A TABU SEARCH ALGORITHM FOR THE VEHICLE ROUTING PROBLEM WITH TIME WINDOWS

Honours Project in Statistical Sciences

University of Cape Town

November 27, 2015

Written by Saarah Parker
Supervised by Dr Jonas Stray

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

Acknowledgments

The following research project would not have been possible without the guidance, support and encouragement from the author's supervisor, Dr Jonas Stray.

Furthermore, the project was partially funded by the DST-NRF Centre of Excellence in Mathematical and Statistical Sciences (CoE-MaSS). Opinions expressed and conclusions arrived at are those of the author and are not necessarily to be attributed to the CoE-MaSS.

Declaration

By submitting this thesis electronically, the author hereby declares that the work on which this study is based is their original work (except where acknowledgements indicate otherwise) to the best of their knowledge.

The study is being submitted for the degree of Bachelor of Business Science specialising in the field of Analytics at the University of Cape Town.

Signed by candidate

TABLE OF CONTENTS

1 INTRODUCTION

The vehicle routing problem (VRP) is an important combinatorial problem, central in distribution management (Laporte & Cordeau 2002; Cordeau et al. 2002; Braysy & Gendreau 2005), where combinatorial problems involve finding an optimal solution within a finite set of objects. Logistics, financial planning, transportation, production, and scheduling are simply a few of the application areas of the VRP. Given the problem's occurrence in everyday situations, attempts to solve the problem have resulted in substantial savings in costs for various industries (Laporte & Cordeau 2002).

The following chapter provides a brief overview of the background of the VRP and its variants as well as a description of the purpose of this investigation, that is, to create a simple tabu search algorithm in attempt to solve a variant of the VRP involving time windows. Furthermore, the objectives, scope and limitations of this study are stated, followed by a layout of the remainder of the paper.

1.1 BACKGROUND

Consider a network of traversable links joining a transport vehicle depot with a set of customers. The VRP is an optimization problem such that routes between some central point, for example the depot, and scattered locations, for example the customers, are designed in a manner that brings about minimal costs (Braysy & Gendreau 2005). The customers must be visited exactly once by one vehicle, each starting and ending at this common central location. Finally, the transport vehicles have a limited capacity and each customer has a specific requirement.

Frequently, locations can only be visited within given time windows, this is known as the VRP with time windows (VRPTW). Time windows are a natural occurrence in business activity, especially if they operate on fixed time schedules, for example, bank deliveries, postal services, dial-a-ride, transport routing and scheduling, and industrial refuse collection (Solomon 1987). Given that the VRPTW is a common phenomenon in distribution systems, the problem is a relevant issue in day to day operations (Bräysy & Gendreau 2002).

Consider for example that each location is a customer to whom a delivery must be made within a specific time frame, with the added condition that each customer has a specific demand and that the total sum of these demands amongst all the customers in a single route cannot exceed a given vehicle's capacity. The cost which must be minimized could then be seen as total travelling time or total travelling distance for a fixed number of vehicles. The VRPTW is involved with delegating vehicles to customers so as to minimise the travelling distance of each vehicle and often also minimising the number of vehicles required to service a set of customers' requirements within the given time windows, all the while adhering to the vehicles' capacity constraints. In some cases, an additional total route length or duration constraint is also implemented.

1.2 Problem Description

There are a number of approaches to solving the VRPTW, some of which are metaheuristics. These deviate from classical approaches as they allow for intermediate solutions which are either infeasible or worse than the previous solution (Braysy & Gendreau 2005). A solution space can be seen as all the possible routes that are available, and the purpose would be to find the optimal route in terms of minimizing costs. One of the most successful metaheuristic approaches to the VRPTW is the tabu search algorithm.

Tabu search is an iterative search process. An initial solution is needed to start the procedure, typically it is created with some cheapest insertion heuristic (details of which are covered at a later stage). This initial solution is recorded as the best solution as well as the current solution. Attempts to improve the initial solution by altering it, using local search and neighbourhood structures, are made. This alteration is called a 'move' and refers to the process of transitioning to a new solution (Toth & Vigo 2003). Neighbourhood structure refers to how moves are modified to create new ones, thus all potential moves that can be made for a given solution is known as that solution's neighbourhood, which is effectively a subset of the entire solution space. A best-accept strategy is used when searching a neighbourhood, that is, the entire subset of solutions is considered and the best element is accepted as the new current solution. If the current solution is better than the best solution, the best solution is also updated. This process then continues for a specified number of iterations or until a desired solution is found.

The justification for allowing deteriorating or infeasible solutions at each iteration is so that a greater portion of the solution space is explored instead of simply settling at a local optimum. To avoid cycling and to ensure that new paths are investigated, recently explored solutions are added to what is known as a 'tabu list'. Solutions remain in this list for a pre-specified duration known as the 'tabu tenure'. Alternatively an 'aspiration criterion' can be used where a solution's tabu status can be removed if, for example, the tabu solution is an improvement of the best solution so far, also known as the 'incumbent' (Braysy & Gendreau 2005).

In summary, the tabu search algorithm attempts to make the best possible move based on available choices conditioned by certain constraints. These constraints are put in place to prevent repetition by making certain moves forbidden to free the search from local optimality but still producing good quality moves at each step. The restrictions are subject to change depending on the iteration and some aspiration criteria.

The latest implementations of tabu search are cumbersome, especially as a result of using many user-controlled parameters. There is a need for a simpler design with more flexible code, easily adaptable neighbourhood structures and a few dynamic parameters. The aim of this investigation therefore is to construct a basic tabu search algorithm.

1.3 Objectives

The first objective was to conduct a literature review on the use of tabu search for the VRPTW. The second objective was to design and use a tabu search algorithm with the aim of finding the lowest possible number of routes and total distance travelled for a fixed number of

vehicles. The algorithm was then tested on a subset of the so called Solomon 100 dataset and its performance with respect to solution quality was compared to the best known solutions. All coding was conducted within the optimisation package AIMMS.

1.4 SCOPE AND LIMITATIONS

With regard to the literature review, it is important to note that most solution approaches only address the spatial aspects of the VRP and not the additional temporal aspects that are relevant when solving the VRPTW (Solomon 1987). Hence some of the approaches mentioned are limited in their effectiveness and adaptation is necessary to account for the additional time window constraints in some of the VRP variants.

With regard to the methodology, parallel approaches are beyond the scope of this paper as a search is run on more than one device, meaning multiple computing equipment are needed for those methods. It is also difficult evaluate the performance of parallel implementations given the inaccuracies of recording certain results such as computing times. Thus, some popular but sophisticated neighbourhood structures, such as ejection chains for example, were not considered when constructing the basic tabu search algorithm for this study.

There are limitations with regard to comparing various methods. Given the variability in results recorded by the numerous authors, who often do not mention whether parallel or sequential techniques were used, measurement issues surface (Laporte et al. 2000). The use of post-optimisation procedures are also unclear at times.

Some stricter practices are suggested by the literature (Laporte et al. 2000). Specifically, it is suggested that parameter settings should be on a different set of instances from those used for final tests. Standard results should be reported for one set of parameters and then the corresponding best results and their parameters can also be mentioned. Comparable instances should be used in all comparisons, for example, different distance truncation rules often yield different optima.

Other limitations include limited coding skills and familiarity with the software, limited computing power of the device used for the study, and a limited time frame for the investigation to be conducted in. Finally, the biggest limitation is the use of the set functions in AIMMS, which require relatively high computation times. Although other methods in AIMMS are available such as the CPLEX SOLVER, those methods are aimed at advanced AIMMS users, whereas set functions are more appropriate for beginners. As a result, computation times were not factored in when making comparisons with the best known solutions and the number of iterations were limited to 100 (which is equivalent to approximately ten minutes of computing time). This figure is significantly lower than the amount of iterations used to produce the best quality solutions.

1.5 LAYOUT

A literature review is covered in Chapter 2, followed by the formal problem description and methodology used for the investigation in Chapter 3 and Chapter 4 respectively. Results and

discussion are presented in Chapter 5, followed by concluding chapters, including recommendations for future study in Chapter 7.

2 LITERATURE REVIEW

The following chapter includes a detailed description of a common class of combinatorial optimisation problems known as the VRP as well as the solution approaches that emerged as a result of studying these problems.

The first subsection covers the formal definitions of the VRP and its variants. This is followed by a subsection outlining how various solution approaches were developed. Given the extensive amount of literature available on this topic, only a few heuristic and metaheuristic approaches are discussed. The main focus will however be on the tabu search metaheuristic. The chapter closes with an overview of several of the current state of the art tabu search implementations.

2.1 THE VRPWT AND RELATED PROBLEMS

In simple terms, the travelling salesman problem (TSP) can be described by a salesman needing to visit a given set of houses and return home, incurring the smallest costs possible. The VRP can be seen as a generalisation of the TSP as multiple 'salesman', or rather vehicles, are considered. Finally, the VRPTW is simply one variant of the VRP with the added complexity of only visiting customers during allowable service times. Formal definitions of the TSP, the TSP with time windows (TSPTW), the VRP and the VRPTW are discussed in the following sections.

2.1.1 The travelling salesman problem

An earlier study states that the TSP is one of the most well-known combinatorial optimisation problems (Laporte 2010). Based on the literature, the concept involves creating the shortest set of routes, or rather the shortest tour, for a salesman to travel to a given set of scattered locations and return home. Each location must be visited exactly once. Such a tour is commonly referred to as 'Hamiltonian'.

The combination of a simple problem statement and the complexities involved in solving the problem is a possible explanation for the popularity of the TSP. The TSP is also highly applicable in real-life situations, in fact, it is a subclass of a larger set of combinatorial problems known as the VRP (which will be discussed at a later stage). The main application that is of interest for this paper is that of distribution management and scheduling problems. The study of TSP has resulted in the development of optimisation techniques commonly used in operations research, including both exact and heuristic approaches. For the sake of completeness, a formal definition of the TSP is provided.

The TSP can be defined on a graph $G = (V, A)$ where $V = \{v_0, v_1, \dots, v_n\}$ is a set of vertices that represents locations to be visited and $A = \{(v_i, v_j) : v_i, v_j \in V, i \neq j\}$ represents a set of arcs, which is effectively the journey between the given locations. The exception is v_0 which represents a depot from which the tour should start and end. With each arc there is an

associated cost (c_{ij}). If this cost is the same for travelling from location i to j and travelling from location j to i, then the problem is said to be symmetric. Otherwise the TSP is said to be asymmetric. For the symmetric TSP the edge set $E = \{(v_i, v_j): v_i, v_j \in V, i < j\}$ can be used instead of the arc set and the problem is then defined on the undirected graph $G = (V, E)$.

Finally, the triangle inequality is satisfied when $c_{ij} \leq c_{ik} + c_{kj}$ for every i, j and k. This is usually the case for planar problems, that is, all locations are on the same level. The cost c_{ij} is then simply the 'Euclidean distance' between locations i and j denoted by $\sqrt{(X_i - X_j)^2 + (Y_i - Y_j)^2}$.

2.1.2 The travelling salesman problem with time windows

The TSPTW is an extension of the TSP and can formally be defined in a similar fashion as the original TSP with the added notation of $[a_i, b_i]$ which represents a time window in which vertex v_i must be visited (Gendreau et al. 1998). In attempt to simplify matters it is assumed that the service time at vertex v_i is included in the cost c_{ij}. The idea behind the time windows is that a location must be visited no later than the upper bound of the window b_i. Early arrival before a_i is however permissible, the salesman would simply have to wait and depending on the variant of the problem applied, may incur a waiting cost.

There are two possible objectives when solving the TSPTW. The aim could either be to minimise the total cost of the entire tour or to minimise the arrival time when the salesman returns to the depot. Often the vertex set will include an additional vertex v_{n+1} which denotes the return to the depot and v_0 is used to denote the departure from the depot.

The TSP and TSPTW are subclasses of a larger problem class known as the VRP which will be discussed in the following section.

2.1.3 The vehicle routing problem

The VRP is central in distribution management. Based on the research of several authors (Cordeau et al. 2002; Laporte & Cordeau 2002; Potvin et al. 1996; Braysy & Gendreau 2005; Desrochers 1992) a formal explanation of the classical VRP is provided in this section. It should be noted that there are several variants of the VRP, but describing the classical VRP is an appropriate introduction.

The problem can be illustrated on a graph $G = (V, A)$ where $V = \{v_0, v_1, \ldots, v_n\}$ is a set of vertices and $A = \{(v_i, v_j): v_i, v_j \in V, i \neq j\}$ is a set of arcs. Note that this is rather similar to the definition of the TSP as the vertices represent locations and the arcs represent the journeys between these locations. Specifically, the vertex v_0 represents a depot and the remaining vertices represent customers to be serviced, requiring either a delivery or pickup of goods. With each arc there are corresponding costs (c_{ij}) and travelling times (t_{ij}). These costs and travelling times between the vertices can be represented in matrix notation. Typically, the matrices are symmetrical, meaning that the cost and travel time from v_i to v_j is the same as that from v_j to v_i. The symmetric nature of these values allows the VRP to be defined on an undirected graph $G = (V, E)$ where $E = \{(v_i, v_j): v_i, v_j \in V, i < j\}$ is an edge set. As with the arcs, with each customer there is a corresponding non-negative demand (q_i) and a service time (t_i). At the depot is based a fleet of *m* homogenous vehicles each with a

capacity *Q*. The number of vehicles can be fixed or it can be treated as a decision variable, which may have an upper bound which cannot be exceeded.

The VRP is concerned with designing a set of routes with minimal costs for deliveries or pickups such that the number of routes do not exceed *m*. The following conditions must hold:

(i) all routes start and end at the depot,
(ii) each customer is visited exactly once by one vehicle,
(iii) the total demand of the customers for each route does not exceed the vehicle's capacity *Q*,
(iv) a specified time limit *D* exists, such that the total duration of each route, including travelling and service times, does not exceed *D*, and finally,
(v) the total routing cost in minimized (Cordeau et al. 2002).

There are two major constraints to the classical VRP, namely the vehicles' capacity *Q* and the allowable route duration time *D*. Essentially the whole process involves assigning a set of customers to each vehicle.

The objective of the VRP is usually twofold where one concern is to minimize the number of vehicles (*m*), which translates to minimising the number of routes, and the other objective is concerned with minimising total costs involved. A solution with *m* routes always trumps a solution with *m*+1 routes regardless of total route time. Thus, minimising the number of routes is the primary objective, and then, for a given number of routes, a secondary objective would involve minimising costs involved. These objectives should be met without exceeding vehicle capacities or route length constraints. Note that the terms route length and route duration will be used interchangeably.

2.1.4 The vehicle routing problem with time windows

Several variants of the VRP exists where complexities are added to the classical VRP. Using a fleet of different vehicles (as opposed to identical ones), allowing deliveries and pickups in one route, prohibiting certain vehicles from visiting certain customers, allowing multiple visits to customers, using multiple depots, or splitting deliveries between vehicles are only some of the extensions of the VRP (Cordeau et al. 2002). The variant of main concern for this paper is where a time window $[a_i, b_i]$ is imposed on the visit of each customer. The time window conveys the earliest and latest times that a customer is available to be serviced, this is called the VRPTW.

Thus, given a central depot and identical vehicles, routes starting and ending at the depot are designed such that customers, who each have their own demands, are serviced at least once at the least possible cost, all the while satisfying vehicle capacity, route duration and time window constraints. One variant of the VRPTW is when hard windows are used (Desrochers 1992; Potvin et al. 1996). For the hard window case, vehicles cannot arrive late, that is, after the upper bound of the time windows, but if vehicles arrive too early, they must wait until the allowable service time. This waiting time is included in the total route length calculation.

For the sake of simplicity travel time, travel distance, and travel costs are all used interchangeably. Also, as stated previously, the number of routes, *m,* can either be a fixed

number or be allowed to vary provided that it does not exceed some given upper bound (Gendreau et al. 2010). Customer demands and time windows are given parameters, whereas the starting times of services are decision variables, that is, determined in the solution.

2.2 SOLUTION APPROACHES FOR THE VRP AND RELATED PROBLEMS

Over the last few decades, attempts to solve the VRP and other difficult combinatorial problems have resulted in the development of multiple optimization approaches, particularly metaheuristics, that are used in operations research regularly today (Gendreau & Potvin 2005). Since the introduction of the VRP, an array of solution methodologies evolved including both exact and approximate approaches.

According to the literature, exact algorithms can only optimally solve a small number of instances and no exact algorithm has yet to be discovered that can successfully and consistently provide good quality solutions for problems involving more than 50 customers (Laporte & Cordeau 2002; Cordeau et al. 2002). Due to the complexity and real-life applicability of the VRP, and its variants such as VRPTW, techniques which could produce high quality solutions in limited time were necessary, making heuristic approaches significant (Braysy & Gendreau 2005). Even though heuristic methods compromised on solution quality and did not always produce the best possible results, fairly good quality solutions could be produced relatively quickly (Cordeau et al. 2002; Laporte et al. 2000). Heuristics were found to offer a decent trade-off between solution quality and the time needed to produce them.

It would be appropriate at this stage to outline how exactly a heuristic functions. A popular heuristic is a local search procedure. Based on a previous study (Braysy & Gendreau 2005), the process can be described as follows. An initial feasible solution for a problem is generated, then, using a move-generation mechanism, this solution is repeatedly improved by exploring neighbouring ones. Move generation methods construct neighbouring solutions by changing single or multiple attributes of a given solution. An example of an attribute is the arc connecting a pair of customers. Once a neighbouring solution is identified, it is compared with the current solution, if it improves it, the neighbouring solution replaces the current solution and the search continues. Replacing a current solution with a better solution depends on some acceptance criterion. There are two acceptance strategies, first-accept selects the first solution in the neighbourhood that satisfies the acceptance criterion, and best-accept investigates all neighbouring solutions which satisfy the acceptance criterion and selects the best one (Braysy & Gendreau 2005). A stopping mechanism is also employed to establish when the search ends. For example, when no more improvements can be found, that is, the current solution is better than any of the solutions in its neighbourhood, the procedure stops and accepts the current solution as the optimal one.

Heuristics are however limited in practical applications and display some shortcomings. They conduct a relatively limited exploration of the solution space and the optimal solution found may simply be a local optimum that could be far from the global optimum (Laporte et al. 2000; Braysy & Gendreau 2005; Cordeau et al. 2002). Thus the main drawback of heuristic approaches is becoming trapped in local optima. One possible explanation could be due to the fact that only solutions that improve on the current solution are accepted during the

search process, meaning some paths leading to the global optimum may be overlooked. Hence, heuristic approaches are sometimes referred to as classical heuristics as they do not allow deteriorating solutions to be considered whereas more modern approaches such as metaheuristics do. Furthermore, classical heuristic approaches, such as the local search above, are heavily dependent on the initial solution and the neighbourhood generation mechanism (Braysy & Gendreau 2005). Thus metaheuristics were developed as a way to overcome these faults.

Metaheuristics perform a more thorough exploration of the solution space, conducting deeper searches in more promising areas of the solution space (Laporte et al. 2000; Cordeau et al. 2002). This is achieved through the fact that metaheuristics deviate from classical heuristics by allowing the consideration of inferior and even infeasible solutions throughout the search as well as allowing the recombination of previous solutions to make new ones (Braysy & Gendreau 2005; Cordeau et al. 2002; Laporte & Cordeau 2002). The quality of solutions produced by metaheuristics are relatively much higher than those produced by classical heuristics, but this is compensated with longer computational times. Metaheuristics are also more complex than classical methods given the use of several parameters and applications to specific variants of the VRP. Thus, improvements achieved through the use of metaheuristics were at the expense of speed and simplicity (Cordeau et al. 2002). In summary, metaheuristics can be seen as natural extensions of classical heuristics (Laporte et al. 2000).

The following section provides an overview of the development of approaches to solve the VRP and its variants, however the main focus will be on the VRPTW. First, an explanation of how heuristic approaches are evaluated is provided followed by a brief overview of how classical heuristic approaches are conducted. A short description of the most popular classical heuristics will then be covered. Before introducing some successful and popular metaheuristics, such as simulated annealing and genetic algorithms to name a few, a detailed description of tabu search is provided as it was singled out as the most successful metaheuristic approach for the VRP in numerous studies (Laporte et al. 2000; Cordeau et al. 2002; Laporte & Cordeau 2002). Finally, an overview of some common features present in the most effective approaches is given and comparisons are drawn with tabu search, highlighting it as the best metaheuristic in terms of displaying a majority of the best metaheuristic features.

2.2.1 Evaluation of heuristic approaches

In a previous survey, four main attributes of what constitutes as a good heuristic are identified (Cordeau et al. 2002). The following is a summary based on the findings of this study, identifying accuracy, speed, simplicity and flexibility as the main assessment tools for heuristics. The relevance of each of the criteria as well as the issues that arise when assessing them are discussed.

Accuracy refers to the level of deviation of a solution value to the optimal solution that exists, often measured as a percentage. This measurement is important as it distinguishes the best solution approaches from mediocre ones. Problems arise when analysing VRP heuristic results because only the best several runs or parameter results are given. Different rounding and truncating methods used by various implementation also lead to inconsistent results being

recorded as a bias is introduced by rounding. Another issue is consistency as the most preferable methods perform well on every instance or run as opposed to producing excellent results for some runs and really poor results for other. Finally, it is also preferable if good solutions are produced at an early stage in the search process and improved as the search continues instead of having a final answer after a long computation time without seeing the progress of the procedure.

The speed of an approach refers to its computation time. Faster algorithms are preferred to slower ones, however, what constitutes an acceptable speed depends on the accuracy required and planning level of the problem. For example for problems where solutions are required almost immediately such as ambulance deployment or daily problems for a business, fast algorithms are needed with computation times ranging between minutes and seconds. For long term big investments on the other hand, such as procurement of a fleet of vehicles, a few hours may be seen as an acceptable computation speed. Accurate reporting of computing time is an issue, especially if multiple runs were conducted. Similar to accuracy measurement results, no strict standards for measurement are consistently enforced.

Simplicity refers to how easily a method can be replicated. For example, simple heuristics are not too complicated to understand and code. Short and self-contained codes are preferred, however a minimum level of complexity is needed for good results. That being said, algorithms should not be over-engineered. For example, if parameters are used, they should be limited in number and be easily understood by the user. Thus parameters should either be fixed to some value or be updated throughout the search process in a way that makes sense to the user.

Flexibility refers to how adaptable a heuristic is. To score well on this attribute a heuristic should accommodate various side constraints that arise from real-life applications. That is, they must be robust and their rules should be broad enough to be applied to numerous practical applications of the problem. One way to improve the flexibility of a heuristic is to allow the consideration of intermediate infeasible solutions.

Traditionally, only speed and accuracy were considered when evaluating heuristics. The introduction of assessing simplicity and flexibility are relatively new concepts. However, it can be seen that the four attributes provide a comprehensive overview of the quality of a heuristic. Flexibility goes hand in hand with simplicity of design, thus both must be considered when implementing a solution approach. Also, there seems to be a trade-off between speed and accuracy, as good quality solutions generally tend to require longer computational times. Finally, simplicity and speed are also linked in that the time required by a heuristic depends on the time needed to generate each solution, determine its feasibility and evaluate the costs involved. Thus the speed of a heuristic depends on its cardinality, that is, the number of moves that neighbour a generic solution (Toth & Vigo 2003). Furthermore, cardinality depends on the neighbourhood structure used, which ties in with the simplicity level of an approach.

2.2.2 Classical heuristics

Despite the possibility of becoming trapped in local optima that results from the use of classical heuristics, given their ability to be applied to real-life problems, they are still

commonly used today (Laporte et al. 2000). Thus heuristics still prove to be flexible and efficient with regard to solving the VRP, especially when used in conjunction with other approaches (Cordeau et al. 2002).

Classical heuristics were developed mostly between 1960 and 1990 (Laporte et al. 2000). Early classical heuristics involved quickly getting feasible solutions and were typically used as post-optimisation procedures (Cordeau et al. 2002). According to the literature, there are a number of techniques available when using classical heuristics to solve the VRP and its variants (Laporte et al. 2000; Laporte & Cordeau 2002; Braysy & Gendreau 2005; Potvin et al. 1996).

Constructive heuristics involve gradually assigning vertices to various routes. The idea is to build a feasible solution throughout the process using an insertion heuristic, simultaneously trying to keep costs of the solution to a minimum. Route construction heuristics can be split into two types, specifically sequential methods and parallel methods. The former involves construction of one route at a time and the latter constructs multiple routes simultaneously.

Improvement methods act on single routes or several routes through customer reallocations or exchanges. These heuristics are concerned with merging or making alterations to existing routes using a saving criterion. The local search heuristic described earlier is an example of an improving method. Starting with an initial solution, the heuristic modifies this solution to obtain a new and improved one and the process is repeated until no more improvements can be made.

Finally, two-phase heuristics first group customers into feasible routes and thereafter actual routes are constructed, examples include the sweep algorithm and the cluster-first, route-second algorithm. Post-optimisation procedures can be applied to further improve routes and reduce costs once constructed.

In summary, classical heuristics are mainly concerned with construction or improvement procedures. Common classical heuristics include the savings algorithm, the sweep algorithm and the cluster-first, route-second algorithm (Cordeau et al. 2002). In addition to these heuristics, edge-exchange improvement heuristics as well as construction heuristics such as the nearest neighbour and the so called Solomon insertion heuristic will be outlined below. As mentioned in Chapter 1, some of the approaches mentioned below were originally designed for the VRP and therefore may not explicitly consider time window constraints. An earlier study may be consulted for a more detailed description of how the original algorithms can be modified to be applied to the VRPTW (Solomon 1987) although some of the modifications based on this study are also mentioned in the following discussion.

Savings heuristic

The most widely known heuristic for the VRP was created by Clarke and Wright called the savings algorithm (Laporte et al. 2000). The popularity of this approach stems from how easy the basic principle is to understand and code. The savings heuristic is applied to variants of the VRP where the number of vehicles is a decision variable. As far as symmetry is concerned,

the heuristic performs equally well for directed or undirected problems. Both sequential and parallel versions of the algorithm exist.

The following is a description of the savings heuristic based on the work of several authors (Laporte et al. 2000; Cordeau et al. 2002). The procedure starts with an initial feasible solution consisting of a number of routes that start and end at the depot and visit customers in between. For example, if there are n customers, then the initial solution will consist of n routes. For $i = 1, \dots, n$, the n initial vehicle routes are created of the form $(0, i, 0)$. That is, the route starts at the depot, visits customer i and then returns to the depot.

At any point in the process, two routes can be merged into one. For example, a route whose last visited vertex before the depot is i, denoted by $(0, \dots, i, 0)$ and a route whose first vertex visited after the depot is j, denoted by $(0, j, \dots, 0)$, can be merged to form the single route $(0, \dots, i, j, \dots, 0)$. The condition is that the merge must be feasible, that is, it must not violate any of the constraints. Once a merge is implemented, the saving that result from the merge between customer i and j can be computed as $s_{ij} = c_{i0} + c_{0j} - c_{ij}$ for $i, j = 1, \dots, n$ and $i \neq j$.

For parallel versions, best-accept methods are employed. This means, given a saving, s_{ij}, the solution space is searched for two routes that can be feasibly merged such that one route starts with $(0, j)$ and the other ends with $(i, 0)$. This merge would imply deleting edges $(0, j)$ and $(i, 0)$ and introducing the edge (i, j). The merge resulting in the largest saving is implemented.

For the sequential version, one route is expanded at a time until addition of more customers is no longer feasible. A first-accept route extension method is employed. What this means is that the first saving that can feasibly be used to merge two routes is sought. Let the current route be $(0, i, \dots, j, 0)$. Another route ending with $(k, 0)$ or starting with $(0, l)$ is searched for to obtain a saving of s_{ki} or s_{jl} which can be achieved by merging the routes at customers k and i or j and l respectively. If such a feasible merge exists, it is implemented and the process is repeated with the new current route. If no feasible merge is found, the next route is considered and the same procedure is applied. The whole process is repeated until no more feasible routes remain. Also, it is common to apply a 3-opt post-optimisation procedure to the final solution (details of which will be described at a later stage).

To apply this heuristic to the VRPTW, the literature states that the route orientation must be considered to account for the time window constraints (Solomon 1987). Orientation can be maintained if the end customers of two partial routes to be merged is such that the last customer of the first route is merged with the first customer or the second route. At this point it becomes necessary to check both the vehicle capacity constraints as well as the time window constraints at every iteration. One drawback of this alteration is that it results in longer periods of waiting time which have high opportunity costs as other customers could have been serviced. To remedy this the author suggests limiting the waiting time that results from implementing a merge.

The main benefits of the savings algorithm is that it is speedy and simple with no parameters, making it relatively easy to code. The main disadvantage is that the heuristic lacks flexibility

and solution quality can be poor when additional side constraints have to be considered, hence the need for the above adaptation to account for time window constraints. This lack in flexibility may be a result of the fact that the algorithm is based on the 'greedy principle', that is, it searches for the biggest saving possible. Also, once a merge has been implemented it cannot be undone, hence unsatisfactory route merges remain in the solution. Attempts made to improve this approach such as the matching algorithm (details of which will not be discussed in this paper) are irrelevant given that they do not solve the issues regarding flexibility and simultaneously reduce the heuristic's score with regard to speed and simplicity.

Sweep heuristic

The following is a description of the sweep heuristic based on the work of several authors (Laporte et al. 2000; Cordeau et al. 2002). This approach was popularized by Gillet and Miller and is most commonly used to solve planar instances of the VRP, that is, when all vertices are on the same level. A ray centred at the depot is rotated in a circle to form initial feasible clusters. This is achieved by selecting an unrouted vehicle. The vertex having the smallest angle with the depot is the starting point for the route. Gradually, vertices are assigned to the initial route, increasing the number of customers until the upper limit of one of the constraints are reached. If unrouted vertices remain, a new route is started with the next unused vehicle and the process is repeated until the whole plane has been swept. At this point all the customers would be grouped into clusters, then for each cluster, a route is created by solving a TSP. Hence the approach can be viewed as a two-phase heuristic. Sometimes post-optimisation techniques are included, where customers are exchanged between clusters that are next to each other and then routes are re-optimised.

Although this heuristic is simple, like the savings algorithm, it lacks in flexibility. Once again the greedy nature of the search results in poor solutions when additional side constraints are to be considered.

Attempts to improve this technique include the petal algorithm, which can be seen as a natural extension of the sweep algorithm. Several routes referred to as petals are generated and a set partitioning problem is then solved. Further details of this approach will not be discussed in this paper, as gains in accuracy and accommodation of a wider variety of constraints are at the expense of simplicity.

A more successful approach known as the time-oriented sweep heuristic (Solomon 1987), was developed to remedy the lack of flexibility of the heuristic and introduces adaptations for problems involving time windows. Very similar to the original heuristic, the sweep is performed to group customers together and then a construction heuristic is used to build tours. The type of construction heuristic used is what differentiates the time-orientated heuristic from the original. One strongly recommended heuristic is the insertion heuristic of type I1 which will be introduced at a later stage.

Cluster-first, route-second algorithm

The following is a description of the two-phase heuristic based on the work of several authors (Laporte et al. 2000; Cordeau et al. 2002). The cluster-first, route-second algorithms was

introduced by Fisher & Jaikumar. The algorithm was originally designed to solve another combinatorial optimisation problem known as a generalised assignment problem and was later applied to the VRP.

The number of vehicle routes is fixed beforehand. The process then starts by dividing the customers into feasible groups by splitting the plane into cones, where the number of cones correspond to the number of vehicle routes. This is done by selecting dummy customers, one for each route, referred to as 'seed points'. Customers are then allocated to these dummy points so as to minimise the distances between customers and this seed point. Thus, once clusters have been formed, a TSP is solved using a constraint relaxation approach to construct a route.

This process in not simple to code and past results are unreliable as time depends on seed selection and the constraint relaxation techniques used. The method also scores low in terms of accuracy and flexibility.

Improvement heuristics

The following is an explanation of improvement heuristics based on the work done by several authors (Laporte et al. 2000; Cordeau et al. 2002; Potvin et al. 1996).

Improvement heuristics are mainly used as post-optimisation techniques, the two main types include intra-route methods and inter-route methods. The first refers to a sequential version where each route is considered separately and improved using a TSP algorithm. The second refers to a parallel version where several routes are considered simultaneously. Often, both versions are used within the same improvement heuristic.

The heuristic involves removing a number of edges from a route and reconnecting the remaining segments in all possible ways in attempt to find profitable ones. The procedure stops at a local minimum when no more improvements can be made. Hence the technique is also referred to as an edge-exchange procedure and is also known as *k*-opt, where *k* refers to the number of edges that will be removed. Because the size of *k* determines the cardinality and hence the computing times of an approach, often smaller values such as 2 or 3 are preferred for *k*.

This process is best illustrated with an example. Consider the case where *k* is set to 3, that is, the 3-opt approach. The process repeatedly removes three edges and reconnects the three resulting chains in all possible ways and stops when no further improvement is possible. There are several variants of 3-opt. One example is a first-improving method when the first improvement is accepted. Another example is a best-improvement method, where the whole neighbourhood is explored and the best improvement is selected.

These *k*-opt exchanges can be seen as local search heuristics, that is, starting with an initial solution, modifications are repeatedly made to improve the solution until no more improvements can be found.

The biggest problem with this approach is, because it does not preserve the orientation of the routes, the method is not appropriate for problems involving time windows as route reversals can results in infeasible solutions. However, improvements to this heuristic have been made in attempt to preserve route orientation. For example 2-opt* involves removing the first customer from the first route and linking it to the last customer on the second route and vice versa.

Or-opt is another improvement method where strings of three, two or one consecutive customers are considered for removal and reinsertion elsewhere, either in the same route or another route. Or-opt exchanges are actually a subset of 3-opt exchanges and perform well for problems involving time windows.

Construction heuristics

Two main construction heuristics, sometimes also referred to as tour-building heuristics, will now be discussed. Namely they are nearest neighbour heuristics and insertion heuristics. The following outlines the basic ideas of these sequential construction heuristics based on the findings in an earlier study (Solomon 1987).

The concept is relatively straightforward with regard to the nearest neighbour heuristic. At each iteration the algorithms searches for the 'closest' unrouted customer to the last one. The first customer inserted to the route is the one 'closest' to the depot. If any of the constraints are violated a new route is started and the process is repeated until no more unrouted customers remain.

The term 'closest' is dependent on the variant of the problem being solved. For the VRPTW, both spatial and temporal closeness of customers must be investigated. That is, the direct distance between customers, the time difference between completion of service at the previous customer and start of service at the next, and the remaining time for the last allowable service start at the next customer (termed 'urgency') must all be considered. The original paper may be consulted for a more detailed description of the exact algorithm.

For the more popular insertion heuristic, an initial solution is needed, the heuristic then inserts a new customer between two adjacent customers in the current partial route at each iteration. This is done by determining the best feasible insertion place by evaluating the cost of the partial route at each possible point of insertion. When no more feasible insertions can be made, a new route is started and the process is repeated until all customers are assigned to a route.

Evaluation of insertions depend of the type of insertion heuristic used. Specifically the first type, I1, focuses on minimising a weighted combination of the extra distance and extra time that results from inserting a customer. The final evaluation is therefore highly dependent on how the extra time and distance are weighted. The second type of insertion cost minimises the total route distance and time. Finally, in the third type, the time consideration also factors in the urgency of a customer (as outlined in the description of the nearest neighbour heuristic).

In fact, the insertion heuristic can be viewed as a generalisation of the time-orientated nearest neighbour heuristic as unrouted customers are allowed to be inserted in any feasible position as opposed to simply adding unrouted customers to the end of a route.

It is expected that these heuristics perform well, producing good quality solutions with significantly lower waiting times than solutions produced by distance-driven criteria.

2.2.3 Tabu Search

Based on the work of several authors (Laporte & Cordeau 2002; Glover 1993; Bräysy & Gendreau 2005; Potvin et al. 1996; Glover 1990; Laporte et al. 2000) the following section describes the basic elements of tabu search. Tabu search is one of the best local search methods for combinatorial optimization. This established optimization approach was introduced by Fred Glover and, like other metaheuristics, remedies the shortcoming experienced when using classical heuristics of being trapped at local optima, allowing a greater portion of the solution space to be explored. Tabu search is an iterative search procedure which guides a search to obtain consistently good quality solutions in complex solution spaces. Given that its rules are broad and that the approach is flexible, tabu search can be used in conjunction with numerous other heuristic procedures, specifically, processes that involve transforming solutions so that moves can be evaluated in terms of attractiveness. What constitutes as a move can vary; for example, the value of some variable can be changed, or elements can be added or deleted from a set. Thus, tabu search can be applied to various problems and is considered one of the top performing approaches to solving the VRP.

Tabu search can be seen as an extension of a classical local search heuristic or descent method with regard to minimisation problems. A solution space is explored by repeatedly making moves from one solution to another solution located in the neighbourhood of the previous solution. Specifically, the move is made to the best solution in the subset of the neighbourhood of the previous solution. The current solution is continuously updated with each iteration. In general terms let X be the entire solution space with s as one of the possible solutions. Let $N(s)$ be the neighbourhood of s, which is effectively a subset of X. Since it might be cumbersome to explore the entire neighbourhood, a subset of $N(s)$ is explored for its best possible solution which is denoted by s'. This process is repeated at each iteration until some stopping criterion is satisfied. Some possible stopping criteria include finding the next neighbourhood to be empty, reaching a pre-set maximum number of iterations, finding no improvements for a pre-set maximum number of iterations, computational time expended or finding the optimal solution.

The quality of the solutions are based on the evaluation of some objective function $f(s)$ which is to be minimized. Thus the selected solution s' in the neighbourhood $N(s)$, satisfies the condition $f(s') \leq f(\bar{s})$, where $\bar{s}$ is any element in $N(s)$. In traditional descent methods, if $f(s') \geq f(s)$, that is, the current solution s is lower than the best solution in $N(s)$, then the procedure stops and s is selected as the best solution. Otherwise s' is set as the new current solution and the process continues until some stopping criterion, for example, no better solutions in the neighbourhood of the current solution exists. Tabu search differs to classical local search heuristics in that s' can be selected from $N(s)$ regardless if it is worse (higher)

than the previous solution s. This gives the search an opportunity to explore other solution paths that would otherwise have been missed if only non-deteriorating solutions were considered at each iteration.

Since deteriorating solutions are being considered, a new weakness arises as the process is now prone to repeatedly cycle over a sequence of solutions. The search can once again be trapped in a certain area of the solution space. This is remedied by a tabu mechanism which can forbid the selection of previously visited solutions. However, recording every single solution already considered can amount to excessive bookkeeping, which leads to the introduction of one of the main components of tabu search, its adaptive memory. Instead of the actual recent solutions, attributes of these solutions are registered and these attributes, or rather the reversal of these attributes, are forbidden for the next θ iterations. Not only does this alleviate memory requirements, time is also saved. In other words, attributes of recently visited solutions are declared tabu for a duration of θ iterations, also known as the tabu tenure, whose value can be fixed or varied throughout the search.

Popular tabu search approaches are those that are easy to implement, easy to code and capable of achieving efficient outcomes while requiring minimal computational times. The aim is to get good solutions fast. The quality of a tabu search method therefore depends on its specific features which are worth discussing in some detail. These features include neighbourhood structures, control mechanisms, short term and long term memory features, and intensification and diversification procedures.

At this stage, tabu search can be understood to involve strategically constructing non-empty neighbourhoods of a solution at each iteration and then searching the neighbourhood for its best element, specifically avoiding moves declared as tabu. At each iteration, the neighbourhood is redefined before it is searched (for the next solution to be selected from it). How the neighbourhood is defined is therefore important as it significantly impacts the solution obtained. An efficient tabu search needs a good balance between neighbourhood quality and computational efficiency. More complex neighbourhoods lead to better searches, since more solutions are considered at each iteration, but at the same time, more computational effort is required.

In the VRP context, there are two main ways to define tabu neighbourhood structures. The first is referred to as λ-interchanges which entails exchanging up to λ customers between two routes. The second is referred to as ejection chains, which performs exchanges between more than two routes simultaneously. For the λ-interchanges, λ is often limited to 1 or 2 so that the possible candidate moves considered are restricted. The λ-interchanges include simple swaps between two routes or simple moves taking one customer from the first route and inserting it into another. Variants exist where the routes or vertices considered for exchanges are limited. Sometimes local re-optimisation of routes are carried out in which additional removals or insertions are performed. Ejection chains are slightly more complex but in essence the method is simply a generalisation of λ-interchanges, extending the procedure to more than two routes.

The tabu search short term memory feature, which is flexible and based on recent attributes, refers to the tabu mechanism. The idea is to prevent cycling by prohibiting reverse moves for a given number of iterations. The reverse moves are stored on a tabu list whose size may be fixed or allowed to vary by randomly selecting the list size from a given interval of values. The size of the tabu list is important given that a list that is too short will not remedy the threat of cycling whereas a list that is too long will add unnecessary computation time. The tabu status of a move is a function of the tabu status of its attributes. Examples of these attributes include changes in the values of variables, changes in the positions of elements, or transferring elements between sets. Finally, the tabu list is not static, but rather it is constantly updated throughout the process. Tabu restrictions can be seen as a control mechanism.

Aspiration criteria is another control mechanism, whereby a move's tabu status can be overruled if certain conditions are met, provided that cycling is not a threat. Possible aspiration criteria include tabu moves that yield better solutions than any other non-tabu move in the current neighbourhood, tabu moves resulting in better solutions than the best current solution so far, or tabu moves yielding solutions which are better than some threshold value. Also, multiple aspiration criteria may be considered instead of simply using one criterion.

Longer term memory features of the tabu search involve intensification and diversification. These two functions have memories of varying time spans and counterbalance each other as intensification reinforces a search and focuses on depth, whereas diversification aims to fully cover the entire solution space, thus focusing on widening the search.

Good solutions have common properties which are recorded. Intensification involves adjusting the criteria for evaluating moves to favour solutions possessing properties of the best known solutions or it restricts the search to moves possessing these properties in hopes of locating an even better solution. One way to implement intensification is using adaptive memory. The best solutions so far are pooled together, and solution elements from this pool are combined in attempts to create new and improved solutions. Specifically, a number of non-overlapping routes are used to create new solutions. Since the new routes are constructed from non-overlapping routes, not all the vertices are included, so what this means is that the search is now initiated from the selected routes and attempts to include the unrouted customers. If this process results in improved solutions, they are added to the pool of best solutions. Adaptive memory refers to the fact that the pool is continuously updated, where better solutions are added, and older, worse solutions are removed. Thus intensification can be seen as periodic route improvements.

Diversification on the other hand encourages the search to favour moves resulting in solutions with varying characteristics. The aim of diversification is to ensure that the search is not restricted to a limited portion of the solution space. It can be implemented by keeping a record of past solutions and penalising frequently performed moves. One way to achieve this is by modifying the objective function to decrease for solutions far from previous ones and to increase for those close to the previous one. What constitutes as close or far depend on recency and frequency based memory structures. Varying the tabu list size can therefore be considered as another diversifying technique.

Thus, intensification and diversification are complementary functions playing opposite roles in the search. Intensification favours similar qualities or attributes of good solutions whereas diversification favours solutions with attributes that differ from previous ones in an attempt to broaden the search. Over the long run the search alternates between sequences of iterations that intensify and then diversify.

One final feature of tabu search is the allowance of intermediate infeasible solution. This is another mechanism used to explore solutions paths less travelled. One way of implementing this feature is through a penalised objective function $c'(x) = c(x) + \alpha Q(x) + \beta D(x)$, where $c(x)$ is the routing cost of solution x, $Q(x)$ and $D(x)$ are the deviations of the demand and route duration constraints, respectively, and α and β are self-adjusting parameters. The parameters are initially set to 1 and are periodically reduced or increased depending on if the last, say μ, solutions were feasible or infeasible, where μ is controlled by the user. Otherwise, α and β can be controlled by the user. Allowing infeasible intermediate solutions are appropriate for the simpler neighbourhood structures, and become less relevant for more sophisticated structures such as ejection chains.

A brief overview of the entire tabu search process can now be illustrated in simple terms. The process starts with some initial solution obtained by some classical heuristic. A candidate list of moves are created, whereby each candidate move generates a solution different from the current solution. These moves may result in infeasible solutions depending on the type of tabu search applied. Based on the tabu and aspiration criteria, the best admissible candidate is selected as the new current solution, where admissibility is determined based on a move's tabu status. If the new solution is an improvement of the last one it is registered as the new best solution. The process continues until some stopping criterion is satisfied. Tabu restrictions, neighbourhood of candidate moves and aspiration criteria are continuously updated throughout the process. In general, the tabu list is relatively smaller than the solution neighbourhood, making it advisable to first evaluate all moves before determining its tabu status. Thus, to select the best candidate in a neighbourhood, the first step is to evaluate each move. All the tabu moves are then measured against the aspiration criteria to check for its admissibility. Finally the best admissible move is selected and the process is repeated until some conditions for the stopping criterion are met. Additional diversification and intensification procedures can also be incorporated into the search as they are efficient in producing good results and are relatively computationally inexpensive.

2.2.4 Other Metaheuristic approaches

Metaheuristics have been developed over the last few decades (Laporte et al. 2000). Some common features include sophisticated neighbourhood search rules, memory structures and recombination of old solutions to make new ones. The main challenge is to adapt metaheuristics to particular problems or a problem class (Gendreau & Potvin 2005). All the metaheuristics discussed in this section will be in the context of how it performs as a solution approach for the VRP and its variants. Good metaheuristics are those that are robust and lead to near-optimal solutions in reasonable computational times.

Metaheuristics can be categorised into local and population searches (Cordeau et al. 2002). Local search involves intense exploration of the solution space, sequentially moving from the

current solution to a better one in its neighbourhood. Local searches can also be referred to as single solution metaheuristics (Gendreau & Potvin 2005). Some examples of local searches include simulated annealing and tabu search. Population search consists of compiling a pool of good solutions and recombining some of those solutions in attempt to get new and better solutions, thus multiple solutions are considered simultaneously. An example of this type of metaheuristic include genetic search algorithms which combine two parent solutions to produce offspring solutions. An extension of the genetic search algorithm is the adaptive memory procedure, where several parents are used to produce several offspring instead of only two.

Within each of the two categories, metaheuristics can be further decomposed into constructive and improvement metaheuristics (Gendreau & Potvin 2005). The first involves building a solution from scratch and is usually an iterative process that adds new elements at each step. For the second, a solution is modified at each iteration in attempt to lower the objective function value.

The following is a brief overview of the top metaheuristic approaches based on the findings of several authors (Cordeau et al. 2002; Gendreau & Potvin 2005; Bräysy & Gendreau 2005; Laporte 2009).

Simulated annealing

According to authors (Gendreau & Potvin 2005; Bräysy & Gendreau 2005; Laporte 2009) simulated annealing is a stochastic relaxation technique, it is an example of an improving heuristic with a single solution. The technique is inspired by the real-life process of physical annealing where a solid is heated to high temperatures and gradually cooled to allow it to crystallise in a low energy form. The goal is to find low-energy states of solids.

The solid is melted by increasing its temperature, then it is cooled using progressive temperature reduction to recover the solid state but in a lower energy form. If cooled too rapidly, irregularities occur and high energy states result. Careful cooling leads to regular structures of lower states, but it needs to be done slowly so that at each level an equilibrium is reached. Essentially, the process is less likely to get trapped in a high-energy state when the temperature is prevented from getting too far from the current level.

The analogy makes the solution to be seen as the physical state, and the solution cost can be seen as the energy, which the process aims to reduce. The method is an iterative process such that at each step the current solution modified. Let $c(x)$ be the cost associated with solution x. Given an initial solution s and a new solution s', if $c(s') - c(s)$ is less than zero, s' is accepted as the new current solution. Effectively $c(s')$ is less than $c(s)$. Thus if the new solution is an improvement, it is accepted as the new current solution. This approach allows the search to escape local optima.

If the change is greater than zero, that is, the new solution results in an increase the objective function value, then these moves may be accepted but only with some probability known as the Metropolis criterion. The probability of acceptance is related to the magnitude of cost increase and a parameter T referred to as temperature. T can vary from large values to small

values close to zero. A new solution is more likely to be accepted if temperature is high and the cost increase is low.

The temperature parameter is progressively lowered according to a predefined cooling schedule and a certain number of iterations at each temperature level. When the temperature is sufficiently low, only moves leading to improving solutions are accepted and the process stops at a local optimum.

Thus simulated annealing involves a randomised local search, where the current solution is modified, even if cost is increased. These deteriorating solutions however are only accepted with some probability based on the cost increase and a temperature parameter.

Greedy randomised adaptive search procedure

Greedy randomised adaptive search procedure (GRASP) is an example of a constructive metaheuristic with a single solution (Gendreau & Potvin 2005; Bräysy & Gendreau 2005). This technique employs a multi-start local search, which applies local search repeatedly from different starting points. A quick greedy heuristic is employed to obtain initial starting solutions. Ideally these greedy solutions should have a variety of different local optima so as to get a broad coverage of the solution space.

Each restart applies a randomised greedy construction heuristic, meaning that a solution is generated and then improved throughout the local search. This process is repeated for a given number of restarts and the best overall solution is returned at the end. During the construction part of the process, each element not yet included into the partial solution is evaluated with a greedy function and the best elements are recorded using a memory function called the restricted candidate list. An element is randomly chosen from the list and incorporated into the solution.

Randomising choice allows diversification as the best current element is not necessarily chosen. Hence GRASP differs from the classical greedy heuristics in this respect. A shortcoming is that each restart is independent of the previous one meaning previously explored solutions do not guide the search. This implies that there is no intensification aspect as the search is completely random.

Reactive GRASP has a restricted candidate list whose size adjusts depending on quality of recent solutions, similar to that of the tabu mechanism. Another variant entails recording a pool of elite solutions, where intensification and diversification can be implemented by rewarding or penalising elements often found in this pool of elite solutions.

Variable neighbourhood search

Variable neighbourhood search (VNS) is an example of an improving metaheuristic with a single solution (Gendreau & Potvin 2005; Laporte 2009). The neighbourhood structure is systematically changed within a local search, as opposed to using a single neighbourhood structure adopted. The process is initiated with a pre-selected set of structures which are often nested. A solution is then randomly generated in the first neighbourhood of the current solution. A local descent, as described in the tabu search section, is then performed. If the

local optimum is not better than best solution so far, referred to as the incumbent, the process is repeated in the next neighbourhood. Once every neighbourhood structure has been explored or a solution better than the incumbent has been found, the search restarts from the first neighbourhood.

A popular variant is where the best neighbour of the current solution is selected as opposed to a random neighbouring solution. In this case, no local descent is performed, instead the best neighbouring solution automatically becomes the new current solution provided it is an improvement on the current solution. The search is then restarted from the first neighbourhood. If the new solution is not an improvement, the next neighbourhood is considered.

The search stops when all neighbourhood structures have been considered and no improvement is possible. The current solution then becomes the local optimum for all neighbourhood structures. VNS is often combined with other metaheuristics such as tabu search and a hybrid of the two techniques are implemented.

Genetic search algorithm

The genetic search algorithm will now be described based on the work of several authors (Gendreau & Potvin 2005; Bräysy & Gendreau 2005; Laporte 2009). Genetic search algorithms are a subclass of what is known as evolutionary algorithms. The process simulates the way species evolve and adapt to the environment based on the Darwinian principle of natural selection. Hence genetic search is an adaptive heuristic search method based on population genetics. A population is evolved by creating new generations of offspring using an iterative process until some convergence criteria are met, for example, some maximum number of generations have been reached. There are four major phases in the genetic search algorithm, namely, representation, selection, recombination and a diversifying mechanism known as mutation.

Significant attributes of a member of the population are identified during the representation phase. Selection then entails randomly selecting two elements from the population to act as parents. The probability of an element being selected is dependent on its quality in terms of its objective function value, so in the VRPTW context, the lower the value the better the quality of the solution. Thus, only the best solutions are used as parents to generate offspring. Recombination involves using the most desirable genes, or rather attributes of the parents, to produce offspring to form the next generation. There is a low probability associated with mutations occurring. So before being considered for the new population, mutations randomly modifies some of these attributes so that genetic diversity in the new solutions are maintained.

Repeating the selection, recombination and mutation processes result in a new generation of solution, which is placed in a new population. Depending on the type of genetic algorithm employed, all the old elements in the population may be replaced by new ones, or sometimes a set of old elements are preserved. A good balance between genetic quality and diversity within the population must be maintained.

Customers can be clustered together and then routes are constructed within each cluster using cheapest insertion heuristics. Edge-exchange mechanisms are also used as a post-optimisation procedure to improve solution quality. Since an initial population is heuristically generated and improved until the best solution is found, genetic algorithms are improvement metaheuristics.

Local search metaheuristics can integrate aspects of genetic algorithms by using adaptive memories. Local search is applied initially and the set of best-known solutions are recorded. These best solutions are then combined to form new partial solutions and the local search is restarted from these positions.

Ant colony systems

Ant colonies systems is an example of a constructive population heuristic (Laporte 2009; Bräysy & Gendreau 2005; Gendreau & Potvin 2005). The process is based on the real-life occurrence of ants finding the shortest paths from their nest to food sources.

A chemical compound known as pheromones is their medium of communication. The hormone is laid down in various quantities on the ground. If an ant comes across a trail it will follow it with some probability and strengthen it with its own pheromone. This positive reinforcement leads to higher probability that ants will follow the path in the future, and the more ants that follow the tail, the higher this probability. Thus gradually more weight is given to edges that are common in good solutions resulting in pheromones accumulating faster on the shortest paths.

2.2.5 Common features of best metaheuristics

Based on the literature (Laporte & Cordeau 2002; Laporte 2009; Laporte et al. 2000; Gendreau & Potvin 2005) there is a call for leaner implementations of solution approaches to the VRP instead of over-engineered ones. Particularly, approaches that do not have too many devices or user controlled parameters are preferred. There is a general trend in most of the literature for faster and more robust algorithms with a simpler design, even if it is at the expense of solution quality and accuracy. Furthermore, due to the VRP's occurrence in real-life situations, algorithms must be easily adaptable to handle side constraints of these practical applications. There is also a general trend of using hybrids, that is, combinations of solution approaches, as opposed to single implementation strategies.

In an earlier study, (Gendreau & Potvin 2005) metaheuristics were noticed to converge to a unified framework. A summary of the common features in the development of metaheuristics and their adaptation to the VRP highlighted in this study is provided below.

Firstly there appears to be a search for a good balance between diversification and intensification procedures. That is, a trade-off is sought between algorithms that cover a wide range of the solution space and those that carry out an intense exploration of certain areas.

Randomisation as a diversification mechanism has also become more common. In addition to this, using selective features of old solutions and recombining them to make new ones is also a popular practice.

The reduction of user-controlled parameters and increased employment of self-adjusting parameters has led to the development of more robust algorithms that can accommodate a wider range of constraints.

Learning mechanisms such as memories are also becoming more popular in metaheuristics. In particular, adaptive memories that make use of populations of solutions are occurring more frequently in metaheuristic approaches.

One final interesting observation is that several metaheuristics are inspired by real-life occurrence, for example simulated annealing, genetic algorithms and ant colony systems.

Thus overall, majority of metaheuristics used today possess at least one of the following features. There is usually a construction of an initial solution. This is followed by generating new solutions from current ones through a recombination procedure. Random modification as a diversification technique is often used. In addition to this, an improvement process which entails selecting the best solution in a given neighbourhood is also employed. Finally, most metaheuristics contain memories, parameters and neighbourhoods that are frequently updated throughout the process.

As illustrated in the following section, where majority of metaheuristics display at least one of the above features, tabu search approaches appear to incorporate almost all of the most common features of the best metaheuristic approaches.

2.3 CURRENT STATE OF THE ART TABU SEARCH APPROACHES FOR THE VRPTW

It should be noted that not all tabu search algorithms were a success especially if the neighbourhood structure is not powerful enough to identify high quality solutions and if there are too many devices and user-controlled parameters (Cordeau et al. 2002). This section will therefore cover some of the best tabu search approaches to date based on an earlier survey (Laporte & Cordeau 2002). As stated in the introduction, due to the scope and limitations of this study, parallel implementations will be excluded from the discussion, although the original survey can be consulted for details of these methods.

The main methods that will be covered are taburoute (Gendreau et al. 2010), unified tabu search (Cordeau et al. 2001), granular tabu search (Toth & Vigo 2003), and the adaptive memory procedure (Taillard et al. 2001).

Each explanation will attempt to follow a similar structure. The initial construction phase will be explained followed by the neighbourhood structure, tabu mechanism and aspiration criteria. This is the short term memory features. The long term memory features such as intensification and diversification and post-optimisation improvement procedures will then be outlined. These include the discussion of penalty terms, parameters and the allowance of infeasible solutions. Finally an overall concluding remark will be provided.

2.3.1 Taburoute

The following is an overview of the taburoute process based on the findings of several authors (Gendreau et al. 2010; Cordeau et al. 2002; Laporte et al. 2000; Laporte & Cordeau 2002) who

state the approach to be 'rather involved'. For a more thorough explanation, the original paper can be consulted (Gendreau et al. 2010).

False starts are used to initiate the process. That is, several solutions are generated at the start of the process and a limited search is carried out for each of them. The best solution is then used as the starting point for the main search. A solution is a set S of m routes $R_1, R_2, \dots, R_m$ where $m \in \{1,2, \dots, \overline{m}\}$ and $\overline{m}$ is the maximum number of routes allowed. Each route $R_r = \{v_0, v_{r_1}, v_{r_2}, \dots, v_0\}$ and each vertex v_i, where $i \geq 1$, belongs to only one route.

The neighbourhood structure used consists of all the solutions that can be reached from the current one by removing a vertex from the current route, say s, and inserting it into another route, say r, where the new route contains one of the vertex's p closest neighbours. A generalised insertion (GENI) procedure is used (Gendreau et al. 1998; Gendreau et al. 1992), for a more detailed explanation of the method the original literature can be consulted. The process starts with a route, or tour, consisting of three arbitrary vertices. At any iteration an unrouted customer is inserted into the tour and a local re-optimisation is performed. The neighbourhood of any vertex consists of the p closest neighbours as previously stated, where p is an input parameter. If the vertex has less than p neighbours then all of them belong to the neighbourhood. There are two types of GENI insertion procedures, both are attempted and the best insertion is implemented. The process results in the elimination of existing routes or the creation of new routes and is continued until all the vertices are a part of the tour.

The tabu mechanism used involves forbidding reverse moves. This means that if a vertex is removed from route s to r at iteration t, reinsertion into r is forbidden for $t + \theta$ iterations, where θ is randomly drawn from the interval $[5,10]$. This is an example of a variable tabu tenure. The only time when tabu moves are allowed to be implemented is if the tabu move results in a new incumbent solution, that is, if the new solution is an improvement on the best solution so far. This is considered as the aspiration criterion.

Infeasible solutions with respect to route duration and vehicle capacity are considered throughout the search process as a diversification technique to reduce the likelihood of being trapped in a local minimum. There are two penalty terms in the objective function to account for this, one factors in excess capacity and the other factors in excess duration, each weighted by a self-adjusting parameter. Every 10 iterations, if the previous solutions were feasible, the parameters are divided by 2, and if they were infeasible, they are multiplied by 2, otherwise the parameters remain unchanged.

Another diversification technique is to penalise frequently performed moves so as to increase the probability of considering slower moving vertices. The current vertex is assessed in terms of how often the move appeared in past solutions, the frequency of the move is then used to create a term with which to increase the objective function.

The false starts used to initiate the process is an example of an intensification technique used. In addition to this, if no improvement is found after a given number of iterations (less than the number of iterations without improvement for the stopping criterion), then a more intense search is conducted using the incumbent as a starting point. Also, more vertices are allowed to be moved for this re-initiation.

Re-optimisation occurs at various points in the search for the route where a vertex has just been inserted. This is done with the unstringing and stringing (US) post-optimisation heuristic. Each vertex in a tour is removed using the reverse of GENI and reinserted using GENI. The combined use of GENI and US is known as the GENIUS heuristic, the original text can be consulted for a more detailed explanation (Gendreau et al. 1992).

The process stops when there are no improvements for a given number of consecutive iterations.

In conclusion, taburoute consists of an initialisation phase, a solution improvement phase, and an intensification phase. The approach produces high quality solutions in an average amount of time. Given its total of nine user-controlled parameters, it is not the simplest method available. The GENIUS TSP heuristic also adds to the complexity of the method as this heuristic is not easy to replicate. Finally, due to the penalty mechanism employed, taburoute is relatively very flexible.

2.3.2 Unified tabu search

According to the literature the unified tabu search (UTS) was originally intended for periodic VRP and multi-depot VRP but was later extended to the classical VRPTW (Laporte & Cordeau 2002; Cordeau et al. 2002; Cordeau et al. 2001). The following is describes the UTS approach based on the work of the authors. For a more detailed description, the original paper can be consulted (Cordeau et al. 2001).

Before describing the procedure, some notation is necessary. $[e_i, l_i]$ is used to denote the time window for vertex i. That is, vertex i must be visited no later than time l_i. If vertex i is visited before time e_i, then a waiting time of $w_i = e_i - a_i$ is incurred, where a_i is the arrival time of the vehicle at vertex i.

The process starts with the generation of only one initial solution instead of several. This is done using GENI insertions as described earlier (Gendreau et al. 1998). Essentially though, for planar problems, vertices are sorted in increasing order of the angle that they make with the depot and their radius. For non-planar problems, any other arbitrary ordering can be used. The m routes are constructed by randomly selecting one of the n vertices and using it for the first route. Customers are allocated to this route in a way that minimised additional time added to the route, similar to the savings heuristic. If the capacity or route duration constraints are violated another route is started. To account for the time window constraint if vertex i is to be inserted between two vertices, say j and k, then the condition $e_j \leq e_i \leq e_k$ must hold, where e_x denotes the lower bound on the time window for vertex x. If this condition does not hold then vertex i is inserted at the end of the route. The initial solution S is then a set of s solutions consisting of m routes each which may violate capacity, route duration or time window constraints.

Before describing the neighbourhood structure, it is necessary to describe the tabu mechanism which functions on an attribute set $B(x)$ associated with solution x. The attribute set essentially records which vertex is visited by which vehicle, that is, $B(x) = \{(i,k): vertex\ v_i\ is\ visited\ by\ vehicle\ k\ in\ solution\ x\}$. The tabu tenure is fixed beforehand. This means, once a vertex is removed from a vehicle's route, reinsertion into that

route is forbidden for a fixed number of iterations. The aspiration criteria is defined relative to attribute (i, k) in that, moves declared tabu may only be accepted if it results in an improvement of the best solution so far with that specific attribute.

The neighbourhood structure involves the removal of attribute (i, k) from $B(x)$ and replaces it with (i, k') where $k \neq k'$. That is, a vertex v_i is removed from the one route k and is re-inserted into k' such that the objective function is minimised.

A penalised objective function is used as a diversification mechanism $f(s) = c(s) + \alpha q(s) + \beta d(s) + \gamma w(s)$ where $c(s)$ is the total travel costs and $q(s), d(s)$ and $w(s)$ are the capacity, route length and time window constraint violations, respectively. The parameters α, β and γ are self-adjusting so as to allow exploration of intermediate infeasible solutions. The implementation is similar to that used in the taburoute approach. At each iteration, if the previous solution is infeasible with regard to any of the three above mentioned constraints, the appropriate parameter is multiplied by $1 + \delta$, whereas if the previous solution is feasible, the appropriate parameter is divided by this term.

The process stops when a given maximum number of iterations have been performed. This is followed by a post-optimisation phase. There is no intensification procedure.

The entire procedure can be explained as follows. An initial solution s is created to be used as a starting point for the search. This solution is recorded as s^*, which represents the best solution so far, also referred to as the incumbent. If s is feasible then the cost is recorded as $c(s)$, otherwise the cost is recorded as being infinite. The parameters in the objective function $f(s)$ are initially all set to 1, and after each iteration, they are updated. The best solution, that is, the one that minimises the objective function, is selected from the neighbourhood of the current solution $N(s)$, provided that it is not tabu or satisfies the aspiration criterion. If the solution is feasible and better than the incumbent, it replaces s^*. The process is repeated for a given number of iterations after which a post-optimisation procedure is then applied to each route of the best solution so far using an adaptation of the GENIUS heuristic (Gendreau et al. 1992).

Overall, the unified tabu search algorithm scores high on accuracy and speed. The approach is very flexible and relatively simple given that it only has a few parameters. This approach has successfully been applied to variants of the VRP.

2.3.3 Adaptive memory process

Based on the literature (Taillard et al. 2001; Cordeau et al. 2002; Laporte et al. 2000; Laporte & Cordeau 2002) the adaptive memory process was developed by Rochat and Taillard. The approach is mostly used in tabu search but is not limited to this metaheuristic, like the granular tabu search, the device is portable. Also, the technique is not limited to solving the VRP and its variants.

Adaptive memory can be seen as 'probabilistic intensification and diversification'. If applied continuously throughout the search, it diversifies given that new combinations of the best solutions are produced. When used as a post-optimisation process, the adaptive memory procedure can be seen as an intensification technique.

The process is initiated by selecting vehicle routes from several solutions which are heuristically produced. The success of this heuristic is heavily dependent on the heuristic used for this initial generation of solutions. The good solutions are used to form a pool which can be viewed as initialising a memory structure.

At various points throughout the search elements of the solutions in the pool are selectively extracted and combined to form new solutions. Selection of a solution depends on the weight assigned to it, where routes with better solutions have a higher probability of being selected. Thus provisional solutions are then created by combining attributes of these good solutions recorded in the pool (memory) to form new ones. The attributes of these solutions can be seen as the neighbourhood structure. Similarly, reversal of them can be seen as the tabu mechanism.

Basically, after a pool is created, new partial solutions are made and then a local search is performed to improve these partial solutions. Finally the pool is updated. This process continues throughout the search and the memory is constantly updated as better solutions are added and the worst solutions are removed from the pool.

Double counting must be avoided though, that is, the same customer must not be included twice in one solution. To avoid this, the selection process will often terminate with a partial solution. The pool is scanned for tours that share customers and these are automatically excluded from being considered for reselection. Partial solutions are then formed probabilistically resulting in the process terminating with a set of selected routes and unrouted customers. Unrouted customers are included into the solution using some least cost insertion method such as the savings heuristic and if the new solutions are an improvement of current solutions in the pool, they will replace the worst older solutions.

The stopping criteria for the tabu search can either be a fixed number of iterations or a number of consecutive iterations without improvements.

The use of an adaptive memory adds to computing time, and some basic coding skill is needed but the process is not too involved. The approach scores high on flexibility given that it does not only have to be used for tabu search and VRP.

2.3.4 Granular tabu search

According to the work of several authors (Toth & Vigo 2003; Cordeau et al. 2002; Laporte & Cordeau 2002; Laporte et al. 2000), the granular tabu search was developed by Toth and Vigo and is explained below. The procedure can be applied to a variety of discrete optimisation problems instead of being limited to the VRP and its variants. In fact, the mechanism is portable as previously mentioned and need not only be used with tabu search.

The technique is based on the notion that longer edges of the graph have only a small probability of belonging to an optimal solution, thus, by eliminating these edges whose length, which translates to cost, exceed some granularity threshold, unpromising solutions will never even be considered by the search process. The idea of removing long edges is to ease the computational burden.

The process is initiated with a savings heuristic to find a starting point for the search. The number of vehicle routes are decided beforehand or may have an upper limit. If the savings algorithm results in a solution exceeding the maximum number of routes allowed then routes with the least customers are removed and their customers are inserted into the remaining allowed routes. The reinsertion of these customers are performed in a way which minimises costs involved. The initial solution may be infeasible with regard to capacity or route length constraints, these are accommodated for by using penalty terms in the objective functions in a similar manner as taburoute implements it. A lower and upper bound is set for the parameters of the penalty terms, for example parameters must lie in the range $[1,6400]$. Initially all parameters are set to 100, if the previous, say 10, iterations are feasible then the respective parameters are divided by 2, and if infeasible then the respective parameters are multiplied by 2. Note however that these alterations must lie within the allowable range.

The neighbourhood structure is a highly restricted neighbourhood referred to as 'granular'. It involves performing a limited number of edge exchanges within the same route or between different routes, that is, intra-route and inter-route exchanges. $E(\tau) = \{v_i, v_j) \in E: c_{ij} \leq \tau\} \cup I$ is the restricted edge set where, as mentioned, only routes shorter than some granularity threshold τ is considered, along with set I which is a set of important edges either linked to the depot or forms part of the best solutions found. The granular neighbourhood can be seen as an implementation of a candidate list strategy. Every $2n$ iterations a new neighbourhood is constructed using the mechanism described above, where n is the number of customers.

The tabu mechanism involves forbidding the reversal of moves, for example, if an arc was removed, its reinsertion is declared tabu for a number of iterations. The tabu tenure is variable, similar to the one used in taburoute, where θ is randomly chosen from the interval $[5,10]$.

Granular tabu search facilitates intensification and diversification. $\tau = \beta\bar{c}$ where β is a sparcification parameter usually chosen in the interval [1.0,2.0] and $\bar{c}$ is the average edge length of a solution obtained by a fast heuristic such as the savings algorithm mentioned earlier. If β is part of that interval, then 10-20% of the original solution space remain in the graph. This acts as an intensification procedure as a portion of the solution space is thoroughly searched. As a diversification technique, if the incumbent has not improved for a given number of iterations β increases and a new neighbourhood is constructed. A number of iterations are run and the best solution is used at the new starting point. After this β is factor set to its original value and the search continues. Thus when β is small, long intensifying steps are performed and when β is large, shorter diversifying steps are performed.

Taburoute is often used as a subroutine. Granular tabu search is easy to implement once a good underlying search algorithm is obtained. The approach is fast and produces high quality solutions.

3 FORMAL PROBLEM DESCRIPTION

One of the objectives of this investigation was to design a basic tabu search algorithm to solve the VRPTW. This algorithm was then tested on a subset of the Solomon 100 dataset (details of which are discussed in the following chapter). This chapter covers the details of the VRPTW that the basic tabu search algorithm was applied to. It is in fact the classical VRPTW. It is necessary to reiterate what this entails in detail. Some notation and the main assumptions used in the remainder of the paper will also be referred to in this chapter.

3.1 NOTATION

Consider an undirected graph $G = (V, E)$ where $V = \{v_0, v_1, \dots, v_n\}$ is a set of vertices representing customers (except for v_0 which represents the depot) and $E = \{(v_i, v_j) : v_i, v_j \in V, i < j\}$ is an edge set representing the distance between customers. Recall that $i = 1, \dots, n$, where n is the number of customers.

Each edge has an associated cost (c_{ij}) and travelling time (t_{ij}) that are symmetric in nature. Each customer has a corresponding non-negative demand (q_i), a service time (s_i) and a time window $[e_i, l_i]$ which denotes the earliest and latest times that a customer allows the service to start. Note that the completion of a service does not have to satisfy the time window constraint, merely the start of a service, thus the arrival time of a vehicle at customer i is denoted by a_i.

It is assumed that all vehicles leave at the earliest time where the time window $[e_0, l_0]$ denotes the earliest and latest times that a vehicle may leave and return to the depot, the length of this window represents the allowable route duration time *D.* At the depot is based a fleet of *m* homogenous vehicles, each with a capacity *Q*, where *m* is a decision variable that has an upper limit $\bar{m}$. Also note that the arrival time, a_i, for all customers are also decision variables, meaning that their values are determined by the solution.

3.2 MAIN PROBLEM

A set of customers must be assigned to each vehicle such the following conditions hold:

(i) all routes start and end at the depot,
(ii) each customer is visited exactly once by one vehicle,
(iii) the total demand of the customers for each route does not exceed the vehicle's capacity *Q,*
(iv) the total duration of each route, including travelling and service times, does not exceed *D,*
(v) the arrival time of a vehicle at customer i , a_i, does not exceed the upper limit of the time window l_i,
(vi) the number of vehicles, *m,* are minimised and do not exceed the upper limit $\bar{m}$, and finally,
(vii) the total routing cost in minimized for the given *m.*

To summarise, minimising the number of routes is the primary objective, and then for a given number of routes, the secondary objective is to minimise costs involved. All the while adhering to the given constraints and boundaries involved.

3.3 ASSUMPTIONS

Note that the following assumptions were made:

(i) The service time for each customer s_i varies according to their associated demand q_i.
(ii) If a vehicle arrives early, a waiting time $w_i = e_i - a_i$ is added to the service time.
(iii) The travelling cost c_{ij} only includes the distance travelled by each vehicle, the service time s_i and waiting time w_i are merely calculated to determine the arrival time of vehicles at their assigned customers.

These assumptions hold for all customers, and are best illustrated by an example.

Consider travelling from customer i to j, where the vehicle arrived at customer i too early, incurring a waiting time of $w_i = e_i - a_i$. Ideally, the travelling cost (c_{ij}) from customer i to j would consider this waiting time at customer (w_i), the service time at customer i (s_i) and finally the travelling time from i to j (t_{ij}). Effectively this can be summarised as $c_{ij} = w_i + s_i + t_{ij}$. However, given that not all authors record their costs in this manner, when comparing results to the best known solutions, inconsistencies may occur. Thus, only the travelling time from i to j (t_{ij}) will be regarded as the travelling cost (c_{ij}). Notice that travelling time t_{ij} is equivalent to the Euclidean distance between customers i and j. It is still necessary to consider waiting and service times, given that that these measures dictate the arrival of vehicles at customers, which in turn are used to determine whether or not time window constraints are adhered to. However, these measures are used separately when checking constraints as opposed to being used directly in the objective function to be minimised.

One final note that must be made is the justification for assuming that vehicles leave the depot at the earliest time possible. By starting at the earliest time, some customers can be pushed forward and inserted before others. This helps maintain feasibility with regard to the time window constraints which can be seen as the cause for the computational complexity of solving the VRPTW. A post-optimisation mechanism can be implemented to adjust the times of the final solution so as to eliminate unnecessary waiting times that may be incurred as a result of this assumption. (However, no post-optimisation mechanisms were employed in this study).

4 METHODOLOGY

When deciding which tabu search approaches to include in the construction of the basic tabu search algorithm for this study, the main consideration was the simplicity of an approach. Some of the state of the art approaches, for example taburoute, do not account for the time window constraints, hence extensions are necessary in some cases to consider both spatial

and temporal aspects of the problem. The tabu search approach tested on the data extracts elements from the approaches discussed in Chapter 2 and will be described in the following section. However, before the describing the basic tabu search developed throughout this study, the Solomon dataset will be introduced.

4.1 DATA

The data used for this study are referred to as the Solomon 100 customer problem set. A summary of the description based on the author is presented in this section (Solomon 1987). There are six problem sets. The problem sets differ based on several factors, namely the geographical spread of the data, the number of vehicles needed to service them, and the time window constraints, such as percentage of time-constrained customers, and width and positioning of time windows.

The six problem sets are denoted as R1, R2, C1, C2, RC1, and RC2. R1 and R2 represent random uniformly distributed customers, C1 and C2 denotes clustered customers, and RC1 and RC2 denote semi-clustered problems, that is, these problem sets contain a mixture of randomly distributed data and clusters. R1, C1 and RC1 all have short scheduling horizons, that is, narrow time windows, and small delivery capacity, meaning each vehicle can only satisfy a few customers. The remaining sets R2, C2 and RC2 have long scheduling horizons, that is, larger time windows, and larger delivery capacity, meaning more customers can be served by one vehicle. Thus it is expected that the number of vehicles used in these sets are lower than the number of vehicles used in the sets with short scheduling horizons.

Time window density refers to the percentage of customers with time windows. There exists problems with 25, 50, 75 and 100% time windows.

Together the six problem sets consists of 56 instances of the VRPTW, where customer locations and time windows may vary for each instance. Each data set includes the number of vehicles that are available and each vehicles capacity. Each data set also consists of the x- and y- coordinates of all vertices, the lower and upper time window bound of each vertex, the demand of each customer and the service time of each customer.

All the test problems contain 100 customers, however, only the first 25 of each problem set was used (given the high computation time needed for the set functions in AIMMS as mentioned in Chapter 1). This is known as the Solomon 25 customer problem set.

Finally, Euclidean distances between customers are used as the traveling times between them.

4.2 BASIC TABU SEARCH

The process was initiated with an algorithm that draws from the savings and nearest neighbour heuristics to produce one initial solution to be used as a starting point. The solution had a form similar to that of taburoute, that is, the solution is a set S of m routes $R_1, R_2, \dots, R_m$ where $m \in [1, \overline{m}]$ and $\overline{m}$ is the maximum number of routes allowed. Each route had the form $R_r = \{v_0, v_{r_1}, v_{r_2}, \dots, v_0\}$ and each vertex v_i, where $i \geq 1$, belonged to

only one route. The initial solution S was not allowed to violate any of the constraints. The initial solution was registered as the incumbent, that is, $S^* = S$ with cost $c(S^*) = c(S)$.

A sequential approach was used where one route was expanded at a time until addition of more customers was no longer feasible. The m routes were constructed by selecting the customer closest to the depot in terms of distance and urgency. That is, the customer with the shortest Euclidean distance and lowest upper bound time window was routed first. To achieve this a weighted sum of the two values was minimised. The first route then had the form $(0, i, 0)$. Customers were allocated to this route in a way that minimised additional time added to the route, similar to the savings heuristic. The idea was to focus on minimising a weighted combination of extra distance and extra time that resulted from adding an unrouted customer to the end of a route. To account for the time window constraints, urgency was also factored in. Thus the customer with the highest saving and lowest upper bound time window was added, provided the duration, capacity or time window constraints were not violated with its addition. If any of these constraints were violated, a new route was started and the process was repeated until no more unrouted customers remained.

The neighbourhood structure entailed a simple shift. That is, customers were removed from their current route and were inserted into another, or they may have been shifted to another position in their current routes.

The tabu mechanism used involved forbidding reverse moves using a tabu tenure of $\theta = 10$. All moves in a given neighbourhood were evaluated before checking the tabu status of these moves. All potential moves that had a tabu status were measured against the aspiration criterion to check for its admissibility, where its tabu status was removed if the tabu move resulted in a solution that was better than the incumbent. The accepted new solution was then compared with the current best solution. If the new solution was better than the incumbent and was feasible, it became the new incumbent.

A penalised objective function was used as a diversification mechanism $f(s) = c(s) + \alpha q(s) + \beta d(s) + \gamma z(s)$ where $c(s)$ is the total travel costs and $q(s), d(s)$ and $z(s)$ are the capacity, route length and time window constraint violations, respectively. The parameters α, β and γ were fixed to a value controlled by the user, an arbitrary value of 1 was used when testing the algorithm. Each constraint violation was calculated using the formula $x(s) = \sum_r [(\sum_{v_i \in R_r} x_i) - X]^+$, where $[X]^+ = \max\{0, x(s)\}$. This ensured that the constraint violation would not be negative.

The process stopped after the iteration limit was reached. A maximum of 100 iterations were allowed. No intensification or post-optimisation procedures were employed in attempt to keep the algorithm as simple as possible.

4.3 Implementation in a modelling language

The following provides a detailed explanation of how the basic tabu search algorithm was implemented in the modelling language AIMMS. (It is assumed that the reader has a basic

knowledge of AIMMS). There are three main procedures, that is, the Main Initialisation, the Main Execution and the Main Termination, each of which will be covered in turn.

It should be noted that the Main Execution procedure consists of four sub-procedures which can be thought of as splitting the algorithm into sections or tasks. Since the algorithm required an initial solution, the first sub-procedure 'BuildFirstTour' was responsible for constructing an initial tour. The second sub-procedure 'InitialObjectiveFunction' then calculated the objective function value for this initial solution and conducted feasibility checks. It was not strictly necessary to separate this into a second procedure, but for the sake of monitoring feasibility during the construction of the algorithm it was included to confirm that time window, vehicle load and route duration constraints were adhered to. The third sub-procedure 'Neighbourhood' initialised all the information needed to carry out the tabu search. Finally, the last sub-procedure 'TabuSearchMetaheuristic' conducted the actual tabu search.

4.3.1 Main initialisation

The Main Initialisation is a preliminary procedure which was responsible for reading the data into AIMMS, automatically updating the relevant information for each customer, specifically their location, demand, time windows and required service time. The x and y coordinates were used to calculate the Euclidean distance between all possible customer pairs. The number of customers was set to 25 given that only the top 25 customers of the Solomon 100 dataset were used. This value can be changed by the user for other problem sizes. The vehicle capacity and maximum number of vehicles that are allowed were also specified in this section. Vehicle capacity and number of vehicles were set to the values specified in the respective datasets. The vehicle capacity varied between datasets taking on values of 200, 700 or 1000 whereas the maximum number of vehicles available was always 25. Once again this can also be specified by the user. Finally, the iteration limit that was used as the stopping criterion in the tabu search was defined. This value is controlled by the user.

4.3.2 Main execution: Initial tour construction

Before describing how an initial tour was constructed, it is necessary to cover some notation and describe some of the variables used in the algorithm.

The process was initialised by creating a set 'Vehicles' consisting of the 25 vehicles allowed to be routed and a set 'Customers' containing 26 vertices (one for the depot and one for each of the 25 customers). Subsets of these sets denoted 'RemainingVehicles' and 'RemainingCustomers' were also created so that once a customer had been allocated to a vehicle, it was removed from the RemainingCustomers set. The intention was to exhaust this set so as to allocate all the customers to a vehicle. Note that it was not necessary to exhaust the RemainingVehicles set, as these were simply the maximum number of vehicles available to be routed. Furthermore, another subset of the Customers set, denoted 'RemainingFeasibleCustomers' was created. Its relevance will become apparent at a later stage. It should be noted that both subsets of the Customers set included the entire set except the Depot.

An empty two-dimensional set 'CustomerToVehicle' stored the set of customers belonging to each vehicle. This set simply illustrated whether or not a customer belonged to a vehicle, it

did not specify the order of the customers in each vehicle. Instead a two-dimensional parameter 'CusToVeh' was used to display this information. Furthermore two element parameters 'InitialNext' and 'InitialPrev' were created to illustrate, respectively, the customers succeeding and preceding each customer. The 'InitialTourConnect' matrix displayed which arcs are used in a particular solution was also created. Finally, parameters were created to store the arrival, departure and waiting times of a vehicle at each customer.

To monitor capacity and duration constraints, the vehicle capacity specified in the Main Initialisation and the upper bound time window of the Depot were used as the maximum allowable vehicle load and route duration amounts, denoted as 'RemainingCapacity' and 'RemainingDuration', respectively.

The focus is now shifted to a detailed explanation of the route construction algorithm. A loop was used to select each of the vehicles in turn. For each new vehicle selected, the following was carried out. The selected vehicle was removed from the RemainingVehicles set. RemainingDuration and RemainingCapacity parameters were initially set to the maximum allowed amounts. The depot was then added to both the CustomerToVehicle set matrix and the CusToVeh parameter matrix. Using a second loop which sifts through the RemainingCustomers set, the first customer was added to the selected vehicle based on the criteria explained previously, that is, the most urgent customer with the smallest distance from the Depot. No feasibility checks were necessary for the first customer added to each vehicle. Once a customer was added to a route it was removed from both the RemainingCustomers and RemainingFeasibleCustomers sets.

In fact, once a customer was added to a vehicle a sequence of events occur. The selected customer was added to the CustomerToVehicle set and to the CusToVeh parameter (in the appropriate sequence number) for the selected vehicle. The selected customer also became the next customer of the previously selected customer in the 'InitialNext' element parameter. The arrival, waiting time and departure time of the customer was then updated. Finally, the RemainingCapacity and RemainingDuration were reduced by the demand of the selected customer and the total travelling time of the customer respectively. Recall that the total travelling time of a customer is the sum of the distance from the previous customer to the current customer, the waiting time of the current customer and the service time of the customer.

After the first customer was added to a route, the RemainingCustomers set was then sifted through a second time to complete the rest of the route based on the criteria explained previously, that is, the most urgent customer with the maximum saving. However if no more customers remained, for example, the first customer added to the route was the last customer in the set of remaining customers, then the loop was instructed to break and the initial tour was complete. Furthermore, before adding a potential customer to a route, it was necessary to check that duration, capacity and time window constraints were adhered to.

If the demand of a potential customer was greater than the RemainingCapacity amount, it was considered infeasible. It was easier to check for a capacity violation as only information about a customer's demand was needed. If the capacity constraint was not violated, the

arrival and waiting time of the potential customer was calculated so that the duration constraint could be monitored. For the duration constraint to be violated the time needed for the vehicle to travel from the current customer to the potential customer, carry out its service at the potential customer and return to the depot had to be greater than the RemainingDuration amount. To monitor the time window constraint, for the potential customer to be feasible, the arrival of the vehicle designated to the customer had to be before the upper bound time window for that customer.

If none of the above constraints were violated, the most appropriate customer was added to the route at each iteration until the RemainingCustomers set was exhausted. However, if constraints were violated, the infeasible customer was removed from the RemainingFeasibleCustomers set and a third loop was created to sift through this set and complete the current route. Every time a constraint was violated in this third loop, the infeasible customer was removed from the RemainingFeasibleCustomers set and the next feasible customer was sought until the set was exhausted. Once this set had been exhausted, a new vehicle was selected.

Notice that infeasible customers were not removed from the RemainingCustomers set as these customers still had to be routed and may have only been infeasible for that particular vehicle. Thus, as soon as a constraint was violated the RemainingFeasibleCustomers was reset to equal the RemainingCustomers set, and infeasible customers were systematically removed until a new vehicle was selected.

Before selecting a new vehicle, though, the route of the current vehicle had to be closed by directing the vehicle back to the Depot. Thus for the last selected customer, its next customer in the InitialNext element parameter was the Depot. The InitialPrev element parameter and the TourConnect parameter matrix was updated.

Also, before starting the next iteration and selecting a new vehicle, it first had to be determined whether or not the RemainingCustomers set was empty or not. Once all customers had been routed, the procedure terminated and the arrival of the last vehicle at the Depot was recorded. It should be noted that only the arrival of this last vehicle allocated customers was recorded. This was merely done for the sake of completeness. The arrival of each vehicle at the depot was recorded in the following sub-procedure as this value was equivalent to the duration of the respective route.

4.3.3 Main execution: Evaluate initial solution

This sub-procedure was relatively straightforward compared to the previous one. Essentially all relevant information necessary to calculate the objective function was prepared. The objective function was then evaluated and finally, the feasibility of the initial tour was determined.

The procedure was started by arbitrarily setting the parameters α, β and γ to 1. As previously mentioned, these parameters can be altered by the user as desired.

The total duration of each vehicle's route was calculated using the TourConnect matrix as an indicator variable for particular customer pairs and whether or not the travelling time of the

given arc should have been considered. Vehicle loads were also calculated using the CustomerToVehicle sets, by summing the demands of all the customers allocated to a given vehicle.

The capacity, duration and time window violations were all calculated as previously explained in section 4.2. Although, it was expected that for this initial tour, all violations should have been zero. The objective function value was then calculated using the formula provided earlier.

As previously mentioned, feasibility checks were conducted for the sake of monitoring the progress of the initial tour construction. A feasibility parameter for each of the constraints were calculated, where each parameter was given a value of 1 if the constraint was adhered to. A 'TotalFeasibility' parameter then summed the three constraint feasibility parameters and had to have a value of 3 for the solution to be considered feasible.

4.3.4 Main Execution: Initialise Neighbourhood

In order to conduct the tabu search in the final sub-procedure of the Main Execution, certain variables had to be initialised.

It should be noted that when searching a neighbourhood for the best possible move, two potential moves were compared. Thus 'TrialOne' and 'TrialTwo' denote the two potential moves compared when exploring the neighbourhood at each iteration.

CustomerToVehicle sets and CusToVeh, or rather, 'SolutionMatrix' matrices, were created to store the Incumbent, Current, TrialOne, TrialTwo, and Initial solutions. Parameters and element parameters were also created to store the ObjectiveFunctionValue, NextCustomer, and PreviousCustomer information for each of the five solution types above.

A two-dimensional parameter 'TabuList' was used to denote the tabu list. The ten rows represented the tabu tenure and the four columns represented the reversals of the vehicle the customer was being moved from $(V1)$, the position the customer was being moved from $(P1)$, the vehicle the customer was being moved to $(V2)$ and the position the customer was being moved to $(P2)$. The algorithm could not be executed with an empty matrix, thus the TabuList was initiated to contain 1's instead of 0's. By letting every value in every row equal 1, the tabu moves were simply the redundant move of moving the first customer in the first vehicle to the first customer in the first vehicle, that is, not moving this customer at all. In conclusion, each row in the TabuList had the form $(V1, P1, V2, P2)$ which was forbidden.

The RemainingCustomers set was reset to contain the entire Customers set excluding the Depot. A copy of the NextCustomer and PreviousCustomer for the current solution as well as a copy of the tabu list, TabuListCopy, were created, the relevance of which will become apparent in the following sub-procedure.

4.3.5 Main Execution: Tabu Search

This process was carried out until the stopping criterion of the iteration limit was reached. In terms of computation time, this was the most intensive portion of the algorithm and required several minutes to run whereas the initial tour construction required less than ten seconds.

As previously mentioned, two potential moves were compared at each step when exploring the neighbourhood. The first potential move was conducted and evaluated. It consisted of simply shifting the first customer from the first route into the position of the second customer of the first route. The objective function value of this TrialOne solution was stored.

To compute the second potential move, TrialTwo, first the number of vehicles used was calculated and stored. This value controlled a loop which allowed every vehicle that had customers routed to it an opportunity to be $V1$. A second nested loop then allowed every vehicle that had customers routed to it an opportunity to be $V2$. A third and fourth loop then allowed every non-empty position the opportunity to be $P1$ and $P2$, using the CustomerToVehicle set to identify the number of non-empty positions in each vehicle for the current solution.

Exploring the neighbourhood in this manner may have resulted in some redundant moves which added to computational time. Thus, in an attempt to improve the efficiency of the algorithm only cases where $V1 \neq V2$ and $P1 \neq P2$ were considered as potential moves for TrialTwo.

As the neighbourhood was explored, every new potential move TrialTwo was compared with TrialOne and if the ObjectiveFunctionValue amount for TrialTwo was less than or equal to the amount for TrialOne then TrialOne was replaced with TrialTwo. However, this only occurred if TrialTwo was not on the TabuList. If TrialTwo was on the tabu list, but met the aspiration criterion of being less than the Incumbent, then it was allowed to replace TrialOne. Otherwise, the next potential move was evaluated as TrialTwo.

Every time a potential move was considered for TrialTwo, the following sequence of events occurred. The customer being shifted was removed from its current vehicle and position in the SolutionMatrix parameter for TrialTwo. Then the remaining customers in that vehicle were shifted appropriately by one position to the left, that is, each customer's position decreased by one sequence number if its original sequence number was greater than that of the customer being removed. The insertion of the customer being shifted then occurred, however different mechanism were used depending on if the insertion was into a different position in the same vehicle or if the customer was to be inserted into a different vehicle's route altogether. Furthermore, if a customer was shifted elsewhere within the same vehicle, different mechanisms were used depending on whether $P1 < P2$ or if $P1 > P2$. All mechanisms however had the same purpose, that is, to make space for the customer being inserted into the route by shifting all appropriate customers to the right by one position. Finally, the customer was inserted into the SolutionMatrix parameter for TrialTwo.

Furthermore, if a customer was shifted into another vehicle the CustomerToVehicle set for TrialTwo was updated, removing and adding the shifted customer from and to $V1$ and $V2$ respectively. The NextCustomer and PreviousCustomer information for TrialTwo was also updated. For every removal, the NextCustomer of the customer preceding the shifted customer, referred to as 'OutPrev' changed and became the customer that initially succeeded the shifted customer, referred to as 'OutNext'. Also, the PreviousCustomer of the customer succeeding the shifted customer, OutNext, became OutPrev, that is the customer initially

preceding the customer being shifted. For every insertion, the NextCustomer of the shifted customer became the customer it was 'replacing', referred to as 'OCustomer' and the PreviousCustomer of the customer being shifted was the initial PreviousCustomer of the OCustomer. Finally, the customer being shifted became the NextCustomer and the PreviousCustomer of the initial PreviousCustomer of the OCustomer and the OCustomer, respectively.

Once all of this was completed, only then could the ObjectiveFunctionValue amount of TrialTwo be calculated and a two-dimensional parameter containing $V1$, $P1$, $V2$, and $P2$ in all ten of its rows was then compared row by row to the TabuList to determine the tabu status of TrialTwo.

When updating any solution type, that is, replacing the TrialOne with TrialTwo solution, replacing the Current with TrialOne solution or replacing the Incumbent with Current solution, the following variables were updated: NextCustomer, PreviousCustomer, SolutionMatrix, CustomerToVehicle and ObjectiveFunctionValue. The vehicles' load and duration as well as arrival information of all of the customers were updated as it was used for feasibility checks at a later stage. Finally, the vehicle and position information, $V1$, $P1$, $V2$, and $P2$, was also stored, in case it was needed to update the TabuList at a later stage.

The whole procedure is rather lengthy, but to avoid the complexity of swapping between procedures the TabuList, Current and Incumbent solutions were also updated in this sub-procedure. Thus, once the neighbourhood had been exhausted, the Current solution was replaced by TrialOne. The TabuList was then updated by copying the top nine rows of the TabuList to the bottom nine rows of TabuListCopy, the newest reversal was then placed in the first row of TabuListCopy using the vehicle and position information previously stored. Finally the TabuListCopy was copied back to the original TabuList.

The last step before making the next move was to update the Incumbent solution if necessary. If the Current solution had an ObjectiveFunctionValue less than the Incumbent's, a feasibility check for the Current solution was conducted and only if it was feasible, did the Current solution replace the Incumbent solution.

As mentioned earlier, the process terminated when the iteration limit was reached.

4.3.6 MAIN TERMINATION:

This procedure simply returned the value 1, notifying AIMMS that the Main Execution had terminated.

4.4 PSEUDO-CODE

The following summarises the entire algorithm as a pseudo-code. It should be noted that RemainingVehicles is denoted as RV, RemainingCustomers is denoted as RC, RemainingFeasibleCustomers is denoted as RFC and ObjectiveFunctionValue is denoted as OFV.

1. Main Initialisation
2. Main Execution

a. Initial Tour Construction
    i. Initialise appropriate information
    ii. Select a vehicle from RV
    iii. Remove selected vehicle from RV
    iv. Initialise appropriate information for selected vehicle
    v. Identify first customer to be added to the route using appropriate criteria
    vi. Add customer to the route
    vii. Remove customer from RC
    viii. If RC is empty: STOP and go to b
    ix. If RC is not empty: Go to a.x to complete the rest of the route
    x. Select a potential customer from RC using appropriate criteria
    xi. Check for constraint violations
    xii. If no constraints are violated go to a.vi
    xiii. If either constraint violated go to xiv
    xiv. Select a potential customer from RFC using appropriate criteria
    xv. Check constraints
    xvi. If no constraints are violated go to a.xvii
    xvii. Add customer to route
    xviii. Remove potential customer from RC and RCF
    xix. If RFC not empty: go to a.xiv
    xx. If RFC empty: close route and go to a.ii
    xxi. If either constraint is violated go to a.xxii
    xxii. Remove potential customer from RCF and go to a.xiv
b. Evaluate Initial Objective Function Value
c. Initialise Neighbourhood
d. Tabu Search Metaheuristic
    i. Add 1 to LoopCount
    ii. If iteration limit reached: go to 3
    iii. If iteration limit not reached: go to b.iv
    iv. Create a TrialOne solution
    v. Create a TrialTwo solution
    vi. If TrialOne OFV $>$ TrailTwo OFV then go to b.v
    vii. If TrialOne OFV $\leq$ TrailTwo OFV then go to b.viii
    viii. Check tabu status of TrialTwo
    ix. If TrialTwo is not tabu go to b.xiv
    x. If TrialTwo is tabu go to b.xi
    xi. Check aspiration criterion
    xii. If aspiration criterion met: go to b.xiv
    xiii. If aspiration criterion not met: go to b.v
    xiv. Replace TrialOne with TrialTwo
    xv. If neighbourhood exhausted: go to b.xvii
    xvi. If neighbourhood not exhausted: go to b.v
    xvii. Replace Current with TrialOne
    xviii. Update TabuList

xix. Compare Current with Incumbent
xx. If Current $>$ Incumbent: go to b.i
xxi. If Current $\leq$ Incumbent go to b.xxii
xxii. Check feasibility of Current
xxiii. If Current is not feasible: go to b.i
xxiv. If Current is feasible: go to b.xxv
xxv. Replace Incumbent with Current
xxvi. Go to b.i

3. Main Termination

5 RESULTS & DISCUSSION

The following section discusses the performance of the basic tabu search algorithm in terms of accuracy, flexibility, robustness and speed. The algorithm was coded in the software package AIMMS and was run on a device with Intel® Core™ i5-3210M CPU, 2.50GHz/ 4GB of RAM. The AIMMS code is available as an appendix (see Appendix B). As mentioned in the previous chapter, only the 25-customer Solomon dataset was used to test the tabu search algorithm.

Recall, Chapter 2 referred to accuracy as the level of deviation of a solution value to the optimal solution that exists. Thus, to determine the performance of the algorithm in terms of accuracy, the costs generated by the algorithm are compared with the best known solutions for the Solomon dataset, these figures as well as the data are available on Solomon's webpage (Solomon 2005). The authors who produced the best known solutions appear as abbreviations in Table 1 and are referred to on the webpage, alternatively a legend (Solomon 2005) for Table 1 is provided as an appendix (see Appendix A). It should be noted that the results for the best known solutions appear in an identical format as Table 1 and are only concerned with solution quality in terms of meeting the objectives. Recall, the primary objective was to reduce the number of vehicles necessary, and the secondary objective was to reduce the total travelling distance, that is, cost, given the number of vehicles used.

Table 1 displays the results for the 56 instances of the Solomon dataset, where the problem number is shown in the first column under the heading 'Name'. The number of vehicles used are denoted as 'NV' and the total cost or total distance travelled by the vehicles are denoted by 'TD'. The table is split into three sections, displaying the results of the best known solutions and the authors responsible for them in the columns two to four. The results of the initial solutions generated by the initial construction portion of the algorithm are displayed in columns five and six. Finally, the number of vehicles and total distance travelled for the final solutions produced by the basic tabu search algorithm are displayed in the last two columns of Table 1.

It should be noted that none of the literature consulted display the actual routes generated by their algorithms. Thus it is only possible to make comparisons with regard to the quality of the primary and secondary objectives. However, the final routes generated by the basic tabu search algorithm developed in this study are available as an appendix (see Appendix C).

Table 1: Solomon 25-customer dataset results

Name	Best NV	Best TD	Author	Initial NV	Initial TD	Final NV	Final TD
C101	3	191.3	KDMSS	3	241.99	3	236.16
C102	3	190.3	KDMSS	3	295.24	3	181.60
C103	3	190.3	KDMSS	3	391.44	3	318.74
C104	3	186.9	KDMSS	3	393.13	3	357.18
C105	3	191.3	KDMSS	3	266.58	3	266.58
C106	3	191.3	KDMSS	4	319.44	3	264.75
C107	3	191.3	KDMSS	3	258.71	3	243.81
C108	3	191.3	KDMSS	3	505.54	3	230.37
C109	3	191.3	KDMSS	3	257.49	3	257.49
C201	2	214.7	CR+L	2	328.90	2	320.30
C202	2	214.7	CR+KLM	2	373.53	2	305.97
C203	2	214.7	CR+L	2	443.58	2	373.66
C204	2	213.1	CR+KLM	2	477.35	2	415.69
C205	2	214.7	CR+L	1	336.8	1	336.8
C206	2	214.7	CR+L	1	347.23	1	347.23
C207	2	214.5	CR+L	1	294.16	1	294.16
C208	2	214.5	CR+L	1	296.85	1	294.16
R101	8	617.1	KDMSS	8	669.66	8	669.66
R102	7	547.1	KDMSS	7	707.03	5	567.58
R103	5	454.6	KDMSS	6	648.10	5	579.31
R104	4	416.9	KDMSS	5	609.21	4	548.03
R105	6	530.5	KDMSS	7	616.09	7	616.09
R106	3	465.4	KDMSS	6	613.57	5	490.33
R107	4	424.3	KDMSS	5	629.65	5	503.60
R108	4	397.3	KDMSS	5	622.32	5	518.00
R109	5	441.3	KDMSS	5	666.91	4	590.13
R110	4	444.1	KDMSS	5	609.72	3	535.12
R111	5	428.8	KDMSS	6	643.08	5	533.47
R112	4	393	KDMSS	4	566.17	4	486.39
R201	4	463.3	CR+KLM	2	737.58	2	698.57
R202	4	410.5	CR+KLM	2	780.32	2	698.2
R203	3	391.4	CR+KLM	3	779.74	3	691.11
R204	2	355.0	IV+C	2	772.86	1	595.70
R205	3	393.0	CR+KLM	2	787.67	2	675.87
R206	3	374.4	CR+KLM	2	821.29	2	563.84
R207	3	361.6	KLM	2	822.00	1	636.421
R208	1	328.2	IV+C	2	749.16	2	577.10
R209	2	370.7	KLM	2	764.55	2	704.33
R210	3	404.6	CR+KLM	2	857.25	2	694.34
R211	2	350.9	KLM	2	849.26	2	641.61
RC101	4	461.1	KDMSS	5	531.38	4	394.25
RC102	3	351.8	KDMSS	4	435.60	3	204.84
RC103	3	332.8	KDMSS	4	492.61	4	257.24
RC104	3	306.6	KDMSS	5	525.71	4	265.93
RC105	4	411.3	KDMSS	5	548.75	4	248.21
RC106	3	345.5	KDMSS	5	654.55	4	275.20
RC107	3	298.3	KDMSS	4	488.70	3	181.15
RC108	3	294.5	KDMSS	4	548.83	3	202.30
RC201	3	360.2	CR+L	2	830.31	2	746.97
RC202	3	338.0	CR+KLM	2	729.86	2	452.60
RC203	3	326.9	IV+C	2	673.60	2	328.26
RC204	3	299.7	C	2	721.20	1	415.77
RC205	3	338.0	L+KLM	3	910.67	2	475.73
RC206	3	324.0	KLM	2	940.30	2	554.61

RC207	3	298.3	KLM	2	779.05	2	418.16
RC208	2	269.1	C	2	875.38	2	520.35

With regard to the primary objective of reducing the number of routes, the basic tabu search algorithm outperformed the best solutions for 21 out of the 56 instances. Routes were reduced mainly for instances with longer scheduling horizons, particularly for the RC2 set, where the number of vehicles were reduced in 7 out of the 8 instances in this set.

It should be noted that even though the algorithm reduced the number of vehicles used for a significant number of instances, the costs for these solutions were higher than that of the best known solutions. Strictly speaking, a solution with fewer routes is always preferred to one with more, regardless of distance travelled. That being said, it is worth investigating how the algorithm performed in terms of the secondary objective.

Overlooking the number of vehicles used, the basic tabu search algorithm reduced the total travelling distance in 9 of the instances. However, strictly speaking, travelling distance was only reduced for 6 instances, given that lower costs were incurred for these instances but the number of routes matched that of the best known solutions. With regard to the secondary objective, the algorithm favoured instances with shorter time horizons, particularly instances from the RC1 set.

Overall, there were improvements with regard to solution quality in terms of the two objectives for 27 of the 56 instances. The algorithm favoured the mixed datasets consisting of both random and clustered customers for both of the objectives. It should be noted that improvements on instances did not overlap, meaning the algorithm either reduced the number of vehicles used, without reducing total distance travelled, or it reduced the total distance travelled for the same number of vehicles, but never both.

To investigate this result a bit further, average percentage deviations in travelling distance (cost) from the best known solutions for each of the six dataset types are displayed in Table 2. What this means is that for all the instances in a given set, the difference between the best solution's cost and the cost generated by the algorithm was recorded for every instance and then the average was calculated. This was done to highlight which sets were favoured by the algorithm with regard to reducing distance travelled. Notice that Table 2 only examines the performance of the algorithm in terms of the secondary objective.

Table 2: Percentage deviations from costs

Set Name	**R1**	**C1**	**RC1**	**R2**	**C2**	**RC2**
Cost deviation (%)	+20	+38	-28	+71	+57	+53

Once again it can be concluded that the algorithm favoured instances with short scheduling horizons with regard to the secondary objective. For example, average costs were 20% higher for instances from the R1 set but 71% higher for instances from the R2 sets. Again, it should be noted that these costs do not mean much without considering the number of routes. For example, instances in RC2 had the number of routes reduced, and thus performed better than

the best known solutions despite the fact that instances in this set experienced on average 53% higher travelling distance. Overall, the basic tabu search solutions were on average 36% higher than the best known solution in terms of travelling distance.

It is interesting to note as to why the algorithm was able to reduce the number of routes but did not perform as well with regard to reducing the travelling distance for these routes. One possible explanation is that since the primary objective was to reduce the number of vehicles, this is what was driving the search. Recall from Chapter 4, only non-empty vehicles were considered when shifting a customer. Thus, once a vehicle was empty, it was not allowed to be considered as part of a solution given that a solution with fewer vehicles had been found. Furthermore, it was discovered that despite the tabu mechanism, cycling still occurred. As mentioned in Chapter 2, this happens when the neighbourhood structure used is not powerful enough to tackle the problem.

The following provides an explanation of what is meant by cycling and how it was detected. During the construction of the algorithm, it had to be ensured that the tabu mechanism was in fact working. Instead of running the algorithm for one hundred iterations and accepting the final answer, the current solution was inspected at every iteration for, say, the first ten iterations. Thus once a move was implemented, it was necessary to check that the reversal of the move was added to the tabu list. This is best illustrated with an example.

Consider an arbitrary route (0, 1, 2, 3, 4, 0). Suppose the best move in the shift neighbourhood of this route is (0, 1, 3, 2, 4, 0) where customer 2 is moved from position three to position four in the sequence. The reversal of this move, that is, moving a customer from position four back to position three becomes tabu. Now suppose the best move in the neighbourhood of this new solution is (0, 1, 3, 4, 2, 0). Once again the reversal of this move, that is, moving a customer from position five back to position four is added to the tabu list. Finally, suppose the best move in this neighbourhood results in (0, 1, 2, 3, 4, 0) and the reversal of this move is simply moving customer in position three back to position five. It is interesting to note that the solution is the same as the initial arbitrary solution, and because the same neighbourhood structure will be used, the same three solutions will be cycled for the remainder of the search. This cycling is reflected in the tabu list by having the same few entries repeated on the list.

When running the full search until the stopping criterion of reaching the maximum number of iterations, the tabu list at the final iteration was observed. It was found, for every instance of the Solomon dataset, that the same few entries appeared on the tabu list. This was a clear indication that the current solution was repeatedly being cycled over the same set of solutions, implying that a local optimum had been reached and that the incumbent would not improve regardless of the number of iterations. Technically, all constraints of the tabu search method were adhered to, so the issue of cycling was not as a result of the algorithm being incorrectly implemented or something that could be remedied by simply tweaking the features of the algorithm, for example, changing the tabu tenure. Rather, the shortcoming stemmed from the type of neighbourhood structure that was used. Thus using a more powerful neighbourhood is one possibility for overcoming this cycling issue.

As mentioned in Chapter 1, the algorithm was not competitive with regard to speed given the programming language used. However, ten minutes is a reasonable computation time given the nature of the problem. For example, if the nature of the problem entailed routing ambulances, speed would have been a more important consideration. Also, given that the algorithm performed reasonably well and managed to improve performance for 27 out of the 56 instances, the computation time is acceptable. However, as a result of the cycling that occurred, as previously discussed, the algorithm may have endured unnecessary computation time, especially if cycling occurred long before the 100^{th} iteration. If for example, a local optimum was reached at the 20^{th} iteration, then all remaining iterations and computation time was unnecessary given that the algorithm would have produced the same result at the 20^{th} and 100^{th} iteration. One possible way to remedy this would be to use a more sophisticated stopping criterion, for example, to stop the search if the incumbent is not improved for a given number of iterations. The challenge then however would be to determine what this number of iterations should be.

With regard to the simplicity of the algorithm, recall that simplicity refers to how easy the method can be replicated. Given the simple features used such as a single shift neighbourhood structure, and the absence of intricate intensification, post-optimisation features and user-controlled parameters, the basic tabu search algorithm scores high in terms of simplicity.

Finally, with regard to flexibility, the algorithm also performs well give that the procedure can easily be adapted to accommodate different type of problems. For example, changes to vehicle capacity and time window constraints. However, the algorithm is limited with regard to the size of problems it can tackle due to the inefficient computation time. For example, if the problem size doubled to 50 customers, the cardinality of even the simple shift neighbourhood would increase exponentially, meaning that computation time would also increase exponentially. Thus the basic tabu search algorithm developed in this study is best suited for smaller problem sizes.

One additional point worth discussing is recording results in terms of distance travelled and not total travelling time. The difference being that the latter includes service and waiting times whereas the former only considers the time needed to travel between customers, equating this value with the Euclidean distance between customers. Specifically for the Solomon dataset, given that service times for all customers are equal, it is justified in not considering the service time when calculating the final cost of the tour. Furthermore, because service times and waiting times are factored in with regard to ensuring that all time constraints, such as the time window and route duration constraints, are adhered to, it may also be justified in this respect to exclude travelling times from the final costs. However, consider an example where two routes with equal travelling distances exist. Say for instance, the first route however covers the same distance in less time because it experience lower waiting times than the second route. According to the current standard for measuring costs, both routes perform equally well, but in reality, the first route should be preferred given that the same distance is covered and the same customers are served faster. Thus in this respect, measuring cost only in terms of distance travelled is not a true reflection of all the costs

involved. However, given that the current best known solutions do not include the waiting times in their costs, it was not possible to compare performances in this respect.

6 CONCLUSIONS

The purpose of this investigation was to create a basic tabu search algorithm in attempt to solve the VRPTW. The following section summarises the main findings of the study, commenting on the performance of the basic tabu search algorithm developed, its shortcomings and ways to overcome them. The main takeaway from this study however would be that, even though the literature highlights the tabu search metaheuristic to be the top performing approach when solving instances of the VRPTW, not all implementations of the method are equal.

The basic tabu search algorithm performed reasonably with regard to accuracy, given the limitations of the investigation. It was found that the algorithm favoured longer scheduling horizons with regard to meeting the primary objective of reducing the number of vehicles and favoured shorter scheduling horizons with regard to meeting the secondary objective of reducing total distance travelled. That being said, performance of the algorithm was not consistent across all 56 instances of the dataset.

The algorithm is relatively simple and can easily be replicated and adapted to new instances, however, it is limited in its flexibility with regard to the problem sizes it can accommodate given the computation time needed. With regard to computation time, the algorithm was not particularly competitive, however, an average of ten minutes per instance was reasonable considering the nature of the problem. Furthermore, the algorithm was not efficient and incurred unnecessary computation time due as a result of cycling and using a maximum number of iterations as a stopping criterion.

In summary, despite improving on some of the best known solutions, improvements were not consistent across all instances. Furthermore, given the shortcoming of cycling repeatedly over solutions once a local optimum had been reached, the algorithm only explores a limited portion of the solution space. Had cycling not occurred, it would have been possible to explore more paths not yet travelled. Also, as a result of the cycling, the algorithm displays inefficiencies given that unnecessary computation time is incurred. Overall, the tabu search algorithm created in this study cannot be considered as one of the best implementations of the tabu search method. It appears that simple implementations cannot always compete with the more cumbersome approaches. That being said, there is potential for improvement. Chapter 7 covers suggested recommendations, where improvements are possible at the expenses of simplicity, given that a minimum level of complexity is necessary to improve on the algorithm in its current form.

7 RECOMMENDATIONS

The following section reflects how the basic tabu search algorithm developed in this paper can be improved in terms of its implementation, application and overall design.

With regard to implementation in AIMMS, the CPLEX SOLVER function could be used instead of set functions. Although more computationally involved and aimed at more advanced AIMMS users, this may speed up the generation of results as set functions require a generous amount of time to exhaust a neighbourhood at each iteration. Alternatively, another programming language could be considered altogether.

With regard to application, if the algorithm performs well (using the Solomon dataset as a benchmark) it can be applied to the relatively large VRPTW instance found in the sugar industry of Southern Africa.

With regard to design and overall efficiency, several major improvements are possible. Stepping through the pseudo code at the end of Chapter 4, recommendations for improvement are presented for each section of the algorithm in the following section.

In the Main Initialisation, the maximum number of vehicles available, vehicle capacity, and number of customers must be specified. It is possible to design the algorithm where this information is automatically extracted from the data as opposed to having the user manually enter these values.

When constructing the first tour, the relative weights of the urgency and distance or savings could be altered when considering a customer to be added to a route. Also, instead of only adding additional customers to the end of each route, customers could be inserted in any feasible position, this allows more than one route to be considered when adding a customer to a route. To factor in the time window constraints, if vertex i is to be inserted between two vertices, say j and k, then the condition $e_j \leq e_i \leq e_k$ must hold, where e_x denotes the lower bound on the time window for vertex x. If this condition does not hold then vertex i may be inserted at the end of the route.

The initial tour construction could be even more efficient with regard to capacity considerations, for example, the first customer added to the route could have the greatest demand, and the remaining customers should have smaller demands.

Finally, the initial solution S could be allowed to violate capacity, route duration or time window constraints. If the initial solution is feasible it would registered as the incumbent, that is, $S^* = S$ with cost $c(S^*) = c(S)$. If it is infeasible then the incumbent is the solution with an infinite cost, that is, $c(S^*) = \infty$, implying that the first feasible solution produced by the tabu iterations will become the first incumbent.

When evaluating the objective function value, possible improvements include using self-adjusting parameters α, β and γ so as to allow exploration of intermediate infeasible solutions. The parameters would initially be set to some value, for example 10 and at each iteration, if the previous solution is infeasible with regard to any of the time window, duration or capacity constraints, the appropriate parameter is multiplied by 2, whereas if the previous solution is feasible, the appropriate parameter is divided by 2. Parameter values could also be weighted differently as opposed to using the same value for each.

With regard to improving the neighbourhood it could be considered using a more powerful neighbourhood structure, for example one which entails exchanging up to two customers

between two routes, so as to overcome the issue of cycling. The λ-interchanges include simple swaps between two routes or simple moves taking one customer from the first route and inserting it into another. Otherwise, the current neighbourhood structure could also be altered by only considering shifts between vehicles and then applying a re-optimisation TSP algorithm to each route individually, which would accommodate shifts within vehicles. Also, when shifting customers, it could be considered only inserting them in positions where the lower bound time window of the shifted customer is lower than the customer that is to succeed it. Finally, a variable tabu tenure θ could be used, where θ is randomly drawn from the interval $[5,10]$. Also, with regard to improving the tabu mechanism, it could be considered designing an algorithm that allows the tabu list to be empty at the outset.

With regard to improving the stopping criterion, a more sophisticated stopping criterion could be used instead of simply having a maximum number of iterations. For example, a stopping criterion of a maximum number of iterations without improvement could be employed. This could also allow an intensification technique to be introduced by conducting a more intense search by using the incumbent as the starting point. If this were to be done however, a second stopping criterion would then be necessary, where this value would be lower than the first. It would not be easy to design a stopping criterion which could detect the occurrence of cycling, however, using a stopping criterion which eliminates unnecessary iterations could reduce computation time.

Furthermore, a post-optimisation procedure could be applied to each route of the best solution so far using or-opt. This method would involve removing strings of three, two or one consecutive customers and reinserting them either in the same route or another route. Or-opt exchanges are actually a subset of 3-opt exchanges and perform well for problems involving time windows.

Finally the overall design of the algorithm could be improved by finding a less involved way of representing a single solution and reducing the number of components necessary for calculating the objective function. The overall efficiency of the code could also be improved so as to reduce computation time and explore a greater portion of the solution space in a smaller amount of time. Finally, a user interface could be developed to display all desired information such as graphically displaying the routes.

REFERENCES

Braysy, O. & Gendreau, M., 2005. Vehicle routing problem with time windows part I: route construction and local search algorithms. *Transportation Science*, 39(1), pp.104–118.

Bräysy, O. & Gendreau, M., 2002. Tabu Search heuristics for the Vehicle Routing Problem with Time Windows. *Top*, 10(2), pp.211–237.

Bräysy, O. & Gendreau, M., 2005. Vehicle routing problem with time windows, Part II: Metaheuristics. *Transportation science*, 39(1), pp.119–139. Available at: http://pubsonline.informs.org/doi/abs/10.1287/trsc.1030.0057.

Cordeau, J.-F. et al., 2002. A guide to vehicle routing heuristics. *Journal of the Operational Research Society*, 53(5), pp.512–522.

Cordeau, J.F., Laporte, G. & Mercier, a, 2001. A unified tabu search heuristic for vehicle routing problems with time windows. *The Journal of the Operational Research Society*, 52(8), pp.928–936.

Desrochers, M., 1992. A new optimization algorithm for the vehicle routing problem with time windows. *Operations Research*, 40(2), pp.342–354. Available at: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:No+Title#0\nhttp://pubsonline.informs.org/doi/abs/10.1287/opre.40.2.342.

Gendreau, M., Hertz, A. & Laporte, G., 2010. Tabu Search Heuristic for the Vehicle Routing Problem. , 40(10), pp.1276–1290.

Gendreau, M., Hertz, A. & Laporte, G., 1992. New insertion and postoptimization procedures for the traveling salesman problem. *Operations Research,* 40(6), pp.1086-1094.

Gendreau, M., Hertz, A., Laporte, G. & Stan, M., 1998. A generalized insertion heuristic for the traveling salesman problem with time windows. *Operations Research,* 43(3), pp.330-335.

Gendreau, M. & Potvin, J.-Y., 2005. Metaheuristics in combinatorial optimization. *Annals of Operations Research*, 140, pp.189–213.

Glover, F., 1993. A user ' s guide to tabu search *. , 41(20), pp.3–28.

Glover, F., 1990. Tabu Search: A Tutorial. , 1990(August), pp.74–94.

Laporte, G., 2010. A concise guide to the Traveling Salesman Problem. *Journal of the Operational Research Society*, 61(1), pp.35–40.

Laporte, G., 2009. Fifty Years of Vehicle Routing. *Transportation Science*, 43(4), pp.408–416.

Laporte, G. & Cordeau, J.-F., 2002. Tabu Search Heuristics for the Vehicle Routing Problem.

Laporte, G., Gendreau, M. & Potvin, J., 2000. Classical and modern heuristics for the vehicle routing problem. , 7.

Potvin, J.-Y. et al., 1996. The Vehicle Routing Problem with Time Windows Part I: Tabu Search. *INFORMS Journal on Computing*, 8(2), pp.158–164. Available at: http://joc.journal.informs.org/content/8/2/158.abstract.

Solomon, M.M., 2005. VRPTW Benchmark Problems. Available from: < http://w.cba.neu.edu/~msolomon/problems.htm>. [22 November 2015].

Solomon, M.M., 1987. Algorithms for the Vehicle Routing and Scheduling Problems With Time Window Constraints. *Operations Research*, 35(2), pp.254–265. Available at: http://proxy.lib.uiowa.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=bth&AN=15255268.

Taillard, É.D. et al., 2001. Adaptive memory programming: A unified view of metaheuristics. *European Journal of Operational Research*, 135(1), pp.1–16.

Toth, P. & Vigo, D., 2003. The Granular Tabu Search and Its Application to the Vehicle-Routing Problem. *INFORMS Journal on Computing*, 15(4), pp.333–346.

APPENDIX A – LEGEND FOR TABLE 1

C - A. Chabrier, “Vehicle Routing Problem with Elementary Shortest Path based Column Generation.” Forthcoming in: *Computers and Operations Research*(2005).

CR - W. Cook and J. L. Rich, "A parallel cutting plane algorithm for the vehicle routing problem with time windows," Working Paper, Computational and Applied Mathematics, Rice University, Houston, TX, 1999.

IV - S. Irnich and D. Villeneuve, “The shortest path problem with k-cycle elimination (k ≥ 3): Improving a branch-and-price algorithm for the VRPTW.”Forthcoming in: INFORMS Journal of Computing (2005).

KDMSS - N. Kohl, J. Desrosiers, O. B. G. Madsen, M. M. Solomon, and F. Soumis, "2-Path Cuts for the Vehicle Routing Problem with Time Windows,"*Transportation Science*, Vol. 33 (1), 101-116 (1999).

KLM - B. Kallehauge, J. Larsen, and O.B.G. Madsen. "Lagrangean duality and non-differentiable optimization applied on routing with time windows - experimental results." Internal report IMM-REP-2000-8, Department of Mathematical Modelling, Technical University of Denmark, Lyngby,Denmark, 2000.

L - J. Larsen. "Parallelization of the vehicle routing problem with time windows." Ph.D. Thesis IMM-PHD-1999-62, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, 1999.

APPENDIX B – TEXT REPRESENTATION OF THE AIMMS CODE

```
Model Main_BasicTabuSearchTrial {
    DeclarationSection Initial_Declaration {
        Parameter IterationLimit;
        Parameter NumberOfCustomers {
            Comment: "user controlled: Number of customers";
        }
        Parameter VehicleCapacity {
            Comment: "user controlled: vehicle capacity";
        }
        Parameter NumberOfVehicles {
            Comment: "User controlled: Number of vehicles.";
        }
        Parameter XCoord {
            IndexDomain: i;
        }
        Parameter YCoord {
            IndexDomain: i;
        }
        Parameter LBTW {
            IndexDomain: i;
        }
        Parameter UBTW {
            IndexDomain: i;
        }
        Parameter Demand {
            IndexDomain: i;
        }
        Parameter ServiceTime {
            IndexDomain: i;
        }
        Parameter Distance {
            IndexDomain: (i,j);
            Text: "Distance from customer i to j";
            Definition: sqrt( (XCoord(j) - XCoord(i))^2 + (YCoord(j) - YCoord(i))^2
);
        }
    }
    Section InitialSolution {
        Procedure BuildFirstTour {
            Body: {
                empty CustomerToVehicle, InitTourConnect, InitialNext, InitialPrev,
CusToVeh, Arrival, Departure, WaitingTime;
                empty RemainingCapacity, RemainingDuration, Customer, Next, NewNext,
OCustomer, index1, Vehicle;

                RemainingCustomers := Customers; !start with all customers
                RemainingVehicles := Vehicles; !start with all vehicles

                Depot:= First(Customers); !define depot (this won't change again)
                RemainingCustomers -= Depot; !remove depot from remaining customers

                Savings(i,j):=    Distance(i,Depot)    +    Distance(Depot,j)    -
Distance(i,j);
                Duration := UBTW(Depot); !define duration and upper bound of depot

                block !INITIAL TOUR
                while (Card(RemainingVehicles)<>0) do

                Vehicle := First(RemainingVehicles); !select first vehicle of what's
left
                RemainingVehicles -= Vehicle; !remove selected vehicle from set


                RemainingFeasibleCustomers:= RemainingCustomers; !customers updated
                Customer:=Depot; !reset customer to depot
```

```
                RemainingCapacity := VehicleCapacity; !reset vehicle capacity
constraint
                RemainingDuration:= Duration; !reset duration constraint


                block!FOR EVERY SELECTED VEHICLE
                CustomerToVehicle(Vehicle) += Depot; !add depot to selected vehicle
                index1:= First(Counter);!selects first precision and adds depot as
first stop
                CusToVeh(Vehicle,index1) := Val(Depot);
                Departure(Depot):= 0;

                endblock;

                  Block !ADDS FIRST CUSTOMER OF ROUTE
                  !Adds closest customer to depot as first element on selected route
                  !considers both Euclidean distance as well as urgency

                    InitialNext(Customer)    := ArgMin( c in RemainingCustomers,
Distance(Customer,c) + 10*UBTW(c) );
                    Customer                := InitialNext(Customer);
                    RemainingCustomers     -= Customer;
                    RemainingFeasibleCustomers:=    RemainingCustomers;    !feasible
customers updated
                    CustomerToVehicle(Vehicle) += Customer; !adds customer to route

                    index1+= val(first(counter)); !first update index, first customer
will always be at second stop
                    CusToVeh(Vehicle, index1) := Val(Customer); !add customer to
first stop of vehicle

                    RemainingCapacity -= Demand(Customer); !reduces capacity

                !updates arrival of first stop
                    Arrival(Customer) := floor(Departure(Depot) + Distance(Depot,
Customer));
                !updates waitingtime of first stop
                    WaitingTime(Customer)   :=   floor(   Max(0,   LBTW(Customer)   -
Arrival(Customer) ));
                !reduce duration
                    RemainingDuration    -=    floor(Distance(Depot,    Customer)    +
WaitingTime(Customer) + ServiceTime(Customer));
                !update departure of first stop
                    Departure(Customer) := Arrival(Customer) + WaitingTime(CUstomer)
+ ServiceTime(Customer);

                    EndBlock;

                block !COMPLETES REST OF ROUTE

                   while ( Card( RemainingCustomers ) <> 0 ) do

                block!SELECTS POTENTIAL CUSTOMER AND CHECKS CONSTRAINTS
                   if card(RemainingCustomers) = 0 then break; endif;

                   !next customer should have biggest saving and be most urgent in
terms of tw
                InitialNext(Customer)    :=    ArgMax(  c  in  RemainingCustomers,
Savings(Customer,c) - 10*UBTW(c) );

                    Next := InitialNext(Customer);!don't update customer just yet
                     !first consider capacity
                if (Demand(Next) > RemainingCapacity) then
                RemainingFeasibleCustomers := RemainingCustomers;
                RemainingFeasibleCustomers -= Next; !updates RFC for next loop
                break; endif; !feeds break back to while loop
                !when you break then none of this is carried out
```

```
                !then consider duration
                Arrival(Next)  :=  floor(Departure(Customer)  +  Distance(Customer,
Next));
                    WaitingTime(Next) := floor( Max(0, LBTW(Next) - Arrival(Next) ));
                !must be able to go to next customer and back to the depot in time
                DistanceNeeded  :=  Distance(Customer,Next)  +  WaitingTime(Next)  +
ServiceTime(Next) + Distance(Next,Depot);
                    if (DistanceNeeded > RemainingDuration) then
                   RemainingFeasibleCustomers := RemainingCustomers;
                   RemainingFeasibleCustomers -= Next; !updates RFC for next loop
                break; endif; !feeds break back to while loop
                !when you break then none of this is carried out

                !then consider time window
                 if (Arrival(next) > ubtw(next)) then
                  RemainingFeasibleCustomers := RemainingCustomers;
                   RemainingFeasibleCustomers -= Next; !updates RFC for next loop
                break; endif;


                endblock;

                block! IF NO CONSTRAINT VIOLATED
                RemainingCustomers -= Next;
                RemainingFeasibleCustomers:= RemainingCustomers;

                OCustomer:= Customer;
                Customer := Next;

                  Arrival(Customer)       :=       floor(Departure(OCustomer)       +
Distance(OCustomer, Customer));
                  WaitingTime(Customer)   :=   floor(   Max(0,   LBTW(Customer)   -
Arrival(Customer) ));
                  Departure(Customer) := Arrival(Customer) + WaitingTime(Customer) +
ServiceTime(Customer);

                index1+= val(first(counter)); !adds 1 to index
                CusToVeh(Vehicle, index1) := Val(Customer);
                CustomerToVehicle(Vehicle) += Customer; !add next customer to route

                RemainingCapacity -= Demand(Customer);
                RemainingDuration   -=   floor(Distance(OCustomer,   Customer)   +
WaitingTime(Customer) + ServiceTime(Customer));
                endblock;

                endwhile;

                block !IF A CONSTRAINT WAS VIOLATED
                !use this to search remaining feasible customers if a constraint was
violated
                while ( Card(RemainingFeasibleCustomers) <> 0) do
                empty NotFeasible;

                !if card(RemainingCustomers) = 0 then break; endif;

                InitialNext(Customer)  := ArgMax ( c in RemainingFeasibleCustomers ,
Savings(Customer,c) - UBTW(c)  );

                NewNext := InitialNext(Customer); !selects potential customer

                 Arrival (NewNext) := floor(Departure(Customer) + Distance(Customer,
NewNext));
                 WaitingTime(NewNext)    :=    floor(    Max(0,    LBTW(NewNext)    -
Arrival(NewNext) ));

                DistanceNeeded := Distance(Customer,NewNext) + WaitingTime(NewNext)
+ ServiceTime(NewNext) + Distance(NewNext,Depot);
```

```
                if (DistanceNeeded > RemainingDuration) then NotFeasible += 1;
endif;

                 if (Demand(NewNext) > RemainingCapacity) then  NotFeasible += 1;
endif;

                 if (arrival(NewNext) > ubtw(NewNext)) then NotFeasible += 1; endif;

                 if (NotFeasible >= 1) then RemainingFeasibleCustomers -= NewNext;

               !basically a new route is started if constraints are violated again
               else

               OCustomer:= Customer;
               Customer := NewNext;

               CustomerToVehicle(Vehicle) += Customer;

               RemainingCapacity -= Demand(Customer);
               index1+= val(first(counter)); !adds 1 to index
               CusToVeh(Vehicle, index1) := Val(Customer);


                 Arrival(Customer)      :=      floor(Departure(OCustomer)      +
Distance(OCustomer, Customer));
                 WaitingTime(Customer)   :=   floor(   Max(0,   LBTW(Customer)   -
Arrival(Customer) ));
                 Departure(Customer) := Arrival(Customer) + WaitingTime(Customer) +
ServiceTime(Customer);

               RemainingDuration   -=   floor(Distance(OCustomer,   Customer)   +
WaitingTime(Customer) + ServiceTime(Customer));

               RemainingCustomers -= NewNext;
               RemainingFeasibleCustomers -= NewNext;
                endif;
               endwhile;
               endblock;

               endblock;

               block !CLOSES ROUTE

               !close the first route
               InitialNext(Customer) := Depot;
               InitialPrev(InitialNext(c))  := c;
               InitTourConnect(c,InitialNext(c)) := 1;

               index1+= val(first(counter)); !adds 1 to index
               CusToVeh(Vehicle, index1) := Val(Depot);

                if card(RemainingCustomers) = 0 then break; endif;
               endblock;

               endwhile;
               endblock;

               Arrival(Depot):=       floor(Departure(InitialPrev(Depot))       +
Distance(InitialPrev(depot), Depot));
            }
        }
        DeclarationSection First_Tour_Declaration {
            Parameter DistanceNeeded;
            Parameter NotFeasible;
            Set Vehicles {
                Index: v;
                Parameter: Vehicle, Route, Vehicle2;
                Definition: {
```

```
                    ElementRange (1,NumberOfVehicles);
                }
            }
            Set Customers {
                Index: i, j, c;
                Parameter:
Customer,NewNext,OCustomer,DCustomer,ODCustomer,Depot,Next,OutPrev,OutNext,InPrev,I
nNext;
                Definition: {
                    ElementRange( from: 0, to: NumberOfCustomers, fill: 0 );
                }
            }
            Set CustomerToVehicle {
                IndexDomain: v;
                SubsetOf: Customers;
                Comment: "each row represents a vehicle and the set of customers it
visits.";
            }
            Parameter InitTourConnect {
                IndexDomain: (i,j);
                Comment: "indicator matrix to show if a vehicle travels from vertex
i to j";
            }
            ElementParameter InitialNext {
                IndexDomain: c;
                Range: Customers;
                Comment: "shows which customer follows customer c in initial tour";
            }
            ElementParameter InitialPrev {
                IndexDomain: c;
                Range: Customers;
                Comment: "shows which customer preceeds customer c in initial tour";
            }
            Parameter CusToVeh {
                IndexDomain: (v,z);
                Comment: "matrix containing sequence of customers in routes";
            }
            Parameter Arrival {
                IndexDomain: i;
                Comment: "arrival of customer i in initial tour";
            }
            Parameter Departure {
                IndexDomain: i;
            }
            Parameter WaitingTime {
                IndexDomain: c;
                Comment: "waiting time of customer c in initial tour";
            }
            Parameter RemainingDuration;
            Parameter RemainingCapacity;
            Set RemainingVehicles {
                SubsetOf: Vehicles;
            }
            Set RemainingCustomers {
                SubsetOf: Customers;
            }
            Set RemainingFeasibleCustomers {
                SubsetOf: Customers;
            }
            Parameter Savings {
                IndexDomain: (i,j);
                Comment: "savings (in terms of Euclidean distance) obtained when
going from vertex i to j";
            }
            Parameter Duration {
                Comment: "maximum route duration allowed - upper bound of time window
for depot";
            }
```

```
            Set counter {
                Index: z;
                Parameter: index1,index2,index1p,index1n,index2p,index2n,removal;
                Definition: {
                    ElementRange( from: 1, to: 25, fill: 0 );
                }
            }
        }
        Procedure InitialObjectiveFunction {
            Body: {
                empty     InitialVehicleTravelTime,     InitialDurationFeasibility,
InitialTimeWindowFeasibility, InitialVehicleLoad;

                alpha:= 1; beta:= 1; gamma:= 1; !initialise parameters

                !Initial load and duration of each vehicle
                InitialTravelTime(i,j):=   WaitingTime(i)   +   ServiceTime(i)   +
Distance(i,j);

                for (v in Vehicles) do
                InitialVehicleDistanceOnly(v)     :=     sum     ((i,j)|     (i     in
CustomerToVehicle(v) and j in CustomerToVehicle(v)),
                distance(i,j)*InitTourConnect(i,j));
                endfor;

                for (v in Vehicles) do
                InitialVehicleTravelTime(v) := sum ((i,j)| (i in CustomerToVehicle(v)
and j in CustomerToVehicle(v)),
                InitialTravelTime(i,j)*InitTourConnect(i,j));
                InitialVehicleLoad(v):= sum(i|i in CustomerToVehicle(v) ,Demand(i));
                endfor;
                !initial duration, load and time window violations of each vehicle
                for (v in Vehicles) do
                InitialTimeWindowViolation(v) := sum(c| c in CustomerToVehicle(v),
max(0,(Arrival(c) - UBTW(c))));
                InitialCapacityViolation(v)  :=  max(0,  (InitialVehicleLoad(v)  -
VehicleCapacity));
                InitialDurationViolation(v) := max(0, (InitialVehicleTravelTime(v) -
UBTW(Depot)));
                endfor;

                InitialOFV     :=     sum(v,     InitialVehicleTravelTime(v))     +
alpha*sum(v,InitialCapacityViolation(v)) + beta *sum(v, InitialDurationViolation(v))
+ gamma *sum(v, InitialTimeWindowViolation(v));
                InitialObjFuncVal   :=   sum(v,   InitialVehicleDistanceOnly(v))   +
alpha*sum(v,InitialCapacityViolation(v)) + beta *sum(v, InitialDurationViolation(v))
+ gamma *sum(v, InitialTimeWindowViolation(v));

                SolnVehNum:= 0;
                for (v in vehicles | sum(z,CusToVeh(v,z)) <> 0) do
                SolnVehNum += 1;
                endfor;

                block !FEASIBLITY CHECKS
                for (v in Vehicles) do
                if     InitialVehicleTravelTime(v)     <=     UBTW(Depot)     then
InitialDurationFeasibility(v) := 1; endif;
                if           InitialVehicleTravelTime(v)    >    UBTW(Depot)    then
InitialDurationFeasibility(v) := 0; endif;
                if      InitialVehicleLoad(v)      <=      VehicleCapacity      then
InitialLoadFeasibility(v) := 1; endif;
                if      InitialVehicleLoad(v)      >      VehicleCapacity      then
InitialLoadFeasibility(v) := 0; endif;
                endfor;

                for (c in Customers) do
                if Arrival(c) <= UBTW(c) then  InitialTimeWindowFeasibility(c) := 1;
endif;
```

```
                if   Arrival(c) > UBTW(c) then  InitialTimeWindowFeasibility(c) :=
0; endif;
                endfor;


                InitialTWPercentage  :=  ((sum(c,  InitialTimeWindowFeasibility(c))-
1)/NumberOfCustomers) * 100;


                if (InitialTWPercentage > 20) then
                InitialTWCheck := 1; else InitialTWCheck := 0; endif;
                if  (sum(v, initialLoadFeasibility(v)) = NumberOfVehicles) then
                initialLoadCheck := 1; else initialLoadCheck := 0; endif;
                if (sum(v, initialDurationFeasibility(v)) = NumberOfVehicles) then
                initialdurationcheck:= 1 ;else initialdurationcheck := 0; endif;

                initialFeasibility   :=   initialTWCheck   +   initialLoadCheck   +
initialdurationcheck;

                endblock;
            }
        }
        DeclarationSection Initial_Objective_Function_Declaration {
            Parameter beta;
            Parameter alpha;
            Parameter gamma;
            Parameter InitialTravelTime {
                IndexDomain: (i,j);
                Comment: "distance(i,j) + service(i)";
            }
            Parameter InitialVehicleTravelTime {
                IndexDomain: v;
                Comment: "Total distance travelled by each vehicle";
            }
            Parameter InitialVehicleDistanceOnly {
                IndexDomain: v;
            }
            Parameter InitialVehicleLoad {
                IndexDomain: v;
            }
            Parameter InitialTimeWindowViolation {
                IndexDomain: (v);
            }
            Parameter InitialCapacityViolation {
                IndexDomain: v;
            }
            Parameter InitialDurationViolation {
                IndexDomain: v;
            }
            Parameter InitialObjFuncVal;
            Parameter InitialOFV;
        }
        DeclarationSection Initial_Feasibility_Declaration {
            Parameter InitialDurationCheck;
            Parameter InitialFeasibility;
            Parameter InitialLoadCheck;
            Parameter InitialLoadFeasibility {
                IndexDomain: v;
            }
            Parameter InitialDurationFeasibility {
                IndexDomain: v;
            }
            Parameter InitialTimeWindowFeasibility {
                IndexDomain: c;
            }
            Parameter InitialTWCheck;
            Parameter InitialTWPercentage;
        }
```

```
    }
    Section TabuSearchProcess {
        Procedure neighbourhood {
            Body: {


                block !INITIALISE NEIGHBOURHOOD

                empty            initialSolnMatrix,            IncumbentSolnMatrix,
CurrentSolnMatrix,TrialOneSolnMatrix,TrialTwoSolnMatrix;
                empty                          IncumbentObjFuncVal,CurrentObjFuncVal,
TrialOneObjFuncVal,TrialTwoObjFuncVal;
                empty BestNext, BestPrev, CurrentNext, CurrentPrev, OldNext, OldPrev,
TrialTwoNext, TrialTwoPrev, TrialOneNext, TrialOnePrev;
                empty CurrentCtoV;
                empty       bestTWPercentage,       bestTWCheck       ,BestLoadCheck
,bestdurationcheck ;

                block!first fill in tabu list with all 1s status
                    for (t in TabuTenure) do
                    TabuList(t, para):= Val(first(Vehicles));
                    TabuList(t, para):= Val(first(Vehicles));
                    TabuList(t, para):=  Val(first(Vehicles));
                    TabuList(t, para):= Val(first(Vehicles));
                    endfor;

                    TabuListcopy(t, para) := TabuList(t, para);
                    endblock;

                !initialise parameters of tabu list (this will stay the same)
                V1:= element(TabuParameters,1);
                P1:= element(TabuParameters,2);
                V2:= element(TabuParameters,3);
                P2:= element(TabuParameters,4);

                RemainingCustomers := customers;
                RemainingCustomers -= depot;

                initialSolnMatrix(v,z) := CusToVeh(v,z);

                CurrentCtoV(v) := CustomerToVehicle(v); !create a copy
                CurrentSolnMatrix(v,z):= initialSolnMatrix(v,z); !set current soln
to intial

                IncumbentSolnMatrix(v,z) := initialSolnMatrix(v,z); !set incumbent
soln to initial

                TrialOneSolnMatrix(v,z) := initialSolnMatrix(v,z); !create a matrix
for for potential moves to be compared
                TrialTwoSolnMatrix(v,z) := initialSolnMatrix(v,z);

                IncumbentObjFuncVal := InitialObjFuncVal; !set incumbent to initial
soln obj fnc
                CurrentObjFuncVal := InitialObjFuncVal; !set current soln to intial
soln obj fnc

                BestNext(c):= InitialNext(c); BestPrev (c) := InitialPrev(c); !store
incumbent soln info
                CurrentNext(c):=  InitialNext(c);  CurrentPrev(c):=  InitialPrev(c);
!store current soln info
                OldPrev(c):= InitialPrev(c);OldNext(c):= InitialNext(c);!create copy
of old soln info

                TrialOnePrev(c) := OldPrev(c); TrialOneNext(c) := OldNext(c); !this
is where new info will go
                TrialTwoPrev(c) := OldPrev(c); TrialTwoNext(c) := OldNext(c); !this
is where new info will go
```

```
        !update !this happens for each move
        CurrentCtoV(v) := CustomerToVehicle(v);
        BestCtoV(v):= CurrentCtoV(v);
        TrialOneCtoV(v):= CurrentCtoV(v);
        TrialTwoCtoV(v):= CurrentCtoV(v);
        endblock;

            !and for feasibility checks
            bestVehicleTravelTime(v) := initialVehicleTravelTime(v) ;
            bestVehicleLoad(v) := initialVehicleLoad(v);
            bestTWPercentage := initialTWPercentage;
    }
}
DeclarationSection initialise_neighbourhood_declaration {
    Parameter initialSolnMatrix {
        IndexDomain: (v,z);
        Comment: "copy of initial tour matrix";
    }
    Set CurrentCtoV {
        IndexDomain: v;
        SubsetOf: Customers;
    }
    Set BestCtoV {
        IndexDomain: v;
        SubsetOf: Customers;
    }
    Set TrialOneCtoV {
        IndexDomain: v;
        SubsetOf: Customers;
    }
    Set TrialTwoCtoV {
        IndexDomain: v;
        SubsetOf: Customers;
    }
    Parameter IncumbentSolnMatrix {
        IndexDomain: (v,z);
        Comment: "best solution matrix";
    }
    Parameter CurrentSolnMatrix {
        IndexDomain: (v,z);
    }
    Parameter TrialOneSolnMatrix {
        IndexDomain: (v,z);
    }
    Parameter TrialTwoSolnMatrix {
        IndexDomain: (v,z);
    }
    Parameter IncumbentObjFuncVal;
    Parameter CurrentObjFuncVal;
    Parameter TrialOneObjFuncVal;
    Parameter TrialTwoObjFuncVal;
    ElementParameter BestNext {
        IndexDomain: c;
        Range: Customers;
    }
    ElementParameter BestPrev {
        IndexDomain: c;
        Range: Customers;
    }
    ElementParameter CurrentNext {
        IndexDomain: c;
        Range: Customers;
    }
    ElementParameter CurrentPrev {
        IndexDomain: c;
        Range: Customers;
    }
    ElementParameter OldNext {
```

```
                IndexDomain: c;
                Range: Customers;
            }
            ElementParameter OldPrev {
                IndexDomain: c;
                Range: Customers;
            }
            ElementParameter TrialOneNext {
                IndexDomain: c;
                Range: Customers;
            }
            ElementParameter TrialOnePrev {
                IndexDomain: c;
                Range: Customers;
            }
            ElementParameter TrialTwoNext {
                IndexDomain: c;
                Range: Customers;
            }
            ElementParameter TrialTwoPrev {
                IndexDomain: c;
                Range: Customers;
            }
        }
        Procedure TabuSearchMetaheuristic {
            Body: {
                !TABU MECHANISM
                ! REPEAT IT UNTIL STOPPING CRITERION MET
                 !simple shifts within and between vehicles
                while (loopcount <= IterationLimit) do
                !always create a trialOne outside the big shift loop

                block !DO A SIMPLE INSERTION FOR TRIAL ONE
                vehicle:= element(vehicles,1); vehicle2:= element(vehicles,1);
                index1 :=  element(counter,2); index2 :=  element(counter,3);

                TrialoneSolnMatrix(v,z) := currentSolnMatrix(v,z);!trial1 must reset
to current for first shift

                customer  :=  element(RemainingCustomers,  currentSolnMatrix(Vehicle
,index1)); !record customer being removed
                OCustomer := element(RemainingCustomers, currentSolnMatrix(Vehicle2
,index2)); !record customer being 'replaced'

                    !ALWAYS REMOVE CUSTOMER BEING MOVED
                    TrialoneSolnMatrix(vehicle, index1) := 0;

                block!SHIFTS ALL CUSTOMERS UP DURING REMOVAL
                    empty trialonectov;
                    trialonectov(v):= currentctov(v);
                    removal := element(counter,card(currentctov(vehicle)));
                    for (z in counter | currentSolnMatrix(Vehicle ,z) <> 0) do
                    if val(z) > val(index1) then
                    TrialoneSolnMatrix(vehicle,               z-1)               :=
CurrentSolnMatrix(vehicle,z); endif;
                    endfor;
                    !removes duplicate at end of route
                    TrialoneSolnMatrix(vehicle, removal) := 0;
                endblock;

                block !INSERTION

                    TrialoneSolnMatrix(vehicle2,index2)  :=  0;  !make  space  for
insertion

                    for (z in counter | currentSolnMatrix(Vehicle2 ,z) <> 0) do
                    if val(z) >= val(index2) then
```

```
                    TrialoneSolnMatrix(vehicle2,              z+1)              :=
CurrentSolnMatrix(vehicle2,z);
                    endif;endfor;

                    TrialoneSolnMatrix(vehicle2,index2)  :=  val(customer);  !Insert
customer

                    endblock;

                block!UPDATE OF CtoV SETS

                    TrialoneCtoV(v):= CurrentCtoV(v);
                    !only change sets if vehicles are different
                    if val(vehicle) <> val(vehicle2) then
                    TrialoneCtoV(vehicle) -= customer;
                    TrialoneCtoV(vehicle2) += customer;
                    endif;
                endblock;

                block !UPDATE INDICES INFO
                                index1n:=  index1  +  val(first(counter));  !update:
outnext and index1 + 1
                                if CurrentNext(customer) = depot then
                                Outnext:= depot; else
                                OutNext       :=       element(remainingcustomers,
CurrentSolnMatrix(Vehicle,index1n)); endif;

                                index1p:=  index1  -  val(first(counter));  !update
outprev and index1 - 1
                                if val(index1p) = 1 then OutPrev:= depot;
                                else     OutPrev:=     element(remainingcustomers,
CurrentSolnMatrix(Vehicle,index1p)); endif;

                                index2n:=  index2  +  val(first(counter));  !update:
innext & index2 + 1
                                if currentnext(ocustomer) = depot then
                                InNext := depot; else
                                InNext       :=       element(remainingcustomers,
CurrentSolnMatrix(Vehicle2,index2n));endif;

                                index2p:=     index2     -     val(first(counter));
!update:inprev and index2 -1
                                if val(index2p) = 1 then inprev:= depot;
                                else    InPrev    :=    element(remainingcustomers,
CurrentSolnMatrix(Vehicle2,index2p)); endif;
                endblock;

                block!NEED UP UPDATE NEXT AND PREV INFO
                    !first let trial2 next and prev = current next and prev
                    TrialoneNext(c) := CurrentNext(c);
                    TrialonePrev(c) := Currentprev(c);

                    block!for every REMOVAL of a customer

                           !newnext(cOut-1) = oldnext(c)
                           TrialoneNext(OutPrev) := OldNext(customer); !next(Cout -
1) changes
                           !newprev(cOut+1) = oldprev(c)
                           TrialonePrev(OutNext):=  Oldprev(customer);  !prev(cOut  +
1) changes
                    endblock;

                    block!for every INSERTION of a customer
                           TrialoneNext(customer)   :=   element(RemainingCustomers,
trialoneSolnMatrix(Vehicle2 ,index2n));
                           TrialoneNext(element(RemainingCustomers,
trialoneSolnMatrix(Vehicle2 ,index2p))) := customer;!next(Cin - 1)
```

```
                        TrialonePrev(element(RemainingCustomers,
trialoneSolnMatrix(Vehicle2 ,index2n))):= customer;
                        TrialonePrev(customer):=      element(RemainingCustomers,
trialoneSolnMatrix(Vehicle2 ,index2p));
                    endblock;
                endblock;

                Block !EVALUATE OBJFNCVAL FOR TRIAL ONE
                    !this is what takes long, calculated for every shift

                    empty    TrialoneTourConnect,TrialoneArrival,TrialoneDeparture,
Trialonewaitingtime ;
                    empty TrialoneDeparture, TrialoneTravelTime, TrialoneTravelTime,
TrialOneTimeWindowViolation;
                    empty   TrialoneDurationViolation,   TrialoneCapacityViolation,
TrialoneVehicleLoad;

                    TrialoneTourConnect(c,TrialoneNext(c)) := 1;
                    TrialoneTourConnect(TrialonePrev(c),c) := 1;

                    TrialoneDeparture(Depot):= 0;

                    for (v in Vehicles| card(TrialoneCtoV(v)) <> 0) do
                    for (z in counter | trialoneSolnMatrix(v ,z) <> 0) do
                    Dcustomer  :=  element(remainingcustomers,  trialoneSolnMatrix(v
,z)) ;
                    TrialoneArrival(DCustomer)                                    :=
floor(TrialoneDeparture(TrialonePrev(Dcustomer)) + Distance(TrialonePrev(DCustomer),
DCustomer));
                    TrialoneWaitingTime(DCustomer) := floor( Max(0, LBTW(DCustomer)
- TrialoneArrival(DCustomer) ));
                    TrialoneDeparture(DCustomer)  :=  TrialoneArrival(DCustomer)  +
TrialoneWaitingTime(DCustomer) + ServiceTime(DCustomer);
                    endfor;endfor;
                    trialoneArrival(Depot):=
floor(trialoneDeparture(trialonePrev(Depot))     +     Distance(trialonePrev(depot),
Depot));

                    TrialoneTravelTime(i,j):=        TrialoneWaitingTime(i)        +
ServiceTime(i) + Distance(i,j);

                    for (v in Vehicles) do
                    TrialoneVehicleTravelTime(v) := sum ((i,j)| (i in TrialoneCtoV(v)
and j in TrialoneCtoV(v)),
                    Distance(i,j)*TrialoneTourConnect(i,j));
                    trialoneVehicleLoad(v):= sum(i|i in TrialoneCtoV(v) ,Demand(i));
                    endfor;

                    for (v in Vehicles) do
                    TrialOneTimeWindowViolation(v) := sum(c| c in TrialOneCtoV(v),
max(0,(TrialOneArrival(c) - UBTW(c))));
                    TrialOneCapacityViolation(v) := max(0, (TrialOneVehicleLoad(v) -
VehicleCapacity));
                    TrialOneDurationViolation(v)              :=              max(0,
(TrialOnevehicleTravelTime(v) - UBTW(Depot))); endfor;

                    empty TrialOneObjFuncVal;
                    TrialOneObjFuncVal  :=  sum(v,  TrialOneVehicleTravelTime(v)  +
(alpha*TrialOneCapacityViolation(v))
                    +   (beta   *   TrialOneDurationViolation(v))   +   (   gamma   *
TrialOneTimeWindowViolation(v)));
                    endblock;

                endblock;

                SolnVehNum:= 0;
                for (v in vehicles | sum(z,CurrentSolnMatrix(v,z)) <> 0) do
                SolnVehNum += 1;
```

```
                endfor;

                !use the big shift loop for trial 2, and let trial 1 be updated if
trial 2 is better
                !when neighbourhood exhausted, trial 1 becomes new current
                !use big loop to do all other shifts to compare trial 2 with trial 1
and select a move

                block !TRIAL TWO SHIFT
                !perform the shift and evaluate the obj function

                while(loopcount <= SolnVehNum ) do
                !while(loopcount <= 5) do
                Vehicle :=     element(vehicles, loopcount); !everytime the loop
increases, the v1 is set to that count

                    while(loopcount <= SolnVehNum) do
                    !while(loopcount <= 3) do
                    Vehicle2 :=   element(vehicles, loopcount); !everytime the loop
increases, the v2 is set to that count

                          while (loopcount <= (card(currentctov(vehicle))-1)) do
                          !while (loopcount <= 2) do
                          index1     :=     element(counter,     loopcount)     +
val(first(counter)); !everytime the loop increases, index1 is set to that count

                                  while (loopcount <= (card(currentctov(vehicle2))-
1)) do
                                  !while (loopcount <= 5) do
                                  index2    :=    element(counter,    loopcount)    +
val(first(counter));!everytime the loop increases, index2 is set to that count

                empty redundant;

                If (val(Vehicle) = val(vehicle2)) then redundant += 1; endif;
                if (val(index1) = val(index2)) then redundant +=1; endif;

                if (redundant < 2) then

                    TrialtwoSolnMatrix(v,z) := currentSolnMatrix(v,z);!trial2 must
reset to current for every shift

                    customer             :=             element(RemainingCustomers,
currentSolnMatrix(Vehicle ,index1)); !record customer being removed
                    OCustomer            :=             element(RemainingCustomers,
currentSolnMatrix(Vehicle2 ,index2)); !record customer being 'replaced'

                    !ALWAYS REMOVE CUSTOMER BEING MOVED
                    TrialTwoSolnMatrix(vehicle, index1) := 0;

                !happens regardless of vehicles or precisions
                    block!SHIFTS ALL CUSTOMERS UP DURING REMOVAL
                    empty trialtwoctov;
                    trialtwoctov(v):= currentctov(v);
                    removal := element(counter,card(currentctov(vehicle)));
                    for (z in counter | currentSolnMatrix(Vehicle ,z) <> 0) do
                    if val(z) > val(index1) then
                    TrialTwoSolnMatrix(vehicle,               z-1)               :=
CurrentSolnMatrix(vehicle,z); endif;
                    endfor;
                    !removes duplicate at end of route
                    TrialTwoSolnMatrix(vehicle, removal) := 0;
                    endblock;

                    block !SAME VEHICLE INSERTION

                          if val(vehicle) = val(vehicle2) then
```

```
                                    if val(index1) < val(index2) then
                                    block !P1<P2
                                     !make space for insertion
                                    TrialtwoSolnMatrix(vehicle2,index2) := 0;
                                    for (z in counter | currentSolnMatrix(Vehicle2 ,z)
<> 0) do
                                    if val(z) > val(index2) then
                                    TrialTwoSolnMatrix(vehicle2,         z)         :=
CurrentSolnMatrix(vehicle2,z); endif;
                                    endfor;
                                    endblock;
                                    elseif val(index1) > val(index2) then
                                    block !P1>P2
                                    TrialTwoSolnMatrix(vehicle2, index2) := 0;

                                          for       (z       in       counter       |
currentSolnMatrix(Vehicle2 ,z) <> 0) do
                                          if val(z) < val(index2) then
                                          TrialTwoSolnMatrix(vehicle,      z)      :=
CurrentSolnMatrix(vehicle,z);
                                          elseif val(z) > val(index1) then
                                          TrialTwoSolnMatrix(vehicle,      z)      :=
CurrentSolnMatrix(vehicle,z);
                                          elseif val(index2) <= val(z) < val(index1)
then
                                          TrialTwoSolnMatrix(vehicle,     z+1)     :=
CurrentSolnMatrix(vehicle,z);
                                          endif;
                                          endfor;

                                    endblock;
                                    endif;
                                    !INSERT CUSTOMER
                                    TrialtwoSolnMatrix(vehicle2,index2)             :=
val(customer);
                                    endif;
                                    endblock;

                        block !DIFFERENT VEHICLE INSERTION

                        if val(vehicle) <> val(vehicle2) then

                        TrialtwoSolnMatrix(vehicle2,index2)  :=  0;  !make  space  for
insertion

                        for (z in counter | currentSolnMatrix(Vehicle2 ,z) <> 0) do
                        if val(z) >= val(index2) then
                        TrialTwoSolnMatrix(vehicle2,              z+1)              :=
CurrentSolnMatrix(vehicle2,z);
                        endif;endfor;

                        TrialtwoSolnMatrix(vehicle2,index2)  :=  val(customer);  !Insert
customer

                        endif;

                        endblock;

                                    block!UPDATE OF CtoV SETS
                        TrialtwoCtoV(v):= CurrentCtoV(v);
                        !only change sets if vehicles are different
                        if val(vehicle) <> val(vehicle2) then
                        TrialtwoCtoV(vehicle) -= customer;
                        TrialtwoCtoV(vehicle2) += customer;
                        endif;
                        endblock;

                                    block !UPDATE INDICES INFO
```

```
                                    index1n:= index1 + val(first(counter)); !update:
outnext and index1 + 1
                                    if CurrentNext(customer) = depot then
                                    Outnext:= depot; else
                                    OutNext       :=       element(remainingcustomers,
CurrentSolnMatrix(Vehicle,index1n)); endif;

                                    index1p:= index1 - val(first(counter)); !update
outprev and index1 - 1
                                    if val(index1p) = 1 then OutPrev:= depot;
                                    else     OutPrev:=     element(remainingcustomers,
CurrentSolnMatrix(Vehicle,index1p)); endif;

                                    index2n:= index2 + val(first(counter)); !update:
innext & index2 + 1
                                    if currentnext(ocustomer) = depot then
                                    InNext := depot; else
                                    InNext       :=       element(remainingcustomers,
CurrentSolnMatrix(Vehicle2,index2n));endif;

                                    index2p:=     index2     -     val(first(counter));
!update:inprev and index2 -1
                                    if val(index2p) = 1 then inprev:= depot;
                                    else    InPrev    :=    element(remainingcustomers,
CurrentSolnMatrix(Vehicle2,index2p)); endif;
                                    endblock;

                     block!NEED UP UPDATE NEXT AND PREV INFO
                     !first let trial2 next and prev = current next and prev
                     TrialTwoNext(c) := CurrentNext(c);
                     TrialTwoPrev(c) := Currentprev(c);

                            block!for every REMOVAL of a customer

                            !newnext(cOut-1) = oldnext(c)
                            TrialTwoNext(OutPrev) := OldNext(customer); !next(Cout -
1) changes
                            !newprev(cOut+1) = oldprev(c)
                            TrialTwoPrev(OutNext):= Oldprev(customer); !prev(cOut +
1) changes
                            endblock;

                            block!for every INSERTION of a customer
                            TrialTwoNext(customer)   :=   element(RemainingCustomers,
trialtwoSolnMatrix(Vehicle2 ,index2n));
                            TrialTwoNext(element(RemainingCustomers,
trialtwoSolnMatrix(Vehicle2 ,index2p))) := customer;!next(Cin - 1)

                            TrialTwoPrev(element(RemainingCustomers,
trialtwoSolnMatrix(Vehicle2 ,index2n))):= customer;
                            TrialTwoPrev(customer):=      element(RemainingCustomers,
trialtwoSolnMatrix(Vehicle2 ,index2p));
                            endblock;
                     endblock;

                     Block !EVALUATE OBJFNCVAL FOR TRIAL TWO
                     !this is what takes long, calculated for every shift

                     empty     TrialTwoTourConnect,TrialTwoArrival,TrialTwoDeparture,
TrialTwowaitingtime ;
                     empty TrialTwoDeparture, TrialTwoTravelTime, TrialTwoTravelTime,
TrialTwoTWViolation;
                     empty    TrialTwoDurationViolation,    TrialTwoCapacityViolation,
TrialtwoVehicleLoad;

                     TrialTwoTourConnect(c,TrialTwoNext(c)) := 1;
                     TrialTwoTourConnect(TrialTwoPrev(c),c) := 1;
```

```
                    TrialTwoDeparture(Depot):= 0;


                    for (v in Vehicles| card(TrialtwoCtoV(v)) <> 0) do
                    for (z in counter | trialtwoSolnMatrix(v ,z) <> 0) do
                    Dcustomer  :=  element(remainingcustomers,  trialtwoSolnMatrix(v
,z)) ;
                    TrialtwoArrival(DCustomer)                                    :=
floor(TrialtwoDeparture(TrialtwoPrev(Dcustomer)) + Distance(TrialtwoPrev(DCustomer),
DCustomer));
                    TrialtwoWaitingTime(DCustomer) := floor( Max(0, LBTW(DCustomer)
- TrialtwoArrival(DCustomer) ));
                    TrialtwoDeparture(DCustomer)  :=  TrialtwoArrival(DCustomer)  +
TrialtwoWaitingTime(DCustomer) + ServiceTime(DCustomer);
                    endfor;endfor;
                    trialtwoArrival(Depot):=
floor(trialtwoDeparture(trialtwoPrev(Depot))     +     Distance(trialtwoPrev(depot),
Depot));

                    TrialTwoTravelTime(i,j):=        TrialTwoWaitingTime(i)        +
ServiceTime(i) + Distance(i,j);

                    for (v in Vehicles) do
                    TrialTwoVehicleTravelTime(v) := sum ((i,j)| (i in TrialTwoCtoV(v)
and j in TrialTwoCtoV(v)),
                    Distance(i,j)*TrialTwoTourConnect(i,j));
                    trialtwoVehicleLoad(v):= sum(i|i in TrialtwoCtoV(v) ,Demand(i));
                    endfor;

                    for (v in Vehicles) do
                    TrialTwoTWViolation(v)   :=   sum(c|   c   in   TrialtwoCtoV(v),
max(0,(TrialTwoArrival(c) - UBTW(c))));
                    TrialTwoCapacityViolation(v) := max(0, (TrialtwoVehicleLoad(v) -
VehicleCapacity));
                    TrialTwoDurationViolation(v)              :=              max(0,
(TrialTwoVehicleTravelTime(v) - UBTW(Depot))); endfor;

                    empty TrialtwoObjFuncVal;
                    TrialtwoObjFuncVal  :=  sum(v,  TrialTwoVehicleTravelTime(v)  +
(alpha*TrialTwoCapacityViolation(v))
                    +   (beta   *   TrialTwoDurationViolation(v))   +   (   gamma   *
TrialTwoTWViolation(v)));
                    endblock;

                                !then do comparisons of trial1 and trial2
                                !every time trial 2 < trial 1 it gets updated

                    if (TrialTwoObjFuncVal < TrialOneObjFuncVal) then

                    empty tabucheck, SecondTabuCheck, tabuclear;

                    for (t in TabuTenure) do
                    TabuStatusCheck(t, V1):= Val(Vehicle);
                    TabuStatusCheck(t, P1):= val(index1);
                    TabuStatusCheck(t, V2):=  val(Vehicle2);
                    TabuStatusCheck(t, p2):= val(index2);
                    endfor;

                    block!TABU STUFF
                    !do first tabu status check
                    for (t in TabuTenure) do

                    if (TabuStatusCheck(t, V1) = TabuList(t,V1)) then TabuCheck(t)
+=1; endif;
                    if (TabuStatusCheck(t, P1) = TabuList(t,P1)) then TabuCheck(t)
+=1; endif;
                    if (TabuStatusCheck(t, V2) = TabuList(t,V2)) then TabuCheck(t)
+=1; endif;
```

```
                    if (TabuStatusCheck(t, p2) = TabuList(t,p2)) then TabuCheck(t)
+=1; endif;

                    endfor;

                    !do second tabu status check
                    for (t in TabuTenure) do
                    if  TabuCheck(t) = 4 then SecondTabuCheck(t) := 0;
                    else SecondTabuCheck(t) := 1; endif;
                    endfor;
                    endblock;

                TabuClear := sum(t, secondTabuCheck(t));

                    if (TabuClear = 10) then

                    block !REPLACE TRIAL ONE WITH TRIAL TWO
                    !replace trial1 with trial2 in terms of matrix, ctov sets, prevs
and nexts, and obj func value
                    TrialOneNext(c) := TrialTwoNext(c);
                    TrialOnePrev(c) := TrialTwoNext(c);
                    TrialOneSolnMatrix(v,z) := TrialTwoSolnMatrix(v,z);

                    empty TrialOneCtoV;
                    TrialOneCtoV(v):= TrialTwoCtoV(v);

                    TrialoneObjFuncVal := TrialtwoObjFuncVal;;
                    !and for feasibility checks
                    TrialOneVehicleTravelTime(v) := TrialTwoVehicleTravelTime(v);
                    TrialOneVehicleLoad(v) := TrialtwoVehicleLoad(v);
                    TrialOneArrival(c):= TrialtwoArrival(c);
                    endblock;

                    block !STORE TABU INFO
                    !simply store forbidden reversal v1, v2, p1, and p2
                    !will add to tabu list separately

                    storedv2:= val(vehicle2);
                    storedp2:= val(index2);
                    storedv1:= val(vehicle);
                    storedp1:= val(index1);

                    endblock;

                    !check if it meets aspiration criteria
                    else
                           !trial 2 is tabu, so only replace trial1 if it meets the
aspiration criterion
                           if (TrialTwoObjFuncVal < IncumbentObjFuncVal) then

                    block !REPLACE TRIAL ONE WITH TRIAL TWO
                    !replace trial1 with trial2 in terms of matrix, ctov sets, prevs
and nexts, and obj func value
                    TrialOneNext(c) := TrialTwoNext(c);
                    TrialOnePrev(c) := TrialTwoNext(c);
                    TrialOneSolnMatrix(v,z) := TrialTwoSolnMatrix(v,z);

                    empty TrialOneCtoV;
                    TrialOneCtoV(v):= TrialTwoCtoV(v);

                    TrialoneObjFuncVal := TrialtwoObjFuncVal;;
                    !and for feasibility checks
                    TrialOneVehicleTravelTime(v) := TrialTwoVehicleTravelTime(v);
                    TrialOneVehicleLoad(v) := TrialtwoVehicleLoad(v);
                    TrialOneArrival(c):= TrialtwoArrival(c);
                    endblock;

                    block !STORE TABU INFO
```

```
                    !simply store forbidden reversal v1, v2, p1, and p2
                    !will add to tabu list separately

                    tabustoredv1:= val(vehicle);
                    tabustoredv2:= val(vehicle2);
                    tabustoredp1:= val(index1);
                    tabustoredp2:= val(index2);

                    endblock;
                           else skip;
                           endif;

                    endif;
                                 endif;
                                 endif;
                                 endwhile;
                           endwhile;
                    endwhile;
              endwhile;
              endblock;

              !WHEN LOOP ENDS IT MEANS THE NEIGHBOURHOOD HAS BEEN EXHAUSTED -> MOVE

              block !UPDATE TABU LIST AFTER EACH MOVE
                    !copy top 4 rows of TabuList to bottom 4 rows of TabuListCopy
                    tabuV1:= val(element(vehicles,storedv2));
                    tabuP1:=  val(element(counter,storedp2));
                    tabuv2:=  val(element(vehicles, storedv1));
                    tabup2:=  val(element(counter,storedp1));

                                 tabuCopy:= first(TabuTenure);
                                 for (t in tabutenure) do
                                 tabuCopy+= val(first(TabuTenure));
                                 TabuListCopy(tabucopy,para) := TabuList(t, para);
                                 endfor;

                                 !take newest reversal and add to first row of
TListCopy
                                 tabuCopy:= first(TabuTenure);

                                 TabuListCopy(tabuCopy, V1):= tabuV1;
                                 TabuListCopy(tabuCopy, P1):= tabuP1;
                                 TabuListCopy(tabuCopy, V2):=  tabuv2;
                                 TabuListCopy(tabuCopy, p2):= tabup2;

                                 !then let TList := TListCopy
                                 TabuList(t,para) := TabuListCopy(t, para);
              endblock;

              block !REPLACE CURRENT WITH TRIAL ONE
                    !replace trial1 with trial2 in terms of matrix, ctov sets, prevs
and nexts, and obj func value
                    currentnext(c) := TrialOneNext(c);
                    currentprev(c) := TrialOnePrev(c);

                    CurrentSolnMatrix(v,z) := TrialOneSolnMatrix(v,z) ;

                    empty CurrentCtoV;
                    CurrentCtoV(v) := TrialOneCtoV(v);
                    CurrentObjFuncVal:= TrialoneObjFuncVal;
                    !and for feasibility checks
                    currentVehicleTravelTime(v) := TrialOneVehicleTravelTime(v) ;
                    currentVehicleLoad(v) := TrialOneVehicleLoad(v);
                    currentArrival(c) :=TrialOneArrival(c);

                    !ALSO: update next and prev info for old- and trialtwo-
                    oldnext(c):= CurrentNext(c);
                    oldprev(c):= Currentprev(c);
```

```
                    TrialTwoNext(c) := oldnext(c);
                    TrialTwoPrev(c) := oldprev(c);
                    endblock;

                block !CURRENT FEASBILITY CHECK

                for (v in Vehicles) do
                if     currentVehicleTravelTime(v)     <=     UBTW(Depot)     then
currentDurationFeasibility(v) := 1; endif;
                if         currentVehicleTravelTime(v)     >     UBTW(Depot)     then
currentDurationFeasibility(v) := 0; endif;
                if     currentVehicleLoad(v)      <=      VehicleCapacity      then
currentLoadFeasibility(v) := 1; endif;
                if     currentVehicleLoad(v)      >      VehicleCapacity      then
currentLoadFeasibility(v) := 0; endif;
                endfor;

                for (c in Customers) do
                if currentarrival(c) <= UBTW(c) then  currenttimeWindowFeasibility(c)
:= 1; endif;
                if                  currentArrival(c)      >      UBTW(c)      then
currentTimeWindowFeasibility(c) := 0; endif;
                endfor;

                currentTWPercentage  :=  ((sum(c,  currentTimeWindowFeasibility(c))  -
1)/NumberOfCustomers) * 100;

                if (currentTWPercentage = InitialTWPercentage) then
                currentTWCheck := 1; else currentTWCheck := 0; endif;
                if  (sum(v, currentLoadFeasibility(v)) = NumberOfVehicles) then
                currentLoadCheck := 1; else currentLoadCheck := 0; endif;
                if (sum(v, currentDurationFeasibility(v)) = NumberOfVehicles) then
                currentdurationcheck:= 1 ;else currentdurationcheck := 0; endif;

                currentFeasibility   :=   CurrentTWCheck   +   currentLoadCheck   +
currentdurationcheck;
                !use this for penalty term later, if less than 60 percent of TWs
violated, add 2500 to costs
                endblock;

                !DO A FEASIBILITY CHECK
                ! can only update incumbent if feasible

                block  !REPLACE  INCUMBENT  IF  CURRENT  IS  BETTER  (!BUT  ONLY  IF
FEASIBILE!!!)
                if CurrentObjFuncVal < IncumbentObjFuncVal then
                    if CurrentFeasibility = 3 then

                    bestnext(c) := currentnext(c);
                    bestprev(c) := currentprev(c);

                    IncumbentSolnMatrix(v,z) := CurrentSolnMatrix(v,z)  ;

                    BestCtoV(v) := CurrentCtoV(v);
                    IncumbentObjFuncVal := CurrentObjFuncVal;

                    !and for feasibility checks
                    bestVehicleTravelTime(v) := currentVehicleTravelTime(v) ;
                    bestVehicleLoad(v) := currentVehicleLoad(v);
                    bestArrival(c) := currentArrival(c) ;

                    bestDurationFeasibility(v):= currentDurationFeasibility(v);
                    bestLoadFeasibility(v) := currentLoadFeasibility(v);
                    besttimeWindowFeasibility(c) := currenttimeWindowFeasibility(c);
                    bestTWPercentage := currentTWPercentage;
                    bestTWCheck := currentTWCheck;
                    BestLoadCheck := CurrentLoadCheck;
```

```
            bestdurationcheck := currentdurationcheck

        endif;
        endif;

        endblock;

        endwhile;
    }
}
DeclarationSection Trial_One_Swap_Declaration {
    Parameter SolnVehNum {
        Range: integer;
    }
    Parameter TrialOneTourConnect {
        IndexDomain: (i,j);
    }
    Parameter TrialOneDeparture {
        IndexDomain: c;
    }
    Parameter TrialOneArrival {
        IndexDomain: c;
    }
    Parameter TrialOneWaitingTime {
        IndexDomain: c;
    }
    Parameter TrialOneTravelTime {
        IndexDomain: (i,j);
    }
    Parameter TrialOneVehicleTravelTime {
        IndexDomain: v;
    }
    Parameter TrialOneVehicleLoad {
        IndexDomain: v;
    }
    Parameter TrialOneTimeWindowViolation {
        IndexDomain: v;
    }
    Parameter TrialOneCapacityViolation {
        IndexDomain: v;
    }
    Parameter TrialOneDurationViolation {
        IndexDomain: v;
    }
}
DeclarationSection Trial_One_Feasibility_Check {
    Parameter TrialOneTWCheck;
    Parameter TrialOneTWPercentage;
    Parameter TrialOneDurationFeasibility {
        IndexDomain: v;
    }
    Parameter TrialOneTimeWindowFeasibility {
        IndexDomain: c;
    }
    Parameter TrialOneLoadFeasibility {
        IndexDomain: v;
    }
    Parameter TrialOneLoadCheck;
    Parameter TrialOneDurationCheck;
    Parameter TrialOneFeasibility;
}
DeclarationSection Trial_Two_Declaration {
    Parameter TrialTwoTourConnect {
        IndexDomain: (i,j);
    }
    Parameter TrialTwoArrival {
        IndexDomain: c;
    }
```

```
    Parameter TrialTwoWaitingTime {
        IndexDomain: c;
    }
    Parameter TrialTwoDeparture {
        IndexDomain: c;
        Range: free;
    }
    Parameter TrialTwoTravelTime {
        IndexDomain: (i,j);
    }
    Parameter TrialTwoVehicleTravelTime {
        IndexDomain: v;
    }
    Parameter TrialTwoTWViolation {
        IndexDomain: v;
    }
    Parameter TrialTwoDurationViolation {
        IndexDomain: v;
    }
    Parameter TrialTwoCapacityViolation {
        IndexDomain: v;
    }
    Parameter TrialtwoVehicleLoad {
        IndexDomain: v;
    }
}
DeclarationSection Tabu_Mechanism_Declaration {
    Parameter storedV1;
    Parameter redundant;
    Parameter nontabu1;
    Parameter nontabu2;
    Parameter nontabu3;
    Parameter nontabu4;
    Parameter nontabu5;
    Parameter nontabu6;
    Parameter nontabu7;
    Parameter firstTabuclear;
    Parameter tabustoredv1;
    Parameter actualv1;
    Parameter actualv2;
    Parameter actualp1;
    Parameter actualp2;
    Parameter storedv2;
    Parameter tabustoredv2;
    Parameter storedp1;
    Parameter tabustoredp1;
    Parameter tabustoredp2;
    Parameter storedp2;
    Parameter TabuStatusCheck {
        IndexDomain: (t,para);
        Range: (0, inf);
    }
    Parameter SecondTabuCheck {
        IndexDomain: t;
        Range: binary;
    }
    Parameter fakebest;
    Parameter faketrialone;
    Parameter faketrialtwo;
    Parameter TabuClear;
    Parameter TabuCheck {
        IndexDomain: t;
        Range: integer;
    }
    Set TabuParameters {
        Index: para;
        Parameter: V1, V2, P1, P2;
        Definition: {
```

```
                    ElementRange( from: 1, to: 4, fill: 0 );
                }
            }
            Parameter TabuList {
                IndexDomain: (t,para);
                Range: (0, inf);
            }
            Parameter TabuListCopy {
                IndexDomain: (t,para);
                Range: (0, inf);
            }
            Parameter tabuV1;
            Parameter tabuV2;
            Parameter tabuP1;
            Parameter tabuP2;
            Set TabuTenure {
                Index: t;
                Parameter:
tabu,tabuCopy,tabuIndex1,tabuIndex2,tabuIndex3,tabuIndex4,tabuIndex5,tabuIndex6,Tab
uIndex7;
                Definition: {
                    ElementRange( from: 1, to: 10, fill: 0 );
                }
            }
        }
        DeclarationSection Current_Feasibility_Check {
            Parameter CurrentFeasibility;
            Parameter CurrentLoadCheck;
            Parameter CurrentDurationCheck;
            Parameter CurrentTWCheck;
            Parameter CurrentTWPercentage;
            Parameter CurrentDurationFeasibility {
                IndexDomain: v;
            }
            Parameter CurrentLoadFeasibility {
                IndexDomain: v;
            }
            Parameter CurrentTimeWindowFeasibility {
                IndexDomain: c;
            }
            Parameter CurrentVehicleTravelTime {
                IndexDomain: v;
            }
            Parameter CurrentVehicleLoad {
                IndexDomain: v;
            }
            Parameter CurrentArrival {
                IndexDomain: c;
            }
        }
        DeclarationSection Incumbent_Feasbility_Check {
            Parameter BestLoadCheck;
            Parameter BestDurationCheck;
            Parameter BestTWCheck;
            Parameter BestTwPercentage;
            Parameter BestDurationFeasibility {
                IndexDomain: v;
            }
            Parameter BestLoadFeasibility {
                IndexDomain: v;
            }
            Parameter BestTimeWindowFeasibility {
                IndexDomain: c;
            }
            Parameter BestVehicleTravelTime {
                IndexDomain: v;
            }
            Parameter BestVehicleLoad {
```

```
                IndexDomain: v;
            }
            Parameter BestArrival {
                IndexDomain: c;
            }
        }
    }
    Procedure MainInitialization {
        Body: {

            NumberOfCustomers := 25;
            VehicleCapacity := 200;
            NumberOfVehicles := 25;
            IterationLimit  := 20;
            read from file "myfirstdatafile.txt";
        }
    }
    Procedure MainExecution {
        Body: {
            BuildFirstTour;
            InitialObjectiveFunction;
            neighbourhood;
            TabuSearchMetaheuristic;
        }
    }
    Procedure MainTermination {
        Body: {
            return 1;
        }
    }
}
```

APPENDIX C – SOLUTIONS IN ROUTE FORMAT

Table 3: Solutions as routes for Solomon 25-customer dataset

Problem Set	Routes
C101	(0,5,3,7,8,10,11,9,6,4,2,1,0) (0,20,24,25,19,15,16,14,12,0) (0,12,17,18,23,22,21,0)
C102	(0,8,10,11,9,6,4,7,2,1,5,1,0) (0,20,24,25,23,22,21,0) (0,13,17,18,19,15,14,12,16,0)
C103	(0,13,17,8,10,11,9,6,0) (0,15,14,12,19,18,16,0) (0,2,1,3,5,4,7,23,22,21,20,25,24,0)
C104	(0,25,8,15,11,9,23,19,14,12,0) (0,4,16,18,17,0) (0,13,2,1,6,3,10,5,24,22,21,20,0)
C105	(0,5,20,25,8,10,11,9,6,4,2,1,0) (0,13,17,18,19,15,16,14,12,0) (0,24,3,7,23,22,21,0)
C106	(0,13,17,18,19,15,16,14,12,0) (0,20,24,25,23,22,21,0) (0,5,3,7,8,10,11,9,6,4,2,1,0)
C107	(0,20,24,25,8,10,11,9,6,4,2,1,0) (0,3,5,7,19,15,16,14,12,23,22,0) (0,13,17,18,21,0)
C108	(0,5,3,7,8,10,11,9,0) (0,17,18,13,19,15,16,12,14,0) (0,20,24,25,6,4,2,1,22,23,21,0)
C109	(0,20,24,25,5,8,10,11,9,6,4,2,1,0) (0,3,7,13,17,19,15,16,14,12,0) (0,18,23,22,21,0)
C201	(0,22,20,24,2,1,6,7,3,4,23,18,19,16,14,12,15,17,17,25,9,11,10,8,21,0) (0,5,0)
C202	(0,20,22,24,6,23,18,19,16,14,15,17,12,25,9,11,10,8,12,3,1,0) (0,4,7,2,5,21,0)
C203	(0,22,6,23,15,17,13,25,9,11,10,8,16,14,19,0) (0,12,18,4,3,1,7,2,5,24,21,20,0)
C204	(0,4,23,15,13,25,9,11,8,16,14,19,0) (0,12,18,17,3,1,7,2,10,5,24,22,21,20,6,0)
C205	(0,20,22,5,24,2,1,6,7,3,4,23,18,19,16,14,12,15,17,13,25,9,11,10,8,0)
C206	(0,24,5,22,2,1,20,6,7,3,4,23,18,19,16,14,12,15,17,13,25,11,9,10,8,0)
C207	(0,5,2,1,24,22,6,20,7,3,4,23,18,19,16,14,12,15,13,17,25,11,9,10,8,0)
C208	(0,5,2,1,24,22,6,20,7,3,4,23,18,19,16,14,12,15,13,17,25,11,9,10,8,0)
R101	(0,14,15,22,4,25,0) (0,5,16,6,17,0) (0,2,21,3,24,0) (0,12,9,20,1,0) (0,11,19,10,13,0) (0,23,0) (0,7,8,0) (0,18,0)
R102	(0,11,19,7,18,6,0) (0,21,4,25,24,12,3,9,10,1,20,0) (0,15,14,16,5,0)

	(0,23,22,2,0) (0,8,17,13,0)
R103	(0,11,19,7,18,8,17,16,14,5,0) (0,13,2,15,22,21,23,0) (0,6,0) (0,9,20,10,1,0) (0,4,25,24,12,3,0)
R104	(0,13,2,15,14,16,17,5,8,0) (0,23,22,21,4,0) (0,19,11,10,20,3,9,0) (0,25,24,12,1,7,18,6,0)
R105	(0,5,14,15,22,4,24,0) (0,2,21,23,25,0) (0,12,9,20,10,1,0) (0,11,19,7,17,0) (0,16,18,6,13,0) (0,8,0) (0,3,0)
R106	(0,14,16,5,17,0) (0,11,10,1,0) (0,23,22,4,25,0) (0,19,7,8,18,6,13,2,15,21,0) (0,20,9,3,24,12,0)
R107	(0,17,16,14,15,2,13,0) (0,11,19,0) (0,18,6,5,8,0) (0,23,22,21,4,25,24,3,1,12,0) (0,7,10,20,9,0)
R108	(0,16,14,15,2,21,22,23,4,25,0) (0,19,7,18,6,0) (0,11,10,1,0) (0,20,9,3,12,24,0) (0,8,17,5,13,0)
R109	(0,14,15,21,4,24,25,0) (0,11,19,7,18,5,16,8,17,13,0) (0,23,22,2,6,0) (0,12,3,9,20,10,1,0)
R110	(0,16,17,5,8,18,19,11,10,1,0) (0,6,13,2,21,12,3,9,20,0) (0,14,15,22,23,25,4,24,0)
R111	(0,11,19,7,10,0) (0,9,20,1,3,24,25,4,12,0) (0,8,18,5,6,13,0) (0,14,16,17,0) (0,23,22,21,15,2,0)
R112	(0,6,13,0) (0,14,16,5,17,8,18,7,19,11,10,0) (0,15,2,22,21,4,23,0) (0,9,10,1,3,12,24,25,0)
R201	(0,5,14,2,15,23,11,19,16,7,18,8,9,3,10,20,1,4,24,13,17,25,0) (0,21,12,22,6,0)
R202	(0,14,15,22,23,21,0) (0,16,17,5,8,18,6,2,13,7,19,11,10,3,12,4,25,24,9,20,1,0)
R203	(0,16,14,15,2,13,0) (0,17,5,6,4,25,24,3,12,21,23,22,0) (0,19,11,1,9,20,10,7,8,18,0)

R204	(0,17,16,14,15,2,21,22,23,4,25,24,12,3,9,20,1,10,11,19,7,18,8,5,13,0)
R205	(0,5,16,14,15,2,6,18,7,19,11,10,20,9,3,1,0) (0,8,17,13,4,25,24,12,21,23,22,0)
R206	(0,24,12,3,9,20,1,10,11,19,7,18,8,5,17,0) (0,16,14,15,22,23,25,4,21,2,6,13,0)
R207	(0,20,9,3,24,25,4,21,22,23,15,2,6,13,14,16,17,5,8,7,18,19,11,10,1,0)
R208	(0,17,16,14,15,2,21,4,22,23,25,0) (0,12,24,3,1,9,20,10,11,19,7,8,18,5,13,6,0)
R209	(0,15,2,5,21,12,16,19,7,14,18,11,23,22,8,6,20,10,1,24,25,4,17,13,0) (0,3,9,0)
R210	(0,11,19,7,8,0) (0,23,22,21,2,15,14,16,17,5,6,18,13,24,3,9,20,10,1,12,4,25,0)
R211	(0,19,11,7,18,5,17,8,0) 0,16,14,15,2,6,13,21,22,23,4,25,24,12,3,9,20,10,1,0)
R212	(0,19,11,7,18,5,17,8,0) (0,16,14,15,2,6,13,21,22,23,4,25,24,12,3,9,20,10,1,0)
RC101	(0,14,15,12,11,9,10,17,13,0) (0,5,2,6,7,8,3,4,1,0) (0,23,21,18,19,20,24,25,0) (0,16,22,0)
RC102	(0,14,15,16,11,9,13,10,12,17,0) (0,25,23,21,18,19,22,24,20,0) (0,8,6,2,4,5,3,1,7,0)
RC103	(0,13,11,10,9,0) (0,24,22,20,18,21,23,25,19,0) (0,8,6,2,4,5,3,1,7,0) (0,16,15,17,12,14,0)
RC104	(0,16,15,11,9,13,14,17,0) (0,8,4,5,1,3,7,6,2,0) (0,20,24,19,18,21,23,25,22,0) (0,12,10,0)
RC105	(0,12,14,15,16,17,13,0) (0,11,9,10,4,3,1,5,0) (0,23,22,19,18,21,20,24,25,0) (0,8,7,6,2,0)
RC106	(0,2,6,7,17,13,0) (0,8,5,3,1,4,0) (0,11,14,15,16,9,12,10,0) (0,21,19,23,18,22,20,25,24,0)
RC107	(0,12,14,17,16,15,13,11,10,9,0) (0,23,21,18,19,20,22,24,25,0) (0,8,7,6,2,5,3,1,4,0)
RC108	(0,18,19,21,23,25,22,24,20,0) (0,7,6,8,4,5,3,1,2,0) (0,13,9,11,12,14,15,16,17,10,0)
RC201	(0,14,5,2,15,11,23,21,29,16,7,8,6,9,10,20,24,13,17,25,0) (0,12,18,22,3,4,1,0)
RC202	(0,23,19,22,21,18,20,24,25,0) (0,14,16,9,12,11,15,17,13,10,4,5,3,1,2,6,8,7,0)
RC203	(0,15,13,17,16,14,12,11,9,10,25,21,23,18,19,20,22,24,0) (0,2,6,5,3,1,8,4,7,0)
RC204	(0,5,3,1,4,7,8,2,6,12,14,16,17,13,10,11,15,9,24,20,19,23,21,18,25,0)
RC205	(0,15,11,14,12,9,16,10,17,13,24,20,22,19,18,21,23,25,0)
RC206	(0,5,2,15,17,13,10,9,11,12,14,16,7,8,6,4,3,1,0) (0,18,20,22,19,23,21,25,24,0)