See Algorithm 6
6:         s ← EXCHANGE-ALGORITHM(s)                     ▷ See Algorithm 11
7:         if DISTANCE(s) = d then                       ▷ Travelled distance not improve
8:             localOptimal ← true
9:         end if
10:    end while
11: end procedure
```

using *first-feasible* criterion or *best-feasible* criterion for the exchange algorithm gives similar results. Note that this exchange algorithm is different from the random exchange moves in the perturbation step (section 5.4.1). Details for the exchange algorithm are described in Algorithm 11.

5.6 The complete algorithm

In this section, we will present the complete solution algorithm for MTVRPTW (see Algorithm 12, 13). We will also give exact values for the parameters and constants used so that readers can easily reproduce the same numerical results.

For the initial solution construction algorithm MT-Solomon (see section 4.1.2), we use 2 initialization criteria: the farthest un-routed customer and the un-routed customer with the earliest deadline; 6 different parameter sets: $(\lambda, \mu, \alpha_1, \alpha_2) \in \{(1,1,1,0),\ (2,1,1,0),\ (1,1,0,1),\ (2,1,0,1),\ (1,1,0.5,0.5),\ (2,1,0.5,0.5)\}$. In Al-

Algorithm 11 Exchange algorithm

```
1: procedure EXCHANGE-ALGORITHM(s)                          ▷ Input initial solution s
2:     localOptimal ← false
3:     while !localOptimal do
4:         localOptimal ← true
5:         for routes r_1, r_2 in s, r_1 ≠ r_2 do
6:             for p_1 ← 0, r_1.length − 1 do
7:                 for p_2 ← 0, r_2.length − 1 do
8:                     if EXCHANGE-FEASIBLE(r_1, p_1, r_2, p_2) ∧ EXCHANGE-
   GAIN(r_1, p_1, r_2, p_2) > 0 then
9:                         EXCHANGE-OPERATOR(r_1, p_1, r_2, p_2)      ▷ See Figure 3.7
10:                        localOptimal ← false
11:                    end if
12:                end for
13:            end for
14:        end for
15:        if  then
16:            localOptimal ← false
17:        end if
18:    end while
19: end procedure
```

Algorithm 12 Solution algorithm for MTVRPTW

```
1: procedure SOLUTION-ALGORITHM(input)
2:     s ← MT-SOLOMON(input)
3:     S ← {}                                          ▷ Set of local optima
4:     L ← different number of exchanges
5:     for e in L do
6:         S_e ← ITERATED-LOCAL-SEARCH(s, e)           ▷ See Algorithm 13
7:         S ← S ∪ S_e
8:     end for
9:     s_m ← MIN-VEHICLE-NUMBER-SOLUTION(S)
10:    S* ← {s_i ∈ S | VEHICLE-NUMBER(s_i) = VEHICLE-NUMBER(s_m)}
11:    for s in S* do
12:        DISTANCE-OPTIMIZATION(s)
13:    end for
14:    s* ← MIN-TRAVELLED-DISTANCE-SOLUTION(S*)
15:    return s*
16: end procedure
```

Algorithm 13 Solution algorithm for MTVRPTW (cont.)

17: **procedure** ITERATED-LOCAL-SEARCH(s_0, e)
18: $t_i \leftarrow$ threshold for number of iterations
19: $t_w \leftarrow$ threshold for number of weak-perturbation moves
20: $s \leftarrow s_0$, $S \leftarrow \{\}$, $c_i \leftarrow 0$
21: `While_loop:`
22: **while** $c_i < t_i$ **do**
23: $c_w \leftarrow 0$
24: **while** $c_w < t_w \wedge c_i < t_i$ **do**
25: $c_w \leftarrow c_w + 1$, $c_i \leftarrow c_i + 1$
26: OR-OPT-ALGORITHM(s)
27: $S \leftarrow S \cup s$
28: $s' \leftarrow$ RELOCATE-ALGORITHM(s)
29: **if** VEHICLE-NUMBER(s') $<$ VEHICLE-NUMBER(s) **then**
30: $s \leftarrow s'$, $S \leftarrow \{\}$, $c_i \leftarrow 0$
31: **continue** `While_loop`
32: **else**
33: WEAK-PERTURBATION(s', e)
34: $s \leftarrow s'$
35: **end if**
36: **end while**
37: STRONG-PERTURBATION(s)
38: **end while**
39: **return** S
40: **end procedure**

gorithm 12, we set the number of exchange moves $L = \{10, 100\}$. For Algorithm 13, we choose the threshold for number of iterations t_i to be $10,000$ and the threshold for number of weak-perturbation moves t_w to be 100. In Algorithm 7, we set the threshold for number of iterations t_i to $100,000$. Moreover, when a random number is needed, we use Java's Random class with seed 0. In Algorithm 5 and Algorithm 11, when checking if the gain is positive, we use an $\varepsilon = 0.01$ instead of the value 0 in order to avoid rounding errors that might cause infinite loop.

Implementation can be found online at github.com/truongthanh2606/MTVRPTW.

Chapter 6

Computational results

6.1 Choice of benchmarks

To our knowledge, there is no research paper that generates test sets specifically designed for MTVRPTW. However, due to the similarity between VRPTW and MTVRPTW, we propose using Solomon (1987) [11] and Gehring & Homberger (1999) [20] test sets to measure the performance of the solution algorithm for MTVRPTW.

Solomon (1987) originally designed 56 test problems, which are categorized into 6 sets of problems, namely R1, R2, C1, C2, RC1, RC2. Each problem set contains multiple problems with the same number of customers (100), the problem sets differ in the way customers are geographically distributed in the Euclidean space: random uniform distribution (R1 and R2), clustered (C1 and C2) and semi-clustered (RC1 and RC2). Problems sets R1, C1 and RC1 have lower vehicle capacity and shorter scheduling horizon (earlier ending service time at depot), while problem sets R2, C2 and RC2 have higher vehicle capacity and longer scheduling horizon. We only consider the test problems where all of the customers have time windows.

In addition, we use Gehring & Homberger (1999) [20] test set to check the effec-

tiveness of the algorithm in reducing the vehicle number by making multiple trips (having the same constraints with more customers pushes each vehicle to make multiple trips in order to minimize the number of vehicles needed). Gehring & Homberger's test set is structured similarly to Solomon's test set, however, with more customers in each problem: 200, 400, 600, 800 and 1000 customers. Due to limited existing benchmarks in the literature, we will only report the detailed results for the 200 customers test set.

Chang & Wang (2020) [21] proposed a tabu-search algorithm to solve the MTVRPTW and reported the computational results for Solomon (1987)'s 100 customers and Gehring & Homberger (1999)'s 200 customers test sets. Note that Chang & Wang (2020) consider MTVRPTW with the same problem definition and objective as this report. This will be our primary benchmark.

We also use the results reported in Bachem et al. (1996) [23] as the secondary benchmark for Solomon's test set. For Gehring & Homberger's test set, we refer to Bräysy et al. (2004) [24] as the secondary benchmark for our algorithm. Note that Bachem et al. (1996) and Bräysy et al. (2004) consider the vehicle routing problem in which multiple trips are not allowed.

The solution algorithm for all test problems in this paper are implemented in Java on a laptop with a 1.4GHz Quad-Core Intel Core i5 with 8 GB RAM.

6.2 Final results

We report numerical results below. The computational results for the 100 customers, 200 customers test sets and the cumulative results are shown in Table 6.1, Table 6.2 and Table 6.3, respectively. The letters A and B represents the proposed

solution algorithm and Chang & Wang (2020) benchmark. The letter C denotes Bachem et al. (1996) benchmark for the 100 customers test set, and denotes Bräysy et al. (2004) benchmark for the 200 customers test set. Note that Bachem et al. (1996) only reported the numerical results for problem sets R1, R2, RC1, RC2. For the 200 customers test set, we follow Chang & Wang (2020) and do not compute the travelled distance, this is sufficient to compare our proposed algorithm to those benchmarks due to the large difference in the number of vehicles obtained (primary objective). Note that Chang & Wang (2020) only reported 21 test problems for Solomon's test set and 36 test problems for Gehring & Homberger's test set, thus, some entries are left blank.

As shown in the Table 6.1, our proposed algorithm outperforms Bachem et al. (1996) benchmark for 32 out of 39 test problems, and outperforms Chang & Wang (2020) benchmark for 17 out of 21 test problems.

Looking at Table 6.2 - the 200 customers test set, it is clear that the proposed algorithm is effective in terms of reducing vehicle number by making multiple trips. Comparing to Bräysy et al. (2004) benchmark, the proposed algorithm obtains equal or better results for all of the test problems, noticeably in test sets R1 and RC1, where it is able to reduce the average number of vehicles in each test problem by 4.2. The proposed algorithm also achieves equal or better results for 34 out of 36 test problems in Chang & Wang (2020) benchmark, especially it finds better solutions for 9 out of 36 test problems.

The overall results are illustrated in Table 6.3. The table shows that the proposed algorithm is effective in both reducing the vehicle number and the travelled distance. It is particularly powerful in reducing the number of vehicles for problems where multiple trips are possible (Gehring & Homberger (1999)'s 200 customers test set).

Problem	# Vehicles			Travelled distance			Problem	# Vehicles			Travelled distance		
	A	B	C	A	B	C		A	B	C	A	B	C
R101	**19**	**19**	**19**	**1719.4**	**2398.0**	**2398.0**	**R201**	**4**	**4**	**4**	**1518.8**	**1958.0**	**1975.3**
R102	**17**		**17**	**1576.8**		**1991.7**	**R202**	**3**		**3**	**1296.5**		**1446.9**
R103	**13**		**14**	**1381.4**		**1570.1**	**R203**	**3**		**3**	**1061.9**		**1286.2**
R104	**10**		**10**	**1029.4**		**1061.9**	**R204**	**2**		**3**	**884.5**		**962.7**
R105	**14**	**14**	**14**	**1487.6**	**1592.0**	**1596.0**	**R205**	**3**	**3**	**3**	**1123.3**	**1259.0**	**1263.7**
R106	**12**		**12**	**1338.1**		**1356.9**	**R206**	**3**		**3**	**1033.7**		**1094.0**
R107	**10**		**11**	**1209.4**		**1145.3**	**R207**	**2**		**3**	**923.7**		**1026.7**
R108	**10**		**10**	**987.8**		**989.1**	**R208**	**2**		**3**	**806.1**		**825.9**
R109	**11**	**11**	**12**	**1309.4**	**1343.0**	**1262.1**	R209	3	3	2	1077.0	1170.0	1147.2
R110	11	10	11	1175.9	1253.0	1162.9	**R210**	**3**	**3**	**3**	**1066.6**	**1204.0**	**1236.2**
R111	**10**	**10**	**11**	**1173.4**	**1223.0**	**1164.4**	**R211**	**2**	**3**	**3**	**969.9**	**978.0**	**931.2**
R112	10	10	10	1041.3	1059.0	1005.2							
RC101	**14**	**14**	**15**	**1717.1**	**1929.0**	**1927.1**	**RC201**	**4**	**4**	**4**	**1705.1**	**2063.0**	**2073.1**
RC102	**12**		**13**	**1595.5**		**1671.5**	**RC202**	**3**		**4**	**1522.1**		**1767.4**
RC103	**11**		**11**	**1336.6**		**1361.6**	**RC203**	**3**		**3**	**1179.1**		**1407.9**
RC104	10		10	1235.3		1196.2	**RC204**	**3**		**3**	**929.4**		**1065.0**
RC105	**13**	**14**	**15**	**1697.0**	**1867.0**	**1847.0**	**RC205**	**4**	**4**	**4**	**1564.7**	**1955.0**	**1966.2**
RC106	**12**	**12**	**12**	**1401.6**	**1509.0**	**1491.7**	RC206	3	3	3	1542.1	1412.0	1416.6
RC107	11	11	11	1364.9	1359.0	1199.8	RC207	3	3	3	1342.1	1307.0	1322.6
RC108	**10**	**11**	**11**	**1241.8**	**1159.0**	**1198.9**	**RC208**	**3**	**3**	**3**	**1003.8**	**1023.0**	**1029.2**

Table 6.1: Computational results for the 100 customers test set.

Note. We highlight in **bold** test problems that A outperforms both B and C.

Problem	# Vehicles			Problem	# Vehicles			Problem	# Vehicles		
	A	B	C		A	B	C		A	B	C
C101	**20**	**20**	**20**	R101	20	19	20	**RC101**	**18**	**18**	**18**
C102	**18**		**18**	***R102***	***16***		***18***	***RC102***	***15***		***18***
C103	***17***		***18***	***R103***	***14***		***18***	***RC103***	***11***		***18***
C104	***17***		***18***	***R104***	***11***		***18***	***RC104***	***9***		***18***
C105	**20**	**20**	**20**	***R105***	***15***	***16***	***18***	**RC105**	**16**	**16**	**18**
C106	**20**	**20**	**20**	***R106***	***13***		***18***	**RC106**	**16**	**16**	**18**
C107	**20**	**20**	**20**	***R107***	***12***		***18***	**RC107**	**15**	**15**	**18**
C108	**19**	**20**	**19**	***R108***	***10***		***18***	**RC108**	**14**	**14**	**18**
C109	**18**	**18**	**18**	***R109***	***14***	***15***	***18***	***RC109***	***14***	***15***	***18***
C110	18	17	18	***R110***	***12***	***13***	***18***	***RC110***	***13***	***14***	***18***
C201	**6**	**6**	**6**	**R201**	**4**	**4**	**4**	**RC201**	**6**	**6**	**6**
C202	**6**		**6**	**R202**	**4**		**4**	**RC202**	**5**		**5**
C203	**6**		**6**	**R203**	**4**		**4**	**RC203**	**4**		**4**
C204	**6**		**6**	***R204***	***3***		***4***	***RC204***	***3***		***4***
C205	**6**	**6**	**6**	**R205**	**4**	**4**	**4**	**RC205**	**4**	**4**	**4**
C206	**6**	**6**	**6**	***R206***	***3***		***4***	***RC206***	***4***	***5***	***5***
C207	**6**	**6**	**6**	***R207***	***3***		***4***	**RC207**	**4**	**4**	**4**
C208	**6**	**6**	**6**	***R208***	***2***		***4***	**RC208**	**4**	**4**	**4**
C209	**6**	**6**	**6**	***R209***	***3***	***4***	***4***	**RC209**	**4**	**4**	**4**
C210	**6**	**6**	**6**	**R210**	**3**	**3**	**4**	***RC210***	***3***	***4***	***4***

Table 6.2: Computational results for the 200 customers test set.

Note. We highlight in **bold** test problems that A obtains equal or better results than both B and C. We highlight in ***bold-italic*** test problems that A outperforms both B and C.

Test set	Metric	# Vehicles			Travelled distance		
		A	B	C	A	B	C
100 customers	Cumulative (A, C)	296		309	49570.0		53841.4
	Selected sum (A, B, C)	167	169	173	28242.8	31020.0	30614.4
200 customers	Cumulative (A, C)	599		695			
	Selected sum (A, B, C)	387	394	424			

Table 6.3: Overall results.

Note. The selected sum (A, B, C) refers to the cumulative sum of the test problems reported in Chang & Wang (2020).

6.3 Parameters analysis

In this section, we will report the results of various algorithms that we have experimented with to solve MTVRPTW. We also illustrate the impacts of different parameters to the proposed solution algorithm.

The computational results obtained from implementing different solution construction algorithms to MTVRPTW along with our proposed solution algorithm are shown in Figure 6.1. The chart shows that MT-Solomon and cluster-route-merge are very effective in reducing the number of vehicles. Cluster-route-merge heuristic gives slightly better results than MT-Solomon heuristic, however, they are both still far from the final results obtained from implementing the proposed solution algorithm.

Figure 6.2 and 6.3 illustrate the impact of the number of iterations in the iterated local search algorithm to the total number of vehicles and travelled distance. We can see that the proposed algorithm converges to best solutions quickly and is able to find good solutions even with limited computational time.

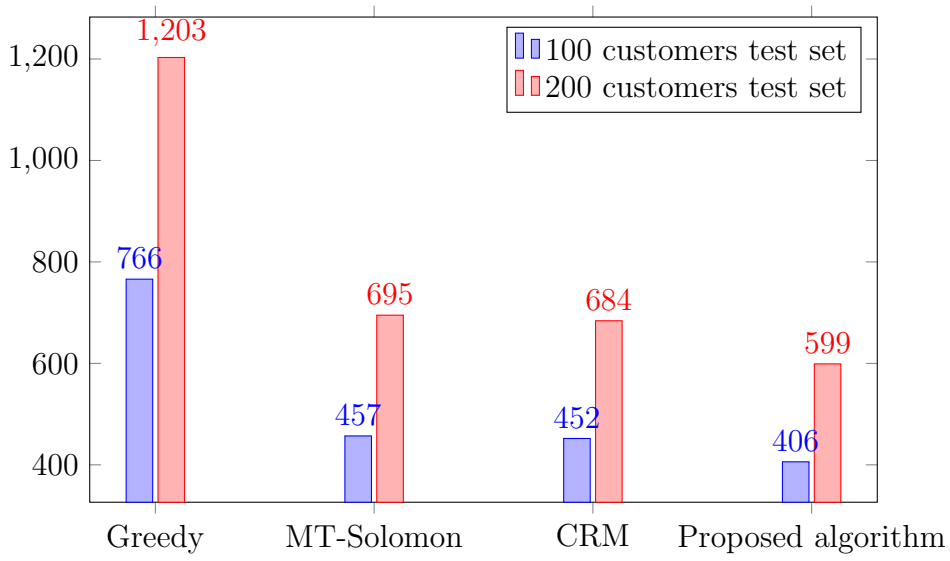

Figure 6.1: Number of vehicles of various solution construction algorithms

Figure 6.4 illustrates the relationship between the number of customers and the overall computational time of the proposed solution algorithm. We can see that the computational time grows rapidly (non-linear) with regard to the number of customers. This is understandable because most of the operators and heuristics used in the proposed algorithm have quadratic and cubic runtime with regard to the customer number.

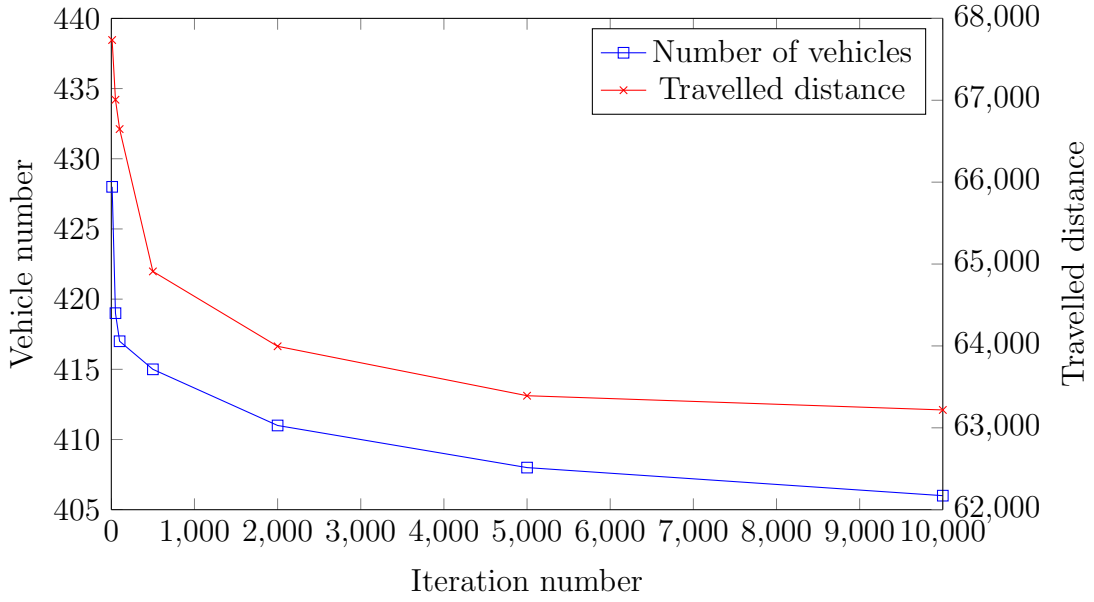

Figure 6.2: ILS iterations number and computational results, 100 customers test set

Note. The results are computed with iteration number $\in \{10, 50, 100, 500, 2,000, 5,000, 10,000\}$

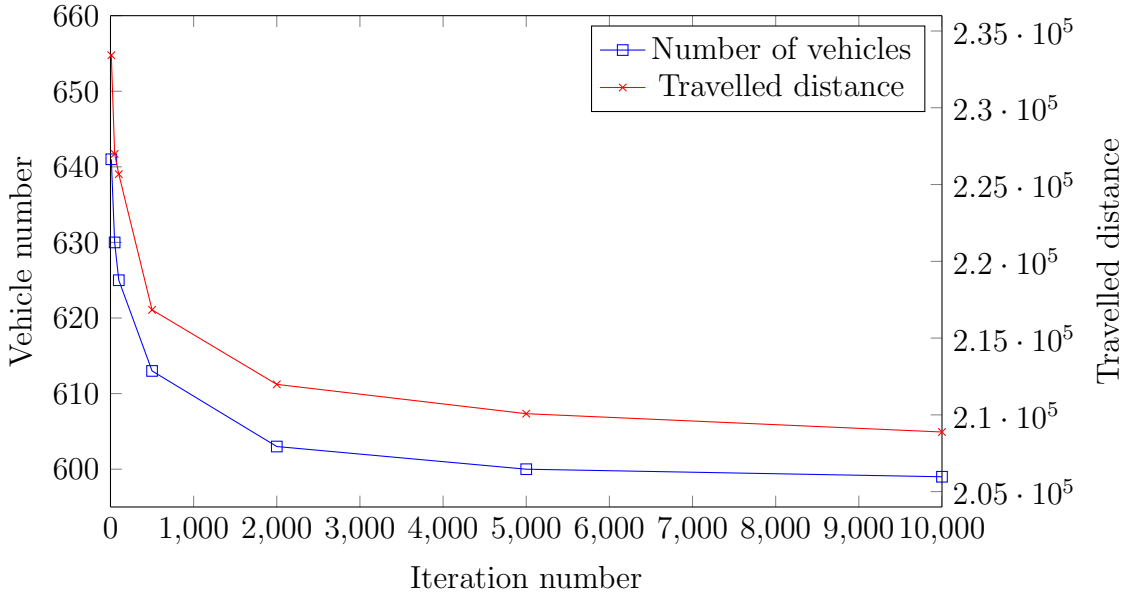

Figure 6.3: ILS iterations number and computational results, 200 customers test set

Note. The results are computed with iteration number $\in \{10, 50, 100, 500, 2,000, 5,000, 10,000\}$

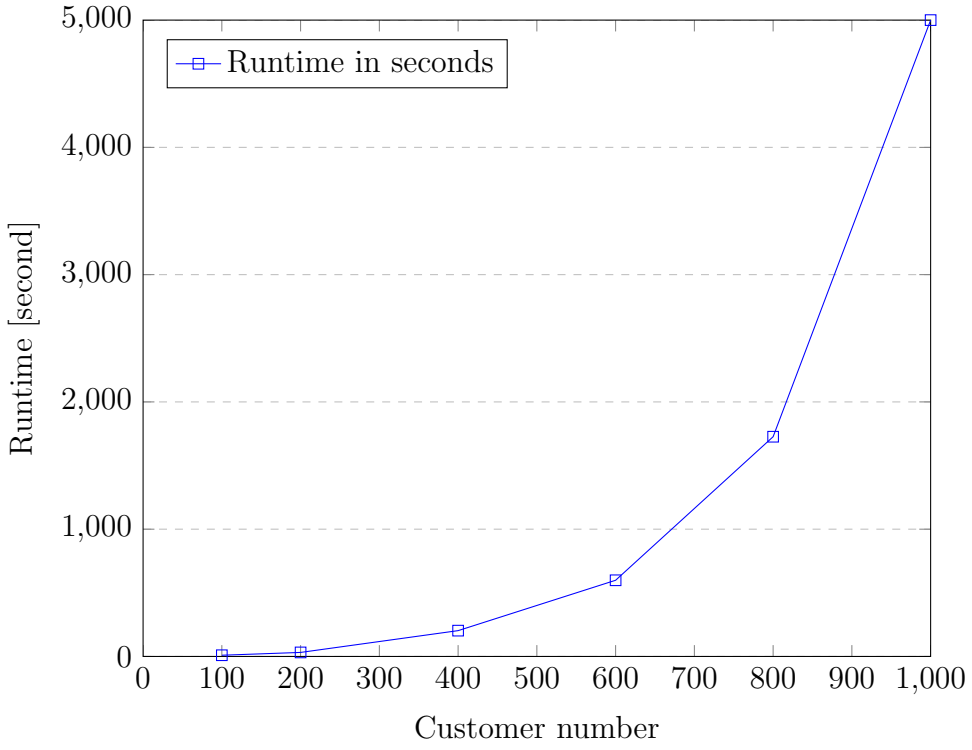

Figure 6.4: Number of customers vs computational time

Note. We set the iteration number for the iterated local search to 100, and only report the computational time for test set C1.

Chapter 7

Conclusion

In conclusion, we have experimented with different heuristics in the literature that originally proposed to solve TSP, VRP and VRPTW. We also extended and modified these heuristics to get various solution construction and local search algorithms for MTVRPTW.

By carefully integrating those algorithms, we were able to design a solution algorithm for MTVRPTW that outperforms all 3 benchmarks: Bachem et al. (1996) [23], Gehring & Homberger (1999) [20] and Chang & Wang (2020) [21].

We also analyzed different design choices that we had made in implementing the proposed algorithm: incorporating time vs distance element into cost function, first-feasible vs best-feasible acceptance criteria, inter-route vs intra-route improvement heuristics, intensification vs diversification.

Due to time limit, we were unable to explore some directions for the project: other meta-heuristics like tabu-search, which has been proven to be powerful in tackling VRPTW; other candidates for local search move and perturbation for the iterated local search, to exemplify, cross-operator (Savelsbergh, 1992) [14], λ-interchange

(Osman, 1993) [26], ejection chains (Glover, 1992) [27]; beyond local search and meta-heuristics - reinforcement learning (Bello et al., 2017) [25]. These are possible directions to further extend the work done in this report.

Bibliography

[1] Held, M. & Karp, R. M. (1961). *A dynamic programming approach to sequencing problems*. Proceedings of the 1961 16th ACM National Meeting. doi:10.1145/800029.808532

[2] Baldacci, R., Christofides, N., & Mingozzi, A. (2007). *An exact algorithm for the vehicle routing problem based on the set partitioning formulation with additional cuts*. Mathematical Programming, 115(2), 351-385. doi:10.1007/s10107-007-0178-5

[3] N. Christofides, *Worst-case analysis of a new heuristic for the travelling salesman problem*, Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[4] Karlin, A. R., Klein, N., & Gharan, S. O. (2020). *A (Slightly) Improved Approximation Algorithm for Metric TSP.*

[5] Cattaruzza, D., Absi, N., & Feillet, D. (2016). *Vehicle routing problems with multiple trips*. 4Or, 14(3), 223-259. doi:10.1007/s10288-016-0306-2

[6] Campbell, A. M., & Savelsbergh, M. W. (2004). *A Decomposition Approach for the Inventory-Routing Problem*. Transportation Science, 38(4), 488-502. doi:10.1287/trsc.1030.0054

[7] Paradiso, R., Roberti, R., Laganá, D., & Dullaert, W. (2020). *An Exact Solution Framework for Multitrip Vehicle-Routing Problems with Time Windows*. Operations Research, 68(1), 180-198. doi:10.1287/opre.2019.1874

[8] Lin, S., & Kernighan, B. W. (1973). *An Effective Heuristic Algorithm for the Traveling-Salesman Problem*. Operations Research, 21(2), 498-516. doi:10.1287/opre.21.2.498

[9] Croes, G. A. (1958). *A Method for Solving Traveling-Salesman Problems*. Operations Research, 6(6), 791-812. doi:10.1287/opre.6.6.791

[10] Gendreau, M., Hertz, A., & Laporte, G. (1992). *New Insertion and Post-optimization Procedures for the Traveling Salesman Problem*. Operations Research, 40(6), 1086-1094. doi:10.1287/opre.40.6.1086

[11] Solomon, M. M. (1987). *Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints*. Operations Research, 35(2), 254-265. doi:10.1287/opre.35.2.254

[12] Gillett, B. E., & Miller, L. R. (1974). *A Heuristic Algorithm for the Vehicle-Dispatch Problem*. Operations Research, 22(2), 340-349. doi:10.1287/opre.22.2.340

[13] Kilby, P., Prosser, P., & Shaw, P. (1999). Guided local search for the vehicle routing problem with time windows. Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization, 473-486. doi:10.1007/978-1-4615-5775-3_32

[14] Savelsbergh, M. W. (1992). The vehicle routing problem with Time WINDOWS: Minimizing Route Duration. ORSA Journal on Computing, 4(2), 146-154. doi:10.1287/ijoc.4.2.146

[15] Potvin, J., & Rousseau, J. (1995). An exchange heuristic for routeing problems with time windows. Journal of the Operational Research Society, 46(12), 1433-1446. doi:10.1057/jors.1995.204

[16] Or, I. 1976. Traveling salesman-type combinatorial problems and their relation to the logistics of regional blood banking. Ph.D. thesis, Northwestern University, Evanston, IL.

[17] Dantzig, G. B., & Ramser, J. H. (1959). The truck dispatching problem. Management Science, 6(1), 80-91. doi:10.1287/mnsc.6.1.80

[18] Blum, C., & Roli, A. (2003). Meta-heuristics in combinatorial optimization. ACM Computing Surveys, 35(3), 268-308. doi:10.1145/937503.937505

[19] Penna, P. H., Subramanian, A., & Ochi, L. S. (2011). An iterated local search heuristic for the HETEROGENEOUS fleet vehicle routing problem. Journal of Heuristics, 19(2), 201-232. doi:10.1007/s10732-011-9186-y

[20] Gehring, H., J. Homberger. A parallel hybrid evolutionary meta-heuristic for the vehicle routing problem with time windows. Proceedings of EUROGEN99, University of Jyvaskya, Javaskyla, Finland, 57-64, 1999

[21] Chang, T. S., & Wang, S. D. (2020). Multi-Trip Vehicle Routing Problems with Time Windows. Unpublished manuscript.

[22] Martí, R., "Multi-start methods In: Fred Glover and Gary A. Kochenberger (eds.) Handbook of Meta-heuristics pp. 355-368, Kluwer Academic Publishers, 2003

[23] Bachem, A., W. Hochstattler, M. Malich, "The simulated trading heuristic for solving vehicle routing problem", Discrete Applied Mathematics, Vol. 45, pp47-52, 1996

[24] Bräysy, O., Hasle, G., & Dullaert, W. (2004). A multi-start local search algorithm for the vehicle routing problem with time windows. European Journal of Operational Research, 159(3), 586-605. doi:10.1016/s0377-2217(03)00435-1

[25] Bello, I., Pham, H., Le, Q. V., Norouzi, M., & Bengio, S. (2017). Neural Combinatorial Optimization with Reinforcement Learning. Retrieved April 02, 2021, from https://openreview.net/pdf?id=Bk9mxlSFx

[26] Osman I.H. (1993). Metastrategy Simulated Annealing and Tabu Search Algorithms for the Vehicle Routing Problems. Annals of Operations Research 41, 421-452.

[27] Glover F. (1992). New Ejection Chain and Alternating Path Methods for Traveling Salesman Problems. In: Balci O., Sharda R. and Zenios S. (eds.), Computer Science and Operations Research: New Developments in Their Interfaces, Pergamon Press, Oxford, 449-509.