Heuristic Algorithms for Dynamic Capacitated Arc Routing

Wasin Padungwech

School of Mathematics
Cardiff University

A thesis submitted for the degree of

Doctor of Philosophy

March 2018

Summary

This thesis concerns the capacitated arc routing problem (CARP), which can be used as a model of various real-life scenarios such as rubbish collection, snow ploughing, and other situations where an emphasis is placed on providing a certain service along streets. The goal of the CARP is to find a minimum-cost set of routes such that (i) each route starts and ends at the depot, (ii) each task is serviced in one of the routes, and (iii) the total demand in each route does not exceed the capacity. Until recently, the study of the CARP is concentrated on its "static" version, that is, it is assumed that the problem remains unchanged after vehicles start their journeys. However, with today's communication technology, a route planner and drivers can communicate with each other in real time, hence the possibility of amending vehicle routes if deemed necessary or appropriate for changes that may occur in the problem. This motivates the study of a dynamic CARP. This thesis focusses on one type of change in the dynamic CARP, namely the appearance of new tasks.

To ensure that a service can be performed smoothly, the ability to update a solution quickly is often preferable to achieving optimality with an excessive amount of computational effort. For this reason, we opt to develop a dynamic CARP solver based on heuristic algorithms. An investigation is conducted to gain more insights about what makes an algorithm improve a solution quickly. Furthermore, factors in the dynamic CARP beyond a solution-seeking algorithm are investigated. This includes the frequency of updating the solution and the idea of instructing vehicles to wait for additional tasks at certain locations. Efforts are focussed on reducing the total distance at the end of the service while ensuring that the service completion time is not excessive.

Acknowledgement

The first persons who have extensively helped me in this research are my supervisors, Dr Jonathan Thompson and Dr Rhyd Lewis. Thank you for bearing with me when I was lost in a maze of thoughts so many times. Thank you for your support throughout this research, from the beginning when I had little idea of what it meant to do a research until this thesis eventually came into existence.

I would also like to express my gratitude to my examiners, Professor Emma Hart and Professor Owen Jones, who provided me with valuable comments that help improve the quality of this thesis. Thank you for thought-provoking discussions and for letting me leave the viva voce room with a smile.

My study in the UK might never have happened in the first place without a scholarship granted by the Institute for the Promotion of Teaching Science and Technology (IPST), and the Development and Promotion of Science and Technology Talents Project (DPST) in Thailand. I am deeply grateful for this rare opportunity. I would also like to thank staff members of the Office of Educational Affairs (OEA), UK, and the Office of Civil Service Commision (OCSC), Thailand, for their support throughout my study in the UK, from A-Levels to PhD.

Thank you my fellow PGR students in Cardiff School of Mathematics for knocking on my office door every now and then and letting me know that I was not the only one who went through many ups and downs of a PhD life. I am also thankful for staff members of Cardiff School of Mathematics and Cardiff University for their support during my time in Cardiff.

I was also fortunate enough to meet fellow Thai PhD students in Cardiff. Although we could not meet very often, it was always a pleasure to hear and share life stories (both academic-related and random topics) with all of you. I could feel nice positive energy every time we met, and that did give me the power to pull through.

The last paragraph is reserved for the most important people in my life: my mom, my dad, and my brother. Thank you for all advice, perspectives, and support, especially when I was overwhelmed by setbacks and obstacles. I could not imagine how I would have completed a PhD by myself. I am deeply grateful to have been in this family.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

There are a variety of tasks or services that need to be performed along streets, for example, street sweeping, snow ploughing, garbage collection, and electric meter reading. Naturally, it is preferable to perform a service in a way that is as efficient as possible with respect to an objective being considered, e.g. minimising total distance. Thus, sensible route planning is often needed, or at least desired. In addition to the existence of tasks on streets, there are some constraints in practice that have an effect on route planning. One common constraint is the *capacity constraint*, i.e. the amount of service that can be performed on each route is limited. This leads to a problem called the *Capacitated Arc Routing Problem* (CARP).

The CARP was originally introduced in the literature by Golden and Wong (1981). Since then, a wide variety of both exact and heuristic algorithms have been proposed for finding (near-)optimal solutions. However, relatively few existing works on the CARP take into account the fact that some information in the problem, such as the existence or the quantities of tasks on each street, may change while vehicles are travelling and performing their services. Adapting or updating routes as the problem changes would allow us to obtain routes that are more "suitable" for the current state of the problem than those that are planned originally. This, however, requires a route planner to know the state of the problem at any given time. Fortunately, the availability of global positioning systems allow a route planner to track the positions of vehicles. Furthermore, with today's communication technology, it is possible for a route planner to detect (or be informed about) changes that occur in the CARP and notify drivers of any changes in their routes in real time. This makes it worthwhile to find a way of updating routes to ensure that a service being considered can still be performed "efficiently" as the problem changes over time.

There are many types of changes that can occur in the CARP. Examples of those changes, given by Liu et al. (2014b), include vehicles breaking down, streets becoming inaccessible, new tasks appearing, existing tasks being cancelled, changes in amounts of tasks, and changes in costs of travelling due to congestion. Nevertheless, this thesis will be focussed on one particular type of change: the appearance of new tasks.

For clarity, Section 1.1 and Section 1.2 give a definition of the CARP, as well as related terminology and notations that will be used throughout the thesis. Section 1.3 describes the dynamic CARP in greater detail. Section 1.4 presents research aims and the structure of this thesis.

1.1 Definition of the CARP

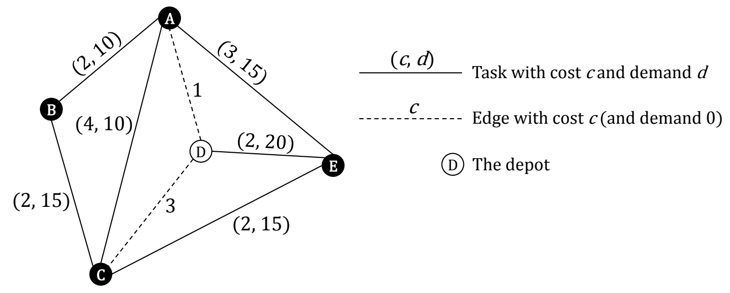
Let the following be given: a graph $G = (V, E)$ with a set of vertices V and a set of edges E; a positive cost (or distance[1]) c_{ij} and a non-negative demand d_{ij} for each edge $\{i, j\} \in E$; a capacity Q (assumed to be no less than any demand); and a depot $v_0 \in V$.

A path is an alternating sequence of vertices and edges $(v_1, e_1, v_2, \ldots, v_n, e_n, v_{n+1})$, where $v_1, \ldots, v_{n+1} \in V$ are vertices and $e_1, \ldots, e_n \in E$ are edges such that v_i and v_{i+1} are the endpoints of e_i for each $i = 1, \ldots, n$. A route is a path together with a binary sequence $(s_1, \ldots, s_n)$, i.e. each s_i is either 0 or 1. The edge e_i is said to be *serviced* in the route if the corresponding binary element $s_i = 1$. This is similar to a solution representation used by Brandão and Eglese (2008).

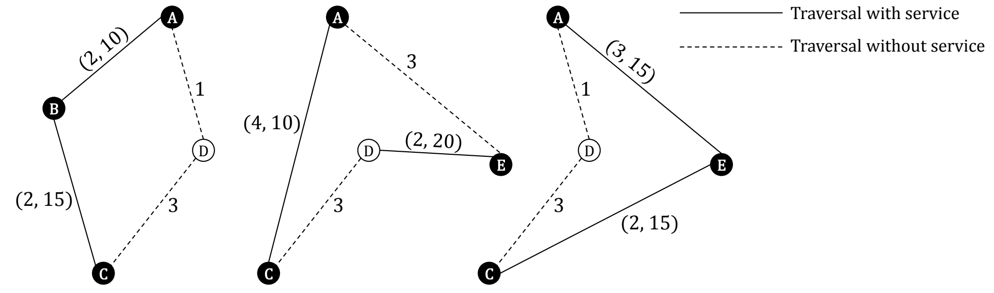
The objective of the CARP is to find a minimum-cost set of routes that satisfy the following conditions: (i) each edge with non-zero demand, also called a *task*[2], is serviced in one of the routes; (ii) the total demand of edges serviced in each route does not exceed the capacity; and (iii) each route starts and ends at the depot. A sample CARP instance and a feasible solution are shown in Figure 1.1. Note that in this thesis, we will focus on the CARP that is defined on an *undirected* graph: that is, each edge can be traversed in any direction, and the costs of travelling in both directions are the same.

[1]In this thesis, the terms "cost" and "distance" will be used interchangeably; they are both regarded as a numerical value that needs to be minimised.

[2]In some papers, this is called a *required edge*.



(a) CARP instance



(b) Feasible solution to the CARP instance in (a)

Figure 1.1: Sample CARP instance and feasible solution, where the given capacity is 30.

1.2 Notation

For each edge $\{i, j\}$, the two possible directions of traversal "from i to j" and "from j to i" are denoted by *arcs* (i, j) and (j, i), respectively. Let $h(a)$ and $t(a)$ denote the head and the tail of an arc a, respectively; for example, if $a = (i, j)$, then $h(a) = j$ and $t(a) = i$.

Instead of a path and a binary sequence, a route can be represented more concisely by a sequence $(v_s, a_1, \dots a_n, v_0)$, where v_s is the starting vertex, v_0 is the depot, and $a_1, \dots, a_n$ are tasks that are serviced in the route in the order they appear in the sequence (Beullens et al., 2003). Since the objective is to minimise the total cost, a path between consecutive tasks can be easily deduced: it is a shortest path between them, which can be found by an algorithm such as that of Dijkstra (1959). For example, the three routes in Figure 1.1(b) are (D, AB, BC, D), (D, DE, AC, D), and (D, CE, EA, D). In the standard CARP defined in Section 1.1, the starting vertex v_s of any route is the depot. However, later in this thesis, we will encounter routes that do not start at the depot (but still need to return to the depot), so we opt to include the starting vertex in the representation of routes.

Let $D(u, v)$ denote the shortest distance between vertices u and v. Let $c(a)$ and $d(a)$

denote the cost and the demand of an edge corresponding to an arc a, respectively. The cost of a route $R = (v_s, a_1, \dots a_n, v_0)$ is equal to

$$C(R) = \sum_{k=1}^{n} [D(h(a_{k-1}), t(a_k)) + c(a_k)] + D(h(a_n), v_0), \tag{1.1}$$

where, for ease of notation, $a_0 = (v_s, v_s)$. The total demand in the route R is equal to

$$D(R) = \sum_{k=1}^{n} d(a_k) \tag{1.2}$$

With this representation, the CARP can then be viewed as a problem of partitioning a set of tasks (each element of the partition corresponding to a route), permuting tasks in each route, and choosing the direction of service on each task to obtain a set of routes $S = \{R_1, \dots, R_K\}$ (for some positive integer K) such that $D(R_i)$ does not exceed a given capacity Q for all i, and the total cost

$$f(S) = \sum_{i=1}^{K} C(R_i) \tag{1.3}$$

is minimised.

1.3 Dynamic CARP

The CARP in which some information in the problem changes over time is also referred to as a *dynamic* CARP. In a dynamic CARP, a solution (i.e. a set of routes) needs to be updated in order to maintain the quality or feasibility of the solution in the face of changes in the problem. With new tasks appearing over time (the type of changes that will be focussed on in this thesis), a solution needs to be updated to ensure that all tasks are serviced, regardless of when they appear (as long as they appear within a time frame being considered).

Once it has been specified when the solution is to be updated, which can happen more than once, the dynamic CARP can be viewed as a sequence of static CARPs, where each static CARP occurs at one of the specified points in time. One possible approach to the dynamic CARP is to use an existing algorithm for the CARP to solve each static CARP in the sequence. However, it should be noted that obtaining an optimal solution to each static CARP in the sequence is not guaranteed to give the best possible solution to the underlying dynamic CARP. This is illustrated by a sample dynamic CARP instance in Figure 1.2: At time 0, there are 2 tasks, namely AD and BE, each with demand 1. The capacity is 3, so both tasks can be serviced in the same route. Vertex D is the depot. Each edge has distance 1 and the speed of the vehicle is 1 unit distance per unit time. Two feasible solutions at time 0 are shown: in Figure 1.2(top) is the route D–E–B–A–D with cost 4, while in Figure 1.2(bottom) is the route D–E–B–C–A–D with cost 5. In fact, the route in Figure 1.2(top) is the optimal solution for the static CARP instance at time 0. At time 3, task AC appears. In Figure 1.2(top), the vehicle has travelled D–E–B–A and serviced task BE, The best route from vertex A to the depot that services the remaining tasks AC and AD is A–C–A–D, so the whole route is D–E–B–A–C–A–D with cost 6. In Figure 1.2(bottom), at time 3 the vehicle has travelled D–E–B–C and also serviced tasks BE. The best route from vertex C to the depot the services the remaining tasks AC and AD is C–A–D, so the whole route is D–E–B–C–A–D with cost 5. Notice that the final solution in Figure 1.2(top) is obtained from an optimal solution at both time 0 and time 3, but it has higher total cost than the final solution in Figure 1.2(bottom). This is possible because features of the static CARP (e.g. the current position of the vehicle, the remaining tasks) at each update are affected not only by changes that occur in the problem, but also by solutions from previous updates. This highlights the importance of investigating and devising an algorithm specifically for the dynamic CARP, rather than solely relying on algorithms that are specifically designed for the static CARP.

1.4 Research Aims

The study of this thesis includes the following main points:

- To compare several variants of a tabu search algorithm for the static CARP and identify variants that can provide good solutions within limited time. Such variants will then be used to amend the solution in the dynamic CARP.
- To investigate the effect of different frequencies of updating the solution, and the way of integrating new tasks into an existing solution in each update on the quality of the final solution (i.e. the solution at the end of the planning horizon).
- To investigate the trade-off between the total distance travelled and resultant service completion time given by waiting strategies.

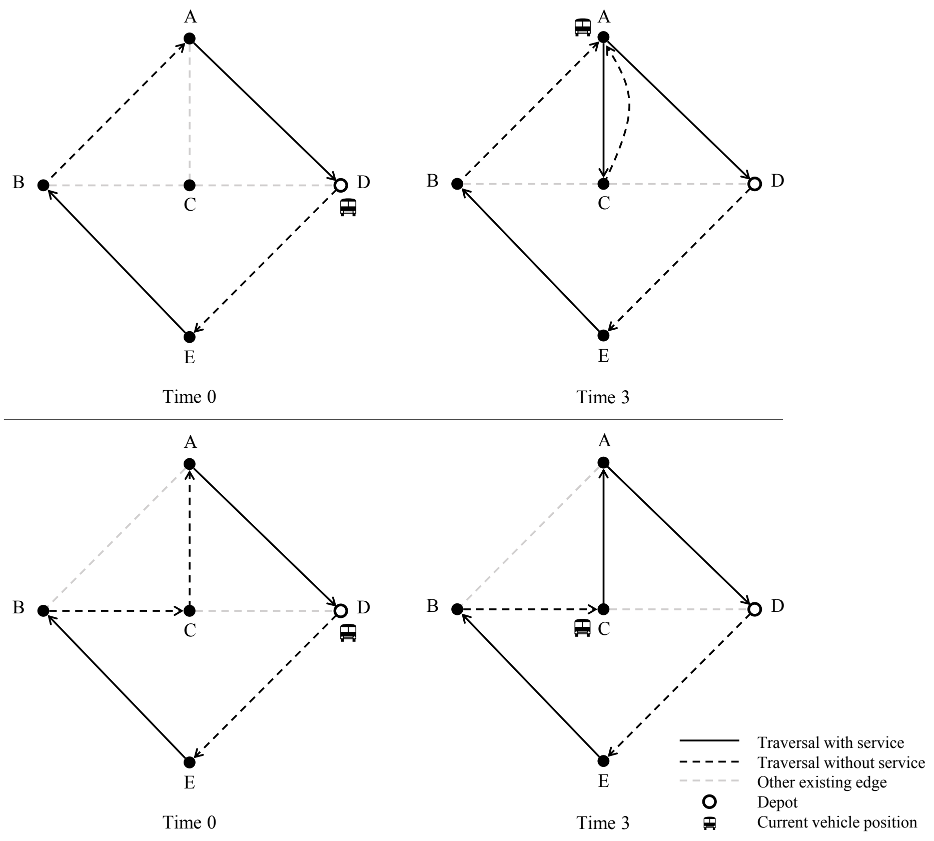


Figure 1.2: Sample dynamic CARP instance

1.5 Contributions of This Thesis

This research aims to better understand factors that help a heuristic algorithm for the dynamic CARP perform well, that is, being capable of amending a solution as new tasks appear over time while ensuring that both total distance (or solution cost) and service completion time do not increase excessively. Three major areas are considered: a heuristic algorithm for finding a solution in each update; the overall configuration of a dynamic CARP solver, concerning in particular an update frequency; and a way of integrating new tasks to an existing solution; and ways of anticipating changes in the future, including waiting strategies and adding extra routes to the solution even before they are needed. The contributions of this thesis are as follows.

- A novel analysis of the performance of tabu search for the static CARP with different ways of defining tabu moves is conducted. In particular, variants of tabu search are compared over a range of iteration limits as opposed to a single stopping criterion. This allows us to better understand what enhances or inhibits the algorithm's ability to improve a static CARP solution quickly.

- A novel operator, namely the deadheading cycle remover (DCR), is proposed. The aim of this operator is to further improve tabu search's speed of finding a good solution. The key idea of the DCR is to detect and remove traversals that are unnecessary (i.e not involving services) while maintaining feasibility of a solution.

- In this thesis, a tabu search algorithm is chosen as a heuristic algorithm for improving a solution in each update in the dynamic CARP. Experiment results show that increasing the maximum iteration limit for tabu search in each update does not consistently lead to significant improvement of the solution quality. This suggests the need to improve the dynamic CARP solver by other means instead of relying solely on running the algorithm for more iterations.

- Different frequencies of solution updates are empirically compared. Also, a novel concept of integrating new tasks to the solution while retaining an existing set of routes (instead of solving each update from scratch) is proposed and tested. Experiment results show that a promising update frequency depends on the degree of dynamism (i.e. the number of new tasks known after initial route planning in relation to the overall number of tasks) and the way of integrating new tasks to an existing set of routes.

- As a way to anticipate new tasks in the future, two waiting strategies are proposed and tested: waiting at the end of last task, and waiting away from other vehicles. These strategies aim to place vehicles at certain locations for them to stand by in case they receive additional tasks.

- A novel idea of adding extra empty routes to the solution before tabu search in each update is proposed and tested. This strategy aims to enhance flexibility of tabu search in amending the solution, especially when many routes are nearly full and thus there are relatively few feasible neighbourhood moves.

1.6 Academic Publication Produced

The following academic articles have been produced as a result of this research.

- Padungwech W., Thompson J., Lewis R. (2016). Investigating Edge-Reordering Procedures in a Tabu Search Algorithm for the Capacitated Arc Routing Problem. In: Blesa M. et al. (eds) *Hybrid Metaheuristics. HM 2016. Lecture*

Notes in Computer Science, vol 9668, pages 62-74. Springer, Cham. [conference paper]

- Padungwech W., Thompson J., Lewis R. A Dynamic Capacitated Arc Routing Problem with Arrival of New Demands, under review. [journal]

Chapter 2

Literature Review

2.1 A Brief Overview of Computational Complexity

One way to measure the intractability of a problem is to consider running time, i.e. the number of basic operations, that an algorithm requires to find or verify a solution. In computational complexity theory, such running time is used to classify problems into a variety of complexity classes, which helps shed some light on how hard a problem might be in comparison with others. In this section, several complexity classes are briefly reviewed to give sufficient background for a discussion on computational complexity of the CARP and related problems.

A common criterion for an algorithm to be efficient is that its running time is bounded above by a polynomial in terms of the size of the input. Such an algorithm is also called a *polynomial-time* algorithm. A problem is said to be in *P*, or *polynomial*, if there exists a polynomial-time algorithm for finding a solution to that problem.

For some problems, no polynomial-time algorithms for finding a solution have yet been found, nor has it been determined that such algorithms do not exist. However, for many of those problems, a proposed solution can be verified efficiently. A problem is said to be in *NP*, or *non-deterministic polynomial*, if there exists a polynomial-time algorithm for verifying any proposed solution. For example, the partition problem asks whether a multiset of positive integers can be partitioned into two subsets with the same sum. Any possible partition can be verified efficiently by calculating the sum in each subset; the number of additions in the calculation is polynomial – linear, in fact – in terms of the number of integers in a given set.

A problem is said to be *NP-hard* if all NP problems can be efficiently reduced (i.e. transformed by a polynomial-time algorithm) into that problem. This provides us with a rough idea of how (relatively) hard a problem can be: being able to efficiently solve a NP-hard problem means that we can also efficiently solve all NP problems. Notice that if one problem is known to be NP-hard and can be reduced in polynomial time to another problem, then the latter problem is also NP-hard. This is because a combination of polynomial-time algorithms is also polynomial-time.

2.2 Arc Routing Problems

As the name suggests, the CARP belongs to a class of arc routing problems: finding a route or a set of routes subject to constraints that are related to edges or arcs in a graph. Prior to the introduction of the CARP, other arc routing problems were studied. In fact, some of them can be viewed as special cases of the CARP. This section reviews those arc routing problems, which will allow us to see how arc routing problems have evolved and will also provide some useful techniques for constructing and improving solutions to the CARP.

2.2.1 Eulerian Graphs

One of the oldest arc routing problems is the "Königsberg bridges" problem. The city was divided by a river into 4 parts which were joined by 7 bridges as shown in Figure 2.1. The problem is to determine whether there exists a path in which each bridge is traversed precisely once. Euler (1741) proved[1] that it is not possible. In fact, Euler showed that for a general undirected graph, such a path does not exist if the graph has more than 2 odd-degree vertices (each part of the city is viewed as a vertex, and each bridge an edge; the degree of each vertex is the number of edges that meets the vertex). The converse result, i.e. it is possible to traverse each bridge precisely once if there exist no more than 2 odd-degree vertices, was proved by Hierholzer in 1873 (according to Biggs et al., 1976). The two results together form the following theorem. In what follows, a *closed* path is defined as a path whose first and last vertices are the same.

Theorem 2.1 *In an undirected (and connected) graph, there exists a path in which each edge is traversed precisely once if and only if at most 2 vertices have odd degrees.*

[1]The original proof was given in Latin. For the English translation, see e.g. Biggs et al. (1976).

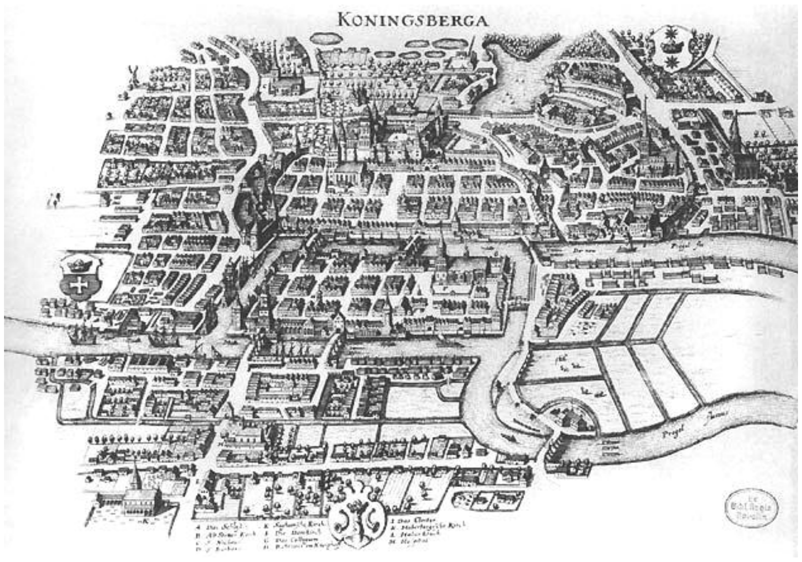


Figure 2.1: A map of Königsberg by Merian-Erben (retrieved 24 March 2017 from https://en.wikipedia.org/wiki/File:Image-Koenigsberg,_Map_by_Merian-Erben_1652.jpg)

Furthermore, there exists a closed path in which each edge is traversed precisely once if and only if all vertices have even degrees.

A path that contains each edge precisely once is called an *Eulerian* path. A graph that has an Eulerian closed path is called an *Eulerian* graph. Theorem 2.1 can then be stated in another way: an undirected and connected graph is Eulerian if and only if all vertices have even degrees. Note that Theorem 2.1 only applies to an undirected graph. For a directed and a mixed graph, Ford and Fulkerson (1962) provided the following theorems (Theorems 2.2 and 2.3) that specify necessary and sufficient conditions for a graph to be Eulerian. In a directed graph, the *in-degree* of a vertex v is defined as the number of arcs that go into v, i.e. having v as their heads. The *out-degree* of a vertex v is defined as the number of arcs that go out of v, i.e. having v as their tails.

Theorem 2.2 *A directed (and connected) graph is Eulerian if and only if the in-degree of any vertex is equal to its out-degree.*

For a mixed graph $G = (V, E \cup A)$ with the edge set E and the arc set A (an edge is undirected, whereas an arc is directed), some notations are needed: For any two subsets S_1, S_2 of the vertex set V, $N_d(S_1, S_2) = |\{(i, j) \in A | i \in S_1, j \in S_2\}|$ is the number of arcs from S_1 to S_2, and $N_u(S_1, S_2) = |\{\{i, j\} \in E | i \in S_1, j \in S_2\}|$ is the number of edges between S_1 and S_2.

Theorem 2.3 *A mixed (and connected) graph in Eulerian if and only if every vertex has even total degree (i.e. counting all edges and arcs that meet a given vertex, irrespective of directions of arcs), and for any subset $S \subset V$, the following inequality holds:*

$$|N_d(S, V\backslash S) - N_d(V\backslash S, S)| \leq N_u(S, V\backslash S).$$

With the above three theorems, the problem of determining whether a given graph is Eulerian is completely solved. The next question that naturally follows is how to traverse all edges in a given graph, Eulerian or not, with the least repetition. More generally, given a graph with edge costs, what is the minimum-cost tour (i.e. closed path) that covers all edges? This is the definition of the Chinese Postman Problem (CPP). If we are interested in a tour that covers a set of edges as opposed to all edges, the corresponding problem is called the Rural Postman Problem (RPP). These problems can be defined on any kind of graphs – undirected, directed, or mixed. Nevertheless, in the rest of this thesis, we will focus only on undirected graphs.

2.2.2 The Chinese Postman Problem

The CPP was introduced in the 1960s by a Chinese mathematician Meigu Guan, who “spent some time as a post office worker during the Chinese cultural revolution” (Eiselt et al., 1995). One application of the CPP is, as the name suggests, mail delivery: a solution to the CPP gives a minimum-cost route for a postman who needs to travel through each street in a given area, starting and ending his journey at the same location (e.g. the post office). If a given graph $G = (V, E)$ is Eulerian, the CPP is trivial as we only need to find an Eulerian tour. Otherwise, some edges need to be traversed more than once. In other words, it is necessary to duplicate some edges in G to obtain an Eulerian multigraph G'. Notice that the multiplicity of each edge in G' needs not be greater than 2 because if an edge is traversed more than twice, a cycle composed of two copies of the edge can be removed from G' without affecting the parity of the degrees of vertices, and thus G' is still Eulerian. This means that each edge in G needs to be duplicated at most once. Since the degree of each vertex v in G' must be even (according to Theorem 2.1), if the original degree of v in G is odd (even), the number of duplicated edges that are incident to v must be odd (even). Let V_{odd} be the set of odd-degree vertices in G, $\delta(i) = \{j \in V | \{i, j\} \in E\}$ the set of neighbours of vertex i, c_{ij} the cost of edge $\{i, j\} \in E$, and x_{ij} the number of times the edge $\{i, j\}$ is duplicated. The problem of finding a minimum-cost set of edges to be duplicated to form an Eulerian multigraph can be formulated as the following minimisation problem:

$$\text{Minimise} \sum_{\{i,j\}\in E} c_{ij}x_{ij} \tag{2.1}$$

subject to

$$\sum_{j\in\delta(i)} x_{ij} \equiv \begin{cases} 1 \pmod 2 \text{ if } i \in V_{\text{odd}} \\ 0 \pmod 2 \text{ otherwise} \end{cases} \tag{2.2}$$

$$x_{ij} \in \{0,1\} \text{ for } \{i,j\} \in E \tag{2.3}$$

Instead of directly solving the above integer linear program, a minimum-cost set of edges to be duplicated can be found by determining a minimum-cost set of shortest paths between odd-degree vertices (Edmonds and Johnson, 1973). Duplicating edges in those shortest paths makes all vertex degrees even, resulting in an Eulerian multigraph (by Theorem 2.1). To find a minimum-cost set of shortest paths, a complete graph G_{odd} is constructed from the set of all odd-degree vertices in a given graph G. The cost of each edge $\{i,j\}$ in G_{odd} is equal to the cost of a shortest path between i and j in G; these shortest paths can be found by a polynomial-time algorithm such as that of Dijkstra (1959). Once G_{odd} is constructed, what remains is to find a minimum-cost perfect matching in G_{odd}, which can be achieved in polynomial time by, for example, an algorithm proposed by Cook and Rohe (1999). Once the perfect matching is found, a shortest path in G corresponding to each edge in the matching is added to G to construct an Eulerian multigraph G' as required. This shows that the CPP can be solved in polynomial time.

Recall that the CPP concerns a minimum-cost route that contains all edges in a given graph. Interestingly, it is a much more difficult problem to find a minimum-cost route that contains only a subset of edges, as will be seen in the next section.

2.2.3 The Rural Postman Problem

The RPP is a problem of finding a minimum-cost route that covers a subset $R \subset E$ of edges in a given graph $G = (V, E)$; it is thus a more general version of the CPP. An edge that is in the specified subset R is also referred to as a *required* edge. Unlike the CPP, however, the RPP is NP-hard. One way to prove this concerns another problem, namely the Hamilton circuit problem, which is the problem of determining whether a given graph contains a closed path that visits each vertex precisely once. Lenstra and Kan (1976) showed that the Hamilton circuit problem, which was previously known to be NP-hard (Karp, 1972), can be reduced in polynomial time to the RPP.

As the RPP is NP-hard, several heuristics were proposed for this problem. Some heuristics (Christofides et al., 1981; Pearn and Wu, 1995) share the same idea: First, if required edges form several disjoint components, join them into one connected component by "shrinking" each component into a vertex and finding a minimum spanning tree in the resulting graph. Then, join odd-degree vertices by solving a minimum-cost perfect matching problem similar to the method of solving the CPP in Section 2.2.2. Pearn and Wu (1995) also proposed another heuristic which "reverses" the aforementioned heuristics, i.e. first join odd-degree vertices by shortest paths between them and then, if the resulting graph is not connected, connect the disjoint components by solving a minimum spanning tree problem.

For both the CPP and the RPP, the solution is a single route. In real life, however, more than one route (or vehicle) may be needed. For example, in the context of refuse collection, if the total amount of refuse is too large, one bin lorry would not be able to collect all the refuse in one go due to its limited capacity. Consequently, it needs to return to its depot to refill its capacity before it can collect more refuse or, if it is preferable to have multiple vehicles perform the service simultaneously, more lorries are needed. Such limitation gives rise to the Capacitated Arc Routing Problem (CARP). The relationship between the aforementioned arc routing problems is summarised in Figure 2.2.

2.3 Formulation of the CARP

The CARP can be viewed as an integer linear programming problem (ILP) (Golden and Wong, 1981): Let K denote the number of available vehicles (assumed to be sufficiently large to service all tasks), Q the vehicle capacity, and $|\cdot|$ the number of elements of a set. The objective of the CARP is to

$$\text{Minimise} \sum_{i \in V} \sum_{j \in V} \sum_{p=1}^{K} c_{ij} x_{ij}^{p} \tag{2.4}$$

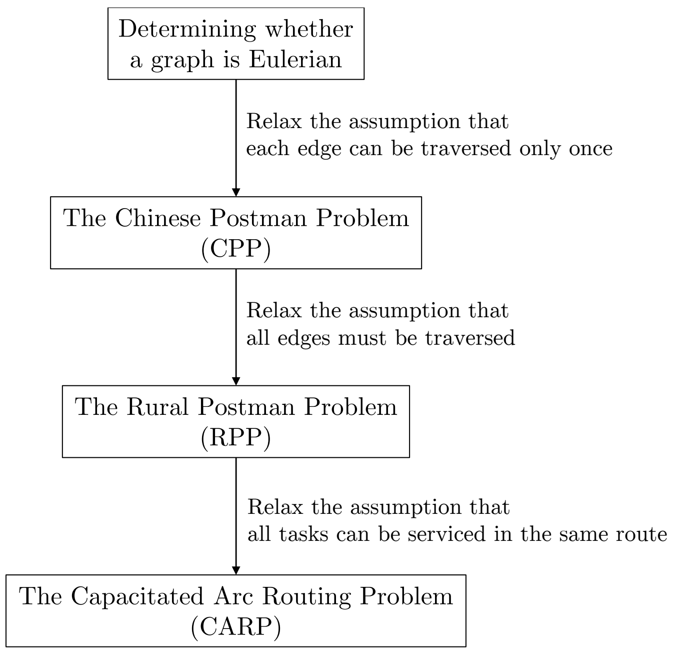

Determining whether a graph is Eulerian
Relax the assumption that each edge can be traversed only once
The Chinese Postman Problem (CPP)
Relax the assumption that all edges must be traversed
The Rural Postman Problem (RPP)
Relax the assumption that all tasks can be serviced in the same route
The Capacitated Arc Routing Problem (CARP)

Figure 2.2: Relationship between arc routing problems

subject to

$$\sum_{k\in V} x_{ki}^p - \sum_{k\in V} x_{ik}^p = 0 \qquad \text{for } i \in V \text{ and } p = 1,\dots,K, \tag{2.5}$$

$$\sum_{p=1}^{K} (l_{ij}^p + l_{ji}^p) = \left\lceil \frac{d_{ij}}{Q} \right\rceil \qquad \text{for } \{i,j\} \in E, \tag{2.6}$$

$$x_{ij}^p \geq l_{ij}^p \qquad \text{for } \{i,j\} \in E \text{ and } p = 1,\dots,K, \tag{2.7}$$

$$\sum_{i\in V}\sum_{j\in V} l_{ij}^p d_{ij} \leq Q \qquad \text{for } p = 1,\dots,K, \tag{2.8}$$

$$x_{ij}^p, l_{ij}^p \in \{0,1\} \qquad \text{for } \{i,j\} \in E \text{ and } p = 1,\dots,K, \tag{2.9}$$

$$\left.\begin{array}{ll} \sum_{i\in S}\sum_{j\in S} x_{ij}^p - |V|^2 y_S^p & \leq |S| - 1 \\ \sum_{i\in S}\sum_{j\notin S} x_{ij}^p + z_S^p & \geq 1 \\ y_S^p + z_S^p & \leq 1 \\ y_S^p, z_S^p & \in \{0,1\} \end{array}\right\} \begin{array}{l}\text{for } p = 1,\dots,K, \\ S \subset V\backslash\{v_0\}, S \neq \phi\end{array} \tag{2.10}$$

The decision variables in the above ILP formulation are

- x_{ij}^p – a decision variable that indicates whether a vehicle p traverses an edge $\{i,j\} \in E$ in the direction from i to j, i.e. $x_{ij}^p = 1$ if it does, $x_{ij}^p = 0$ otherwise (here we assume a vehicle traverses each edge in each direction at most once);

- l_{ij}^p – a decision variable that indicates whether a vehicle p services an edge $\{i,j\} \in E$ from i to j, i.e. $l_{ij}^p = 1$ if it does, $l_{ij}^p = 0$ otherwise (note that a vehicle either services a whole task or does not service it at all, i.e. partial service is not allowed);
- y_S^p, z_S^p – dummy variables for eliminating an illegal route, i.e. it is composed of disjoint cycles.

Constraint (2.5) ensures route continuity. Constraint (2.6) ensures that each task is serviced exactly once. Constraint (2.7) ensures that a vehicle p traverses any edge it services. Constraints (2.8) and (2.9) are the capacity and the integral (binary) constraints, respectively. Constraint (2.10) eliminates illegal routes. To see how Constraint (2.10) works, suppose that there exists a route (say, with index $\tilde{p}$) containing a cycle that does not involve the depot. We will show that this cycle is linked by some edge to the remaining part of the route. Let $\tilde{S}$ be the set of vertices in that cycle. It follows that

$$\sum_{i \in \tilde{S}} \sum_{j \in \tilde{S}} x_{ij}^{\tilde{p}} = |\tilde{S}| > |\tilde{S}| - 1. \tag{2.11}$$

Consequently, the first inequality and the binary constraint in (2.10) imply that $y_{\tilde{S}}^{\tilde{p}} = 1$, which in turn implies that $z_{\tilde{S}}^{\tilde{p}} = 0$ due to the third inequality in (2.10). It follows from the second inequality in (2.10) that

$$\sum_{i \in \tilde{S}} \sum_{j \notin \tilde{S}} x_{ij}^{\tilde{p}} \geq 1, \tag{2.12}$$

which means that the cycle is linked to a vertex outside that cycle, and therefore it is not a route itself.

2.4 Computational Complexity of the CARP

The CARP has been proved to be NP-hard. In fact, this can be proved in two different ways. The first proof, given by Golden and Wong (1981), involves reducing the partition problem to the 0.5-approximate[2] Capacitated Chinese Postman Problem (CCPP); as we have seen, the CCPP is a special case of the CARP in which all edges have positive demands. Since the partition problem is NP-hard (Karp, 1972),

[2]Given an optimisation problem and a positive real number α, the corresponding α-approximate problem is to find a solution whose cost is less than or equal to $(1+\alpha)$ times the optimal cost.

the existence of a polynomial-time reduction from the partition problem to the 0.5-approximate CCPP implies that the latter problem is NP-hard, and so also is the CARP.

The second proof relies on the fact that the RPP is NP-hard (Lenstra and Kan, 1976). Recall that the RPP can be viewed as a special case of the CARP in which the capacity is large enough for all demands to be serviced in a single route. Since one of its special cases is NP-hard, it follows that the CARP is also NP-hard. It should be noted that some special cases of the RPP can be solved in polynomial time: if all tasks (i.e. edges with positive demands) form a connected subgraph, the RPP becomes the CPP, which can be solved in polynomial time, as seen in Section 2.2.2.

Notice that these two proofs point to different factors that make the CARP intractable: the first relies on the existence of vehicle capacity, whereas the second concerns disconnectedness of tasks. Notice that these two factors do not need to appear together to make the CARP intractable; either of them alone can give rise to a NP-hard problem.

The fact that the CARP is NP-hard suggests that finding an optimal solution to the CARP, especially with a "large" size (with respect to the number of tasks), can take a substantial amount of time. In practice, it is usually more important to successfully operate a given service, in which case obtaining a sub-optimal solution within reasonable time would be preferable to spending much longer time finding an optimal solution. This can be achieved by means of (meta)heuristic algorithms.

2.5 Constructive Heuristics

In the early stage of research on the CARP, the problem was mostly solved by constructive heuristics. These heuristics are more recently used as generators of initial solutions for more sophisticated algorithms, including metaheuristic algorithms, which will be reviewed in later sections. Constructive heuristics that have been proposed for the CARP include the work of Golden and Wong (1981), Benavent et al. (1990), Pearn (1991), and Eydi and Javazi (2012). Among the most commonly used constructive heuristics are the Path-Scanning algorithm and Ulusoy's partitioning.

Path Scanning Proposed by Golden et al. (1983), the Path Scanning algorithm constructs routes one by one. Starting from the depot, each route is constructed by iteratively adding a task, which is viewed as an arc here (different directions of servicing tasks lead to different solutions). The arc to be added is the one that is nearest to the current end of route (i.e. the head of last arc) among all arcs that do not exceed the capacity. If there is more than one nearest arc, one of them is chosen according to a given tie-breaking rule. Golden et al. (1983) proposed the following five tie-breaking rules: the next arc (i, j) to be added to the route is the one that

- Rule 1: maximises the shortest distance from the new end of route j to the depot;
- Rule 2: minimises the shortest distance from the new end of route j to the depot;
- Rule 3: obeys Rule 1 if the vehicle's remaining capacity is at least half the whole capacity, and obeys Rule 2 otherwise;
- Rule 4: maximises the ratio demand to cost, i.e. d_{ij}/c_{ij};
- Rule 5: minimises the ratio demand to cost.

If there still remains more than one arc after using a tie-breaking rule, then one of them is chosen randomly. Each rule gives one solution and the best among those five is chosen as the output of the algorithm.

Several variants of the Path Scanning algorithm have also been proposed. Belenguer et al. (2006) suggested two alternative ways of breaking ties: (i) *random criterion* – randomly choosing one of the above five tie-breaking rules for the addition of each arc instead of using the same rule for the whole solution, and (ii) *random link* – a tie is broken by simply choosing one of the nearest edges randomly (so the above five tie-breaking rules are not used). These alternative methods allow more solutions to be generated and hence increase the probability of obtaining better solutions. Their computation results show that these methods could improve the path scanning algorithm provided a sufficient number of solutions are explored.

Santos et al. (2009) proposed the "ellipse rule", which re-defines the set of candidate arcs in the path scanning algorithm. It is motivated by the intuition that when the vehicle is nearly full, it should prioritise returning to the depot. So instead of considering arcs that are nearest to the current end of the route, a candidate arc should be near the shortest path between the current end of the route and the depot. More precisely, when the capacity of a vehicle is less than $\beta \times \frac{\text{total demand}}{\text{number of tasks}}$, for some parameter (real number) β, an arc is randomly chosen from the set of arcs (i, j) that are nearest to the current end of the route *and* satisfy the following inequality:

$$D(v_e, i) + c_{ij} + D(j, v_0) \leq \frac{\text{total cost of all tasks}}{\text{number of tasks}} + D(v_e, v_0), \tag{2.13}$$

where v_e is the current end vertex of the route, v_0 the depot, and $D(u, v)$ the shortest distance between two vertices u and v. Notice that the inequality is similar to one that defines an ellipse with the foci at v_e and v_0 (hence the name "ellipse rule"). If no tasks satisfy the condition (2.13), then the route is closed and, if there still remain some tasks to be completed, a new route is constructed. Computational results given by Santos et al. (2009) indicate that incorporating the ellipse rule into the path scanning algorithm (with random link) helps improve the solution with very little increase in computational time.

Ulusoy's Partitioning Ulusoy (1985) proposed a constructive heuristic that consists of two main phases. First, in the "Route" phase, a "giant" cycle servicing all tasks is constructed by temporarily omitting the capacity constraint. If all tasks form a connected subgraph, the cycle can be found by solving the Chinese Postman Problem. Otherwise, it is the Rural Postman Problem. Suppose that the tasks in the order of being serviced in the route are $T_1, T_2, \ldots, T_n$, where n is the number of tasks.

Next, in the "Cluster" phase, the giant cycle is divided into a number of feasible routes. To this end, an auxiliary graph $G^* = (V^*, E^*)$ is constructed with $V^* = \{0, 1, 2, \ldots, n\}$ containing $n + 1$ vertices. The edge set E^* contains all edges $\{i, j\}$ ($i, j \in V^*$ with $i < j$) such that the total demand of the tasks $T_{i+1}, T_{i+2}, \ldots, T_j$ does not exceed the capacity. The cost of edge $\{i, j\}$ is equal to the cycle composed of a shortest path from the depot to T_i, the part of the giant cycle from T_i to T_j, and a shortest path from T_j to the depot (so each edge in E^* represents a feasible route in G). After constructing the auxiliary graph G^*, a feasible solution can be found by determining a shortest path in G^* from 0 to n.

2.6 Metaheuristic Algorithms

A solution obtained from a constructive heuristic algorithm can usually be further improved. This can be achieved by iteratively making changes to the solution. A variety of frameworks have been proposed to provide a guide as to how to apply given changes to the solution in each iteration; such frameworks are also called *metaheuristics*. This section provides a review of metaheuristic algorithms that have been applied to the CARP. For a more complete review of metaheuristic algorithms, see Talbi (2009).

Depending on the number of solutions that are considered in each iteration, there are two main types of metaheuristics: single-solution-based and population-based.

2.6.1 Single-Solution-Based Metaheuristics

For single-solution-based metaheuristics, a solution iteratively undergoes a *neighbourhood move*, which is an operator that makes small changes to the solution. Following the notation in Section 1.2, let $R_1 = (v_0, a_1, \ldots, a_{n_1}, v_0)$ and $R_2 = (v_0, b_1, \ldots, b_{n_2}, v_0)$ be two routes. Neighbourhood moves for the CARP that have been used in the literature include the following:

- *Insertion* or *Relocate* – Remove (a small number of) consecutive arcs from one route and insert them into another route. It is possible to reverse the direction of an arc before the insertion. Usually, both directions are tested and the better one is chosen. The move is also called *Single Insertion* or *Double Insertion*, respectively when one or two arcs are considered. Beullens et al. (2003) considered Single Insertion, while Greistorfer (2003), Lacomme et al. (2004), Brandão and Eglese (2008), and Tang et al. (2009) considered both Single Insertion and Double Insertion. In the work of Brandão and Eglese (2008), there is an additional restriction: the insertion is only allowed between consecutive arcs that are joined by a shortest path. In other words, the insertion can take place between arcs b_{j-1} and b_j if the head of b_{j-1} is different from the tail of b_j.

- *Swap* or *Exchange* – Swap m consecutive arcs from one route with n consecutive arcs from another route, for some (small) numbers m, n (commonly, m and n are either 1 or 2). As is the case with Insertion, both directions are tested when inserting arcs and the better one is chosen. This type of neighbourhood move was used by Beullens et al. (2003), Greistorfer (2003), Lacomme et al. (2004), Brandão and Eglese (2008), and Tang et al. (2009). In these studies except Brandão and Eglese (2008), the arcs from one route are inserted at the position of the arcs from the other route. In the algorithm proposed by Brandão and Eglese (2008), one arc is selected from each route (i.e. $m = n = 1$), and the arcs can be inserted in any positions of their respective routes, as long as those positions are between consecutive arcs that are joined by a shortest path.

- *2-Opt for a single route* – Pick a subsequence of arcs in a single route and reverse it. This neighbourhood move was used by Beullens et al. (2003) (who referred to this move as "reverse"), Lacomme et al. (2004) and Tang et al. (2009).

- *2-Opt for two routes* – Cut each route into two subsequences, so R_1 becomes $(v_0, \dots, a_{i-1})$ and $(a_i, \dots, v_0)$ and R_2 becomes $(v_0, \dots, b_{j-1})$ and $(b_j, \dots, v_0)$ for some indices i, j. Then join the subsequences to form two different routes. This neighbourhood move was used by Beullens et al. (2003), Lacomme et al. (2004), and Tang et al. (2009).

- *Flip* – Pick a subsequence of arcs $(a_i, \dots, a_{i+k})$ and replace it with the same arcs in the opposite order, i.e. $(a_{i+k}, \dots, a_i)$. This neighbourhood move was used by Beullens et al. (2003).

One of the simplest single-solution-based metaheuristics is Steepest Descent: apply the best move to the solution in each iteration until no improvement can be obtained. However, one crucial issue of Steepest Descent is that it can be stuck at a local optimum that is not a global optimum. In order to circumvent this drawback, several other metaheuristics have been proposed. Those that have been applied to the CARP include tabu search and guided local search.

Tabu search Introduced by Glover (1989), tabu search is a metaheuristic that attempts to avoid the problem of being stuck at a local optimum by two main procedures: (i) allowing acceptance of moves that worsen the solution in the absence of improving moves, and (ii) declaring some moves *tabu*, i.e. they are prohibited, to avoid returning to solutions that have already been visited and being stuck in a cycle of solutions. Tabu search has been applied to the CARP by Hertz et al. (2000); Greistorfer (2003); Brandão and Eglese (2008). A pseudocode of tabu search is shown in Algorithm 1.

A move is said to be *admissible* if either it is not tabu or it is tabu but satisfies a certain criterion called an *aspiration criterion*. The purpose of an aspiration criterion is to help prevent tabu search from being too restrictive; a tabu move could sometimes lead to a solution that is better than the current best solution, so dismissing the move simply because it is tabu would result in a lost opportunity to improve the solution.

Algorithm 1 Tabu search

```
 1: given an initial solution $S_0$
 2: $S = S_0$, $S_{\text{best}} = S_0$, $T = \phi$  ▷ initialise current solution, best solution, and tabu list
 3: repeat
 4:     $M(S)$ = the set of admissible moves that can be applied to $S$
 5:     if $M(S)$ is not empty then
 6:         $m$ = the best admissible move in $M(S)$
 7:         apply $m$ to $S$
 8:         if $S$ is better than $S_{\text{best}}$ then
 9:             $S_{\text{best}} = S$                ▷ update best solution
10:         update $T$
11: until stopping criteria are satisfied
12: return $S_{\text{best}}$
```

One common aspiration criterion (used by e.g. Hertz et al. (2000); Greistorfer (2003); Brandão and Eglese (2008)) is based on the aforementioned scenario: a tabu move is allowed to be selected if it leads to a solution that is better than the best one found so far. Since this solution cannot have been visited (otherwise, it would have been the best solution), this aspiration criterion is consistent with one of the main purposes of tabu search, which is to avoid a cycle of solutions.

The tabu list T keeps track of tabu moves, although it does not necessarily contain full descriptions of the moves. Some existing tabu search algorithms use T to record certain attributes of visited solutions (usually attributes that result from the selected move in each iteration), and a move is regarded as tabu if it causes attributes that are currently in T. To avoid being too restricted, attributes only stay in T for a given number of iterations; this number is also referred to as a *tabu tenure*. One of the studies that define tabu moves based on solution attributes is the study of Hertz et al. (2000). Neighbourhood moves in their tabu search algorithm are of the form "moving a task from one route to another." When a task is removed from a route, it is not allowed to return to the same route for a certain number of iterations. A similar idea was implemented in the tabu search algorithm proposed by Greistorfer (2003). Interestingly, Greistorfer (2003) viewed a tabu list as an array, or more precisely, a two-dimensional array Rec, each of whose elements $Rec(s, p)$ corresponds to a task s and a route p. When task s is removed from route p, the element $Rec(s, p)$ is set to the current iteration number. Consequently, a move that moves task s back to route p is regarded as tabu in the iteration τ if $\tau - Rec(s, p)$ is less than or equal to a given tabu tenure. This helps the algorithm check tabu moves in a more efficient way than going through a list of attributes.

Guided local search Guided local search (GLS) was introduced by Voudouris and Tsang (1996). The main idea of GLS is to augment the objective function whenever a local search reaches a local optimum. To achieve this, a set of features that can appear in a solution are defined, and each feature is associated with a penalty term; the augmented objective function is equal to the sum of the original objective function (i.e. the value to be optimised in the problem) and the penalty terms, i.e.

$$f(S) = g(S) + \lambda \sum_{i \in \mathcal{I}} p_i I_i(S), \tag{2.14}$$

where f is the augmented objective function, g is the original objective function, $\mathcal{I}$ is a set of features, p_i is a penalty term for each feature $i \in \mathcal{I}$, $I_i(S)$ is an indicator function ($I_i(S) = 1$ if a solution S contains feature i, and $I_i(S) = 0$ otherwise), and λ is a parameter for adjusting the balance between g and the penalty sum.

Each time a local optimum is reached, a certain feature that appears in the local optimum is selected and the corresponding penalty term is increased. To decide which feature should be selected, a so-called *utility function* is defined. Usually, the utility function is defined in such a way that a feature that causes the solution to have low quality in some way has a high value of utility function. Once a feature is selected, increasing the corresponding penalty term in the augmented objective function guides the search away from solutions that have the selected feature.

The local search is executed a given number of times, each time with the augmented objective function, and the best solution with respect to the original objective function that has been found throughout the search is returned. A pseudocode of guided local search is shown in Algorithm 2, where a local search $\mathcal{L}$ is performed based on the augmented objective function (2.14).

GLS was applied to the CARP by Beullens et al. (2003). In their algorithm, a penalty term $p(a, b)$ and the utility function $u(a, b)$ are defined for each pair of arcs (a, b). The utility function $u(a, b)$ is defined as the ratio

$$u(a, b) = \frac{\text{shortest distance from the head of } a \text{ to the tail of } b}{1 + p(a, b)}. \tag{2.15}$$

All penalty terms $p(a, b)$ are initially set to 0. After each execution of the local search, the algorithm checks all pairs of arcs (a, b) that are serviced consecutively in the same route in the solution and finds the pair with the largest utility function. The penalty term corresponding to the pair of arcs (a, b) with the largest utility function $u(a, b)$ is then increased by 1. In other words, this penalises the pair of arcs

Algorithm 2 Guided local search

```
1: given an initial solution $S_0$, a local search $\mathcal{L}$, a set of features $\mathcal{I}$
2: set a penalty $p_i = 0$ for each feature $i \in \mathcal{I}$
3: set $S = S_0$ and $S_{\text{best}} = S_0$
4: repeat
5:     apply the local search $\mathcal{L}$ to $S$ with respect to the augmented objective function; let $S'$ be the output
6:     if $S'$ is better than the current $S_{\text{best}}$ with respect to the original objective function then
7:         set $S_{\text{best}} = S'$
8:     compute the utility function $u_i$ for each feature $i \in \mathcal{I}$ based on $S'$
9:     find a feature $i$ such that $u_i$ is maximum
10:    increase the corresponding penalty $p_i$ by 1
11:    set $S = S'$
12: until stopping criteria are satisfied
13: return $S_{\text{best}}$
```

that are serviced consecutively but are far away from each other. Notice that the denominator $(1 + p(a, b))$ in the utility function helps avoid penalising the same pair of arcs repeatedly.

2.6.2 Population-Based Metaheuristics

For population-based metaheuristics, multiple solutions are taken into account when producing new solutions in each generation. Population-based metaheuristics that have been implemented to find solutions to the CARP include ant colony optimisation and genetic algorithms.

Ant colony optimisation Ant colony optimisation (ACO) is a metaheuristic that is inspired by how ants work together to find the shortest path between their nest and a food source. Along a path it walks past, an ant leaves a chemical substance, called *pheromone*, which guides other ants that travel after it: among various possible paths, an ant will more likely choose a path with a more concentrated pheromone. If there is more than one path between the nest and the food source, a shorter path will tend to have a more concentrated pheromone, thus attracting more ants to follow that path. In terms of solving the CARP (or routing problems in general), each ant is regarded as an agent that constructs a solution. Each solution attribute (e.g. a pair of tasks that are serviced consecutively) is associated with a “pheromone value.” The pheromone values are updated after all ants finish constructing solutions.

Changes in the pheromone values of attributes that are used by some ants depend on qualities of the solutions that contain those attributes. The next generation of ants then construct solutions based on the current pheromone values: the higher the value, the higher probability the ant will use the corresponding attribute. This process is repeated for a given number of generations, and the algorithm returns the best found solution. ACO algorithms may also involve an *evaporation* process, whereby the pheromone values "evaporate" or reduce before each generation starts. This helps the algorithms focus more on solutions that are constructed in more recent generations.

When applying ACO to the CARP (or other optimisation problems in general), there are at least two questions that need answering: (i) What is represented by a walk of an ant? (ii) What feature of a solution is a pheromone associated with? In an ACO algorithm that Santos et al. (2010) proposed for the CARP, each ant constructs a single route, which they referred to as a "Single Network Tour (SNT)," containing all tasks. Each "walking step" of an ant corresponds to adding an oriented task (i.e. a task with a specified direction) to its SNT. A pheromone value τ_{ij} is associated with a pair of oriented tasks (a_i, a_j), representing an ant "walking from task a_i to task a_j." Suppose that a_i is the last task in the SNT, the next task to be added in each step depends on pheromone values. There are several rules for determining the next task, for example, choosing task a_j with the highest pheromone value τ_{ik} among all possible tasks a_k, or randomly choosing one of possible tasks with probability proportional to its pheromone value. During the construction of a SNT, the capacity constraint is omitted. Once all tasks are added to a SNT, a feasible solution is obtained by means of Ulusoy's Partitioning (see Section 2.5). The total distance of the feasible solution is used as a measure of the quality of the corresponding SNT. After all ants construct their SNTs (the number of ants is a parameter to be specified), pheromone values are updated. The increase in the pheromone value for each pair of oriented task depends on the quality of SNTs that contain it: a higher quality SNT contributes a larger increase. For an exact way of calculating pheromone values, see Bullnheimer et al. (1997), whose method of updating pheromone values was adopted by Santos et al. (2010).

Genetic algorithms A genetic algorithm (GA) is a search algorithm that is inspired by the way biological entities have evolved: In simple terms, each entity has a "chromosome", which determines the entity's features. Chromosomes are combined to produce new chromosomes and hence new entities. Ideally, a combination of chromosomes that yield "good" features would lead to new chromosomes with good or even better features. Main concepts in a genetic algorithm include representation of chromosomes, crossover operators, population replacement, and mutation operators. A general genetic algorithm operates as follows: First, an initial set of solutions (i.e. initial population) is generated by some constructive heuristics. Then, in each iteration (also called *generation*), pairs of solutions, called *parents*, are selected and each pair undergo a crossover operator to produce new solutions, called *offsprings*. The offsprings are added to the population and some solutions in the population that are deemed low quality are discarded.

- **Chromosomes.** A chromosome in a genetic algorithm can be viewed as a code, usually in the form of a sequence of certain items or symbols, that can be converted into a solution. For the CARP, this may be as simple as a sequence of all tasks (Lacomme et al., 2004). To evaluate the chromosome, it has to be converted into a feasible solution. For example, a sequence of all tasks can be divided into feasible routes by means of Ulusoy's partitioning (described in Section 2.5).

- **Crossover operators.** A crossover operator produces new "offspring" chromosomes from existing (usually two) "parents" chromosomes. This is achieved by cutting each given chromosome into several subsequences and recombining them in some way to form new chromosomes. Examples of crossover operators are the linear order crossover (LOX) and the order crossover (OX). Given two parents $P_1 = (P_1(1), \ldots, P_1(n))$ and $P_2 = (P_2(1), \ldots, P_2(n))$, where each $P_k(l)$ is a task and n is the number of all tasks, and given "cutting" indices i, j with $1 \leq i \leq j \leq n$, the LOX produces an offspring C_1 by first setting $C_1(l) = P_1(l)$ for $l = i, \ldots, j$. Then, the remaining tasks $C_1(1) \ldots, C_1(i-1), C_1(j+1), \ldots, C_1(n)$ are obtained by taking each task in P_2 successively form $P_2(1)$ to $P_2(n)$, ignoring tasks that are already in C_1. For the OX, an offspring C_1 is produced by first setting $C_1(l) = P_1(l)$ for $l = i, \ldots, j$ (the same as LOX) but the remaining tasks are obtained in a different order: the tasks $P_2(1)$ to $P_2(n)$) (again, ignoring those that are already in C_1) are successively placed at $C_1(j+1), \ldots, C_1(n)$ before $C_1(1) \ldots, C_1(i-1)$. After that, tasks in C_1 are rotated so that $C_1(i) = P_1(i)$. For both LOX and OX, the other offspring C_2 is produced in the same way as C_1 but with P_1 in place of P_2 and vice versa

- **Population replacement.** There are two schemes for evolving the population: incremental and generational (Lacomme et al., 2004). The main difference between these schemes is the number of parents that undergo crossover operators in each generation. In an incremental GA, two parents undergo a crossover operator, and one of the offsprings (selected at random) replaces one of the solutions that is deemed "low quality", e.g. the worst solution or one of the solutions whose total cost is greater than the median cost of the population (assuming the minimisation problem). In a generational GA, the whole population is divided into pairs and each pair undergoes a crossover operator. This results in twice as many chromosomes in the population. All the chromosomes are then ranked from "best" to "worst", and the second half of the population are discarded.

- **Mutation operators.** Some genetic algorithms (Lacomme et al., 2001, 2004) also involve mutation operators, which make some changes to the offsprings after the crossover operator with a given probability. This potentially introduces features that do not exist in the parents, i.e. it can promote diversity in the population, thus preventing early convergence of the population. Some mutation operators are specifically designed to improve the quality of the solutions, for example, a mutation operator can be a local search; a genetic algorithm that involves a local search in this way is also called a *memetic algorithm*.

For examples of how genetic or memetic algorithms can be applied to the CARP, see Lacomme et al. (2004), Tang et al. (2009), and Fu et al. (2010).

2.6.3 Further Improvement Methods

Several methods have been devised to further improve the performance of metaheuristic algorithms as follows. Notice that they are not limited to a particular type of metaheuristic algorithms.

Intensification A search may be enhanced by exploring solutions near a high-quality solution more extensively. One possible idea is to force the search to go back to the best solution that has been found so far and continue the search from there. This idea was implemented in a tabu search algorithm proposed by Brandão and Eglese (2008).

Merge-split operator Tang et al. (2009) introduced a merge-split operator as a "large-step" neighbourhood move to complement other neighbourhood moves which are “small-step” (i.e. they make relatively small changes to the solution). There are two reasons that motivate this operator: (1) using only small-step neighbourhood moves may make the search converge slowly if the solution space is large; (2) if the capacity constraint is tight, it may be unlikely to move from one feasible solution to another without encountering infeasible solutions. Making a large change to the solution can help the search “jump” from one region of the solution space to another. The merge-split operator is a combination of the Path-Scanning algorithm and Ulusoy’s partitioning described in Section 2.5. Given a number of routes, first use the Path-Scanning algorithm to create a single “giant” route that contains all tasks in the routes. Then, use Ulusoy’s partitioning to split the route into smaller feasible routes.

Accepting infeasible solutions In some existing algorithms for the CARP (Hertz et al., 2000; Beullens et al., 2003; Brandão and Eglese, 2008), infeasible solutions are allowed, although a penalty is added to the objective function (the total distance) when assessing the quality of such solutions. One commonly used penalty term is of the form

$$\lambda \sum_{R} max\{0, D(R) - Q\}, \tag{2.16}$$

where λ is an adjustable parameter, $D(R)$ is the sum of demands serviced in route R, and Q is the capacity. In other words, the penalty is proportional to the total excess of demands. A common rule for adjusting the parameter λ is to double it when obtaining infeasible solutions for a certain number of consecutive iterations, and to halve it when obtaining feasible solutions for a certain number of consecutive iterations. Regarding an initial value of λ, Hertz et al. (2000); Beullens et al. (2003) used information about the shortest distances between the depot and the endpoint of all tasks: Let $D(v_0, v)$ denote the distance of a shortest path between the depot v_0 and another vertex v. Hertz et al. (2000) set the initial value of λ to the average of $D(v_0, v)$ for all vertices v that are endpoints of tasks, and Beullens et al. (2003) used a multiple of the maximum $D(v_0, v)$ among all vertices v that are endpoints of tasks. In contrast, Brandão and Eglese (2008) opted for a very simple choice: setting the initial value of λ to 1.

Global repair operator Mei et al. (2009) proposed a global repair operator, which guides the search back to feasible solutions with a “clearer direction” than adding penalties to the objective function. Given an infeasible solution, i.e. the capacity is exceeded in some routes, this operator attempts to find a feasible solution by transferring services of tasks between routes while keeping the path in each route unchanged. In particular, a task is allowed to be transferred from its current route to another route, say route R, only if the edge corresponding to the task is also traversed in route R. This operator can be useful when a search encounters a low-cost but infeasible solution.

2.6.4 Performance of Existing Metaheuristic Algorithms for the CARP

The performance of metaheuristic algorithms for the CARP is commonly reported based on two measures: percentage deviation from the best known lower bound or (if known) the optimal cost, that is,

$$\text{percentage deviation} = \left(\frac{\text{solution cost} - \text{lower bound}}{\text{lower bound}}\right) \times 100, \tag{2.17}$$

and computation time. Figures 2.3 and 2.4 show the distributions of these two measures on a number of benchmark instance sets for various existing metaheuristic algorithms for the CARP, including the following:

- a tabu search algorithm ('CARPET') by Hertz et al. (2000),
- a guided local search algorithm ('GLS') by Beullens et al. (2003),
- a memetic algorithm ('MA') by Lacomme et al. (2004),
- a tabu search algorithm ('TSA') by Brandão and Eglese (2008),
- a memetic algorithm ('MAENS') by Tang et al. (2009),
- an ant colony optimisation algorithm ('Ant-CARP') by Santos et al. (2010).

Details of the benchmark instance sets are given in Table 2.1. Solution costs and computation times on individual instances are shown in Tables A.1 to A.6 in Appendix A. Note that not all of the algorithms have been tested on every benchmark instance set.

Some of the above studies report results of more than one variant of their algorithms. For the guided local search algorithm by Beullens et al. (2003), the results with 10^5 iterations and 5×10^5 iterations are denoted in Figures 2.3 and 2.4 by 'GLS1' and

Table 2.1: Characteristics of benchmark instance sets for the CARP

Instance set[1]		Number of instances	Numbers of tasks	Minimum numbers of vehicles needed[2]
VAL (Benavent et al., 1992)		34	34 to 97	2 to 10
BMCV (Beullens et al., 2003)	subset C	25	32 to 121	3 to 12
	subset D	25	32 to 121	2 to 6
	subset E	25	28 to 107	4 to 12
	subset F	25	28 to 107	2 to 6
EGL (Belenguer and Benavent, 2003)		24	51 to 190	5 to 35

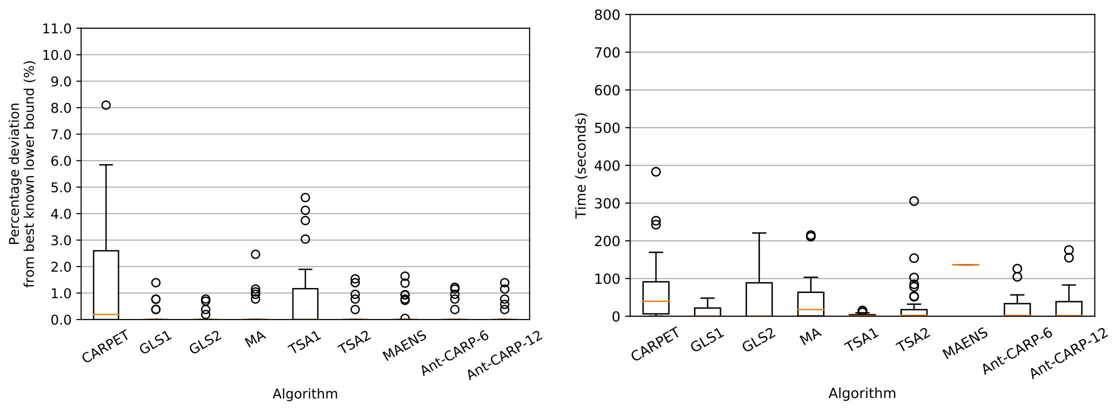
[1] These instances can be downloaded from the website `http://logistik.bwl.uni-mainz.de/benchmarks.php` (last accessed 21 March 2018).

[2] The minimum number of vehicles needed for each instance is the smallest integer that is greater than or equal to the total demand on that instance divided by the vehicle capacity.

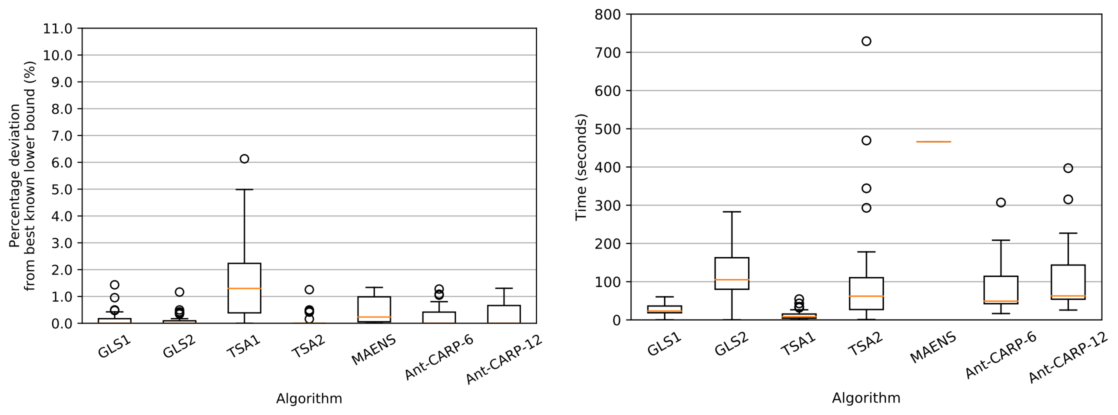
'GLS2', respectively. In the original paper of Brandão and Eglese (2008), their main tabu search algorithm is called 'TSA Version 1', whereas 'TSA Version 2' results from their attempt to obtain better solutions by running their main algorithm several times successively; these variants are denoted here by 'TSA1', and 'TSA2', respectively. In the work of Santos et al. (2010), their ant colony optimisation algorithm involves local search with a variety of neighbourhood moves. They consider two versions of their algorithm: one involves 6 types of moves in the local search, and the other involves 12 types (including the former 6 types). These variants are denoted here by 'Ant-CARP-6' and 'Ant-CARP-12', respectively.

As the algorithms are tested on different computer specifications, computation times reported in their original papers need to be adjusted in order to facilitate a fair comparison. For this reason, the computation times shown in Figures 2.3 and 2.4 result from multiplying the following factors to the computation times reported in the original papers: 0.2 for 'CARPET' (based on a Silicon Graphics Indigo2 machine with a 195 MHz processor), 0.5 for 'GLS' (based on a Pentium II 500 MHz processor), 1.0 for 'MA' (based on a Pentium III 1 GHz processor), 1.4 for 'TSA' (based on a Pentium Mobile 1.4 GHz processor), 2.0 for 'MAENS' (based on an Intel Xeon E5335 2.0GHz processor), and 1.0 for 'Ant-CARP' (based on a Pentium III 1 GHz processor). Note that in the original paper of Tang et al. (2009), there appears no information about computation time of their algorithm ('MAENS') on each instance; only the average computation time over all instances in each benchmark set is reported.

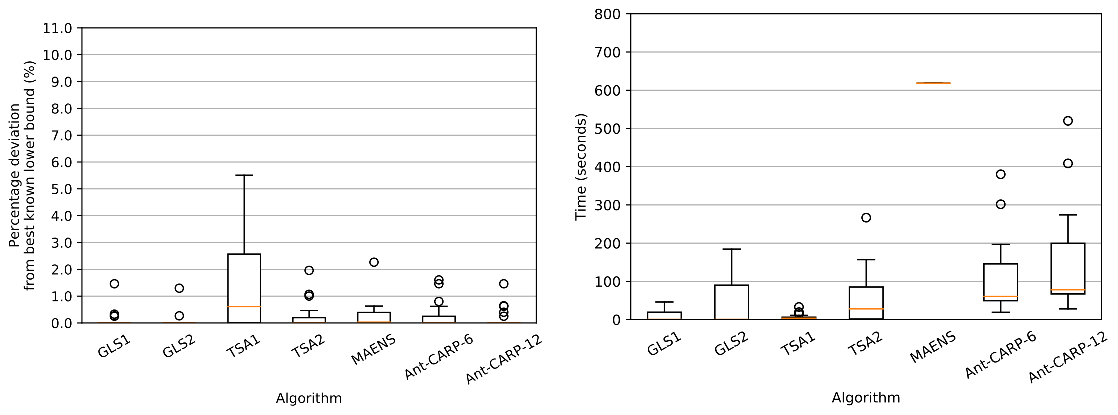
Let us now review the performance of existing single-solution-based metaheuristic



(a) VAL instance set

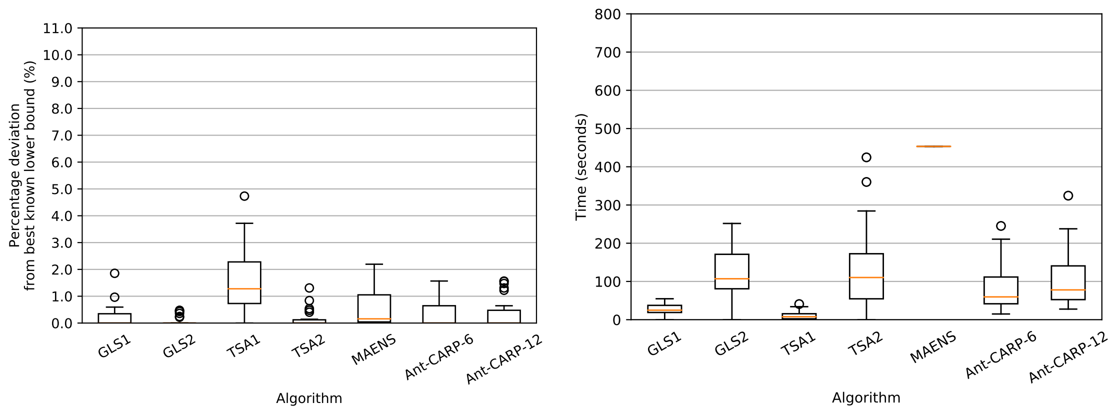


(b) BMCV instance set (subset C)

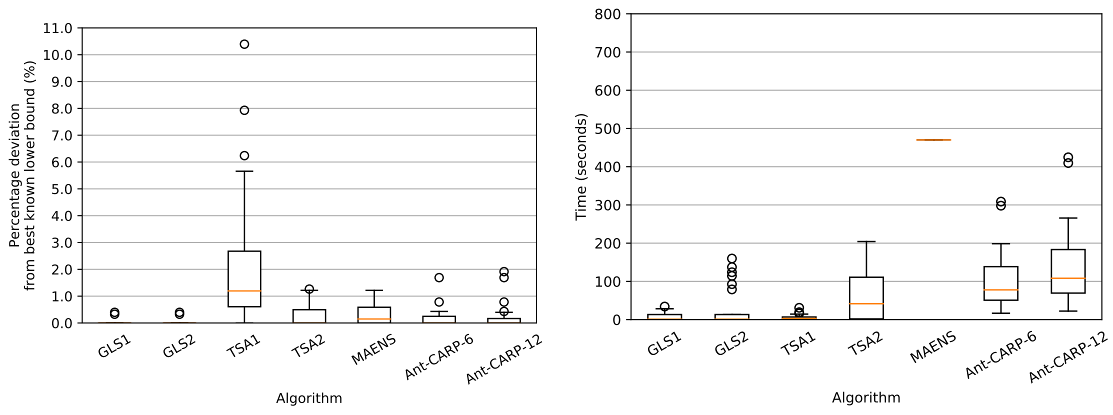


(c) BMCV instance set (subset D)

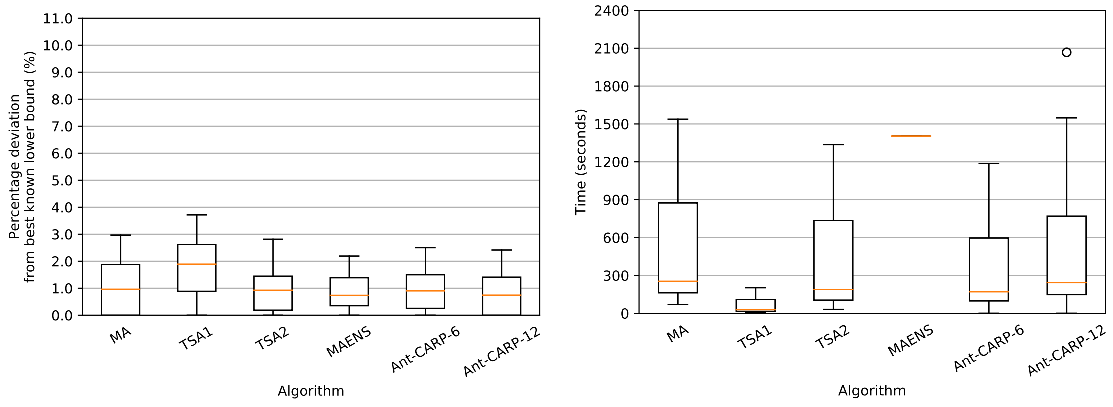
Figure 2.3: Performance of existing metaheuristic algorithms for the CARP on VAL and BMCV benchmark instance sets; for each instance set, the left figure shows the distributions of percentage deviations from best known lower bounds on all instances in the given set, and the right figure shows the distributions of computation time on all instances in the given set



(a) BMCV instance set (subset E)



(b) BMCV instance set (subset F)



(c) EGL instance set

Figure 2.4: Performance of existing metaheuristic algorithms for the CARP on BMCV and EGL benchmark instance sets; for each instance set, the left figure shows the distributions of percentage deviations from best known lower bounds on all instances in the given set, and the right figure shows the distributions of computation time on all instances in the given set

algorithms for the CARP. Both CARPET and TSAs are based on tabu search, but TSAs appear to give better solutions overall (see Figure 2.3(a)). This could be because CARPET only involves one type of neighbourhood move (moving a task from one route to another), whereas TSA involves three types of neighbourhood moves (Single Insertion, Double Insertion, and Swap), thus exploring neighbour solutions more extensively. Interestingly, TSAs perform better than CARPET despite less computation time. Regarding guided local search, GLS2 generally uses a greater amount of time than GLS1 as expected since GLS2 involves 5 times as many iterations as GLS1 does. However, the solutions given by GLS2 are only slightly better than GLS1. Overall, GLS1 and TSA1 give fairly good solutions within a modest amount of time, which would be ideal choices for tackling dynamic CARP.

Regarding population-based metaheuristic algorithms, MA, MAENS, and Ant-CARP all have excellent performance, giving solutions close to optimality across different benchmark instance sets. However, they tend to use a large amount of computation time compared with the other algorithms. If these types of algorithms are to be implemented for dynamic CARP, the issue of large computation time would need to be addressed. Furthermore, memetic algorithms and ant colony optimisation algorithms generally involve relatively many parameters. Particular attention needs to be paid on tuning their parameters so that good algorithm performance is maintained despite changes in the problem. This means that choices of the parameters should be robust (i.e. performing well across different instances), or alternatively, the algorithms should be able to adjust the parameters promptly as the problem changes over time.

It should be noted that the existing studies on the static CARP commonly aim to obtain the best solutions possible. In some cases, this is achieved only after a relatively large amount of computation time. However, when tackling a dynamic version of the CARP, we are most likely interested in achieving a fairly good solution within limited time. To identify algorithms that can improve solution quickly, information about how different algorithms perform over a period of time (or over a range of iterations) would be helpful. Such information, however, can rarely be found in the literature at the time of writing because the performance of existing algorithms for the CARP is usually reported based on specific stopping criteria. As will be seen later, Chapter 3 presents a novel analysis that investigates the performance of tabu search for the CARP over a range of iterations.

Costs of (feasible) solutions from metaheuristic algorithms can be viewed as upper bounds to optimality. The next section reviews ways of obtaining lower bounds and provably optimal solutions.

2.7 Lower Bounds and Exact Algorithms

Despite their ability to find or improve a solution to an intractable problem such as the CARP, heuristic algorithms alone do not provide information about whether or not a solution is optimal or about how far a solution is from optimality. In the absence of optimal solutions, one way to estimate the difference between the total distance of a given solution and the optimal value is to compute a lower bound. In fact, if the total distance of a solution is equal to a lower bound, then it is a proof that the solution is optimal. Existing methods for computing lower bounds are based on two main approaches: graphically and algebraically. In much early work, lower bounds were computed graphically. This includes the Matching Lower Bound (Golden and Wong, 1981), the Node Scanning Lower Bound (Assad et al., 1987), the Matching Node-Scanning Lower Bound (Pearn, 1988), LB1 and LB2 (Benavent et al., 1992), and the Node Duplication Lower Bound (Saruwatari et al., 1992). Those lower bounds are based on the following idea: First, notice that the total distance of traversals with service is fixed (i.e. it is the same in all feasible solutions in a given instance), so when computing a lower bound, we can focus on the total distance of traversals without service, also called the *deadheading cost*. Given a CARP instance on a graph G, construct an "augmented" graph $\tilde{G}$ (usually by amending the graph G in a certain way) and associate each feasible solution S on G with a perfect matching M on $\tilde{G}$ in such a way that the cost of M is less than or equal to the deadheading cost in S. A lower bound on the deadheading cost can then be computed by finding a minimum cost perfect matching in $\tilde{G}$, which can be performed in polynomial time (see Cook and Rohe (1999), for example). Different lower bounds differ in the way the augmented graph $\tilde{G}$ is constructed.

Lower bounds can also be computed algebraically by solving a linear programming relaxation of the CARP such as the "one-index" formulation proposed by Belenguer and Benavent (2003). A linear programming relaxation can be solved by, for example, a branch-and-bound method or by a ready-to-use LP solver (such as CPLEX). Some techniques such as adding cutting planes or column generation may also be used to improve a lower bound. Existing methods for computing lower bounds algebraically include the work of Belenguer and Benavent (2003), Beullens et al. (2003), Baldacci and Maniezzo (2006), Longo et al. (2006), Martinelli et al. (2011), Bode and Irnich (2012), and Bartolini et al. (2013).

With appropriate constraints, some LP formulations of the CARP can give a feasible, and sometimes even optimal, solution for the CARP. An algorithm that finds an optimal solution in this way is referred to as an *exact algorithm*. Even though exact algorithms are guaranteed to give optimal solutions, computational results in existing work show that exact algorithms often require a substantial amount of time to find an optimal solution. For example, in the computational results reported by Bartolini et al. (2013), their exact algorithm was unable to obtain an optimal solution within 6 hours of computation time (using an Intel Xeon E5310 1.6GHz CPU with 8GB RAM) on 34 instances from 120 instances in the BMCV instance set (Beullens et al., 2003); the number of tasks in those 34 instances ranges from 54 to 121 and the number of vehicles needed ranges from 3 to 12. This suggests that the ability of exact algorithms to find solutions quickly is still limited to relatively small instances of the CARP. For this reason, heuristic algorithms would be preferable for finding a solution in a dynamic CARP, where it is more important to obtain a feasible solution within a short time after the problem changes (so that a service can be operated smoothly) than to achieve optimality.

2.8 Transforming Arc Routing into Vehicle Routing

Vehicle routing is similar to arc routing except that demands are associated with vertices instead of edges. As vehicle routing has been studied more extensively than arc routing[3], this offers the possibility of solving an arc routing problem by transforming it into a vehicle routing problem and using algorithms for vehicle routing, which are already readily available. This section gives an overview of ways to transform an arc routing problem into a vehicle routing problem that have been proposed in the literature.

Pearn et al. (1987) proposed the following method of transforming the CARP into the Capacitated Vehicle Routing Problem (CVRP). Given a CARP instance on a graph G, each task $\{i, j\}$ is associated with two "side" vertices s_{ij}, s_{ji} and another "middle" vertex m_{ij}; as will be seen below, this transformation has a mechanism (based on edge costs) that forces m_{ij} to be serviced between s_{ij} and s_{ji}. A complete graph $G_c = (V_c, E_c)$ is then constructed with the vertex set

$$V_c = \bigcup_{\{i,j\} \in R} \{s_{ij}, s_{ji}, m_{ij}\} \cup \{v_0\}$$

[3]On 11 July 2017, a search on Scopus (www.scopus.com) with the keyword "vehicle routing" returned 13,378 document results, while "arc routing" returned 1,197 document results. Moreover, a search on Google Scholar (scholar.google.co.uk) returned about 856,000 results for "vehicle routing" and 180,000 results for "arc routing."

(with vertex v_0 denoting the depot). Let c_{ij} and d_{ij} be the cost and the demand of edge $\{i, j\}$ in G, respectively. Also, let $D(i, j)$ be the shortest distance between vertices i and j in G. The costs $c(u, v)$ for edges $\{u, v\} \in E_c$ are defined as follows:

$$\begin{aligned} c(s_{ij}, s_{kl}) &= \begin{cases} \frac{1}{4}(c_{ij} + c_{kl}) + D(j,k) & \text{if } \{i,j\} \neq \{k,l\} \\ 0 & \text{otherwise} \end{cases} \\ c(v_0, s_{ij}) &= \frac{1}{4}c_{ij} + D(v_0, i) \\ c(m_{ij}, v) &= \begin{cases} \frac{1}{4}c_{ij} & \text{if } v = s_{ij} \text{ or } s_{ji} \\ \infty & \text{otherwise.} \end{cases} \end{aligned} \tag{2.18}$$

For each task $\{i, j\}$, the demands of $s_{ij}, s_{ji}, m_{ij} \in V_c$ are set to $\frac{1}{3}d_{ij}$ (any other positive values are also possible as long as the demands of s_{ij}, s_{ji}, m_{ij} add up to d_{ij}). The edge costs on E_c (2.18) are defined in such a way that a CVRP solution has the same total cost as the corresponding CARP solution; in particular, the cost of task $\{i, j\} \in R$ is divided into four parts in G_c, namely an edge going into s_{ij}, edge (s_{ij}, m_{ij}), edge (m_{ij}, s_{ji}), and an edge leaving s_{ji}. Furthermore, notice that by definition of $c(m_{ij}, v)$, a CVRP solution with finite cost would service m_{ij} between s_{ij} and s_{ji}. This ensures that a solution to this CVRP instance with finite cost corresponds to a valid CARP solution as far as services on individual tasks are concerned. This transformation gives a CVRP instance with $3r + 1$ vertices, where r is the number of tasks in the given CARP instance. Due to a large number of vertices, this transformation is not very practical.

Baldacci and Maniezzo (2006) and Longo et al. (2006) proposed alternative transformations that give a smaller CVRP instance with only $2r + 1$ vertices. Both transformations are similar in the way a graph for the CVRP is constructed but differ in the edge costs. Given a CARP instance on a graph G, each task $\{i, j\}$ is associated with two vertices s_{ij} and s_{ji}. A complete graph $G_c = (V_c, E_c)$ is constructed with the vertex set

$$V_c = \bigcup_{\{i,j\} \in R} \{s_{ij}, s_{ji}\} \cup \{v_0\}.$$

In the work of Baldacci and Maniezzo (2006), the costs $c(u, v)$ for edges $\{u, v\} \in E_c$ are defined as follows:

$$\begin{aligned} c(s_{ij}, s_{kl}) &= \begin{cases} 0 & \text{if } \{i,j\} = \{k,l\} \\ \frac{1}{2}c_{ij} + D(i,k) + \frac{1}{2}c_{kl} & \text{otherwise} \end{cases} \\ c(v_0, s_{ij}) &= D(v_0, i) + \frac{1}{2}c_{ij}. \end{aligned} \tag{2.19}$$

In contrast, Longo et al. (2006) defined the edge costs as follows:

$$\begin{aligned} c(s_{ij}, s_{kl}) &= \begin{cases} c_{ij} & \text{if } \{i,j\} = \{k,l\} \\ D(i,k) & \text{otherwise} \end{cases} \\ c(v_0, s_{ij}) &= D(v_0, i). \end{aligned} \tag{2.20}$$

However, as far as solution costs are concerned, there is no difference between the above cost structures (2.19) and (2.20): in both cases, the cost of a CVRP solution is equal to the cost of the corresponding CARP solution. For each task $\{i,j\}$, the demands of s_{ij} and s_{ji} are set to $\frac{1}{2}d_{ij}$ (any other positive values are also possible as long as the demands of s_{ij} and s_{ji} add up to d_{ij}). To ensure that a CVRP solution corresponds to a valid CARP solution, additional constraints are introduced to the CVRP: for all tasks $\{i,j\}$, the edge $\{s_{ij}, s_{ji}\} \in E_c$ must be included in the solution.

More recently, Foulds et al. (2015) proposed a transformation that involves a smaller number of vertices in the resulting CVRP ($r+1$ as opposed to $2r+1$ vertices). Given a CARP instance on a graph G, each task $\{i,j\}$ is associated with a single vertex m_{ij}. A complete graph $G_c = (V_c, E_c)$ is constructed with the vertex set

$$V_c = \bigcup_{\{i,j\} \in R} \{m_{ij}\} \cup \{v_0\}.$$

The demand of each vertex $m_{ij} \in V_c$ is equal to the demand of task $\{i,j\}$. A route $(v_0, m_{i_1 j_1}, m_{i_2 j_2}, \ldots, m_{i_n j_n}, v_0)$ in G_c (for some positive integer n) corresponds to a route in G that services tasks $\{i_1, j_1\}, \{i_2, j_2\}, \ldots, \{i_n, j_n\}$ in such an order. To identify the direction in which the tasks are serviced, additional variables a_{ij} are introduced: for each vertex $m_{ij} \in V_c$, $a_{ij} = 0$ denotes servicing the task $\{i,j\} \in R$ from j to i, and $a_{ij} = 1$ denotes the opposite direction (i.e. from i to j). The edge costs on E_c are defined as follows:

$$c(m_{ij}^{a_{ij}}, m_{kl}^{a_{kl}}) = \frac{1}{2}c_{ij} + D\left((1 - a_{ij}) \cdot i + a_{ij} \cdot j, (1 - a_{kl}) \cdot k + a_{kl} \cdot l\right) + \frac{1}{2}c_{kl}. \tag{2.21}$$

Note that the middle term in the right hand side of (2.21) is the shortest distance in G between one endpoint of $\{i,j\}$ and one endpoint of $\{k,l\}$; this depends on the directions in which the tasks are traversed, which are specified by a_{ij}, a_{kl}. Since each task can be serviced in two possible directions, there are 2^r possible cost structures (2.21) of the CVRP, where r is the number of tasks in R. Instead of solving all of those 2^r versions of the CVRP, Foulds et al. (2015) uses a branch-cut-and-price algorithm, in which appropriate values of a_{ij} (i.e. directions of servicing the tasks in the CARP) are determined in the pricing subproblem.

2.9 Variants of the CARP

In addition to the "standard" version of the CARP described in Section 1.1, there have been studies on other variants of the CARP. They are adapted from the standard CARP to reflect practical applications of arc routing more accurately. We now review some of these.

2.9.1 CARP with Time Windows

The CARP with time windows (CARPTW) is the CARP with an additional constraint: the service of each task must begin within a given time interval. Thus, in addition to cost and demand, the amount of time for traversing each edge with and without service are given as part of the problem. Labadi et al. (2008) proposed an algorithm for the CARPTW based on the greedy randomized adaptive search procedure (GRASP). In their GRASP, a solution is generated by first creating a giant tour covering all required edges (ignoring the capacity and time windows for now) and then dividing it into several feasible routes. The solution is then improved by local search (choosing the best move in each iteration). The local search uses three types of moves: Or-Opt, Swap, and 2-Opt, which can operate on one or two routes. Their GRASP is also enhanced by means of path relinking. The idea of path relinking is to explore a certain path in the solution space that links two "high-quality" solutions (in this case, two solutions given by the local search in the GRASP). Such a path depends on a distance measure, i.e. a function which defines a distance between two solutions. By "exploring a certain path between two solutions," it means that we take one of the solutions and iteratively make some changes to it by some neighbourhood move which decreases the distance (based on a given distance measure) between them. They found that, when generating a solution, a better solution can be achieved when using two greedy algorithms together than when using either of them solely. Moreover, path relinking was shown to improve the performance of the GRASP. In fact, using path relinking at different parts of the GRASP could lead to different results: better results on average could be obtained from implementing path relinking "internally" (i.e. after the local search in each iteration of the GRASP) than "externally" (i.e. after the GRASP terminates), although both cases lead to better solutions than not using path relinking. Their algorithm was also tested on benchmarks instances of the CARP (without time windows, or equivalently, with infinite time windows) and was found to be competitive with previously existing algorithms in terms of average deviations from known lower bounds but using noticeably less computational time.

Vansteenwegen et al. (2010) studied the CARP on a mixed graph with soft time windows to find a route for a van that needs to take pictures of all streets and road signs in a given city. Soft time windows arise from the fact that a picture of each street should not be taken in the direction of sunlight as otherwise the pictures may not be usable, so a picture of each street should ideally be taken within a certain period of time. In other words, an earliest beginning time and a latest end time of the service on each task (a street that needs to be taken pictures of) are given. In their particular CARP, a fixed maximum travel time per day is also given. The goal is to minimise a weighted sum of the number of days and the violation of time windows. They solved the CARP with soft time windows by first converting the problem into a vehicle routing problem and solving it by a hybrid metaheuristic: minimising the number of days by variable neighbourhood search and minimising the violations of time windows by iterated local search and linear programming.

2.9.2 CARP with Multiple Starting and/or Ending Vertices

Recall that in the standard CARP, it is assumed that there is only one depot and all vehicles must start and end at the depot.

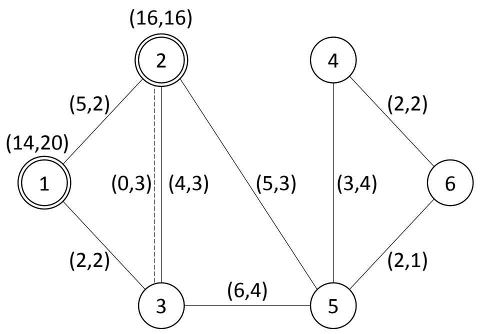
CARP with multiple depots Amberg et al. (2000) studied the CARP with multiple depots, also called M-CARP in their original paper, where different vehicles can have different depots (each vehicle must start and end at their designated depot). Their approach involves transforming the M-CARP into the capacitated minimum spanning tree problem (CMSTP). The transformation is illustrated by Figure 2.5, reproduced from Amberg et al. (2000). It should be noted that the type of CARP in their study involves an additional constraint: a maximum limit on the cost of each route. Their approach begins by augmenting a graph G given in the M-CARP (i.e. duplicating some edges) to obtain an Eulerian graph (Figure 2.5(a)), which is then decomposed into several cycles (Figure 2.5(b)). A new graph $G' = (V', E')$ for the CMSTP is constructed (Figure 2.5(c)) such that the number of vertices in V' is equal to the number of depots plus the number of cycles in the decomposition (so each vertex in V' represents either a depot or a cycle in the decomposition), and E' contains edges joining any two vertices in V' except when both vertices correspond to depots. Each edge $\{i, j\}$ in E' represents a cycle formed by two copies of a shortest path between a depot or a cycle represented by $i, j \in V'$. For example, vertex $3'$ and $5'$ in Figure 2.5 represent the cycles 1–2–3–1 and 4–5–6–4, respectively, and a shortest path between these two cycles is 2–5 (with cost 3), so edge $\{3', 5'\}$ in G'

represents a cycle 2–5–2 in G. After the transformation, a M-CARP solution in G is obtained by finding minimum spanning trees in G' such that the number of trees is equal to the number of depots, each tree contains exactly one depot, and the total demand (resp. cost) in each tree does not exceed the capacity (resp. maximum cost) of the depot in that tree. Without loss of generality, it is assumed that the number of depots is equal to the number of vehicles (otherwise, any depot with more than one vehicle is duplicated for a sufficient number of times so that the assumption is true). By doing so, each tree in the CMSTP represents one route in the M-CARP.

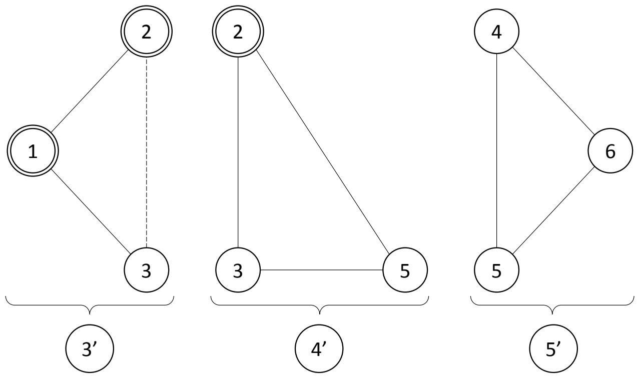
Open CARP In some cases, vehicles do not need to return to the depot after completion of service, resulting in *open* routes (Fung et al., 2013). For example, a service provider may hire vehicles from an external company and require them to start their journeys from a designated location, e.g. vehicle drivers may need to receive orders from a main office of the service provider. However, the vehicles do not need to return to the main office after finishing their service; rather, they should return to the external company that owns the vehicles. In this case, the service provider wishes to minimise the total cost incurred as a result of the journey of each vehicle from the main office up to its last task. Note that there is no restriction on the ending vertex of each vehicle, i.e. their routes can end at any vertices.

Even though a solution to the open CARP can be obtained from a solution to the CARP by removing a path from the last task to the depot in each route, it is not guaranteed that an optimal CARP solution will give an optimal open CARP solution via this process; this is illustrated by Figure 2.6, adapted from Fung et al. (2013). This means that rather than relying on solution methods for the CARP, there is a need to develop an algorithm specifically for the open CARP.

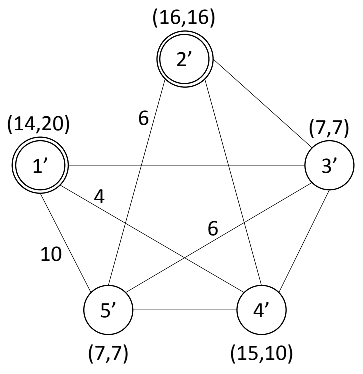
Fung et al. (2013) attempted to solve the open CARP by transforming the open CARP into an open vehicle routing problem (VRP) and subsequently into a closed VRP. It should be noted that the open CARP considered in their study is based on a directed graph, so each task in the CARP corresponds to one vertex in the VRP (as opposed to two vertices in the case of an undirected graph). To solve the resulting closed VRP, they proposed a memetic algorithm. In their algorithm, a chromosome is a sequence of all tasks (which are now represented by vertices in the VRP) without trip delimiters. To evaluate a chromosome, the split algorithm (Prins, 2004) is used to convert the chromosome into feasible routes, and the total cost of those routes is then regarded as the fitness of the chromosome. The split algorithm is similar to Ulusoy's partitioning described in Section 2.5, the difference being that



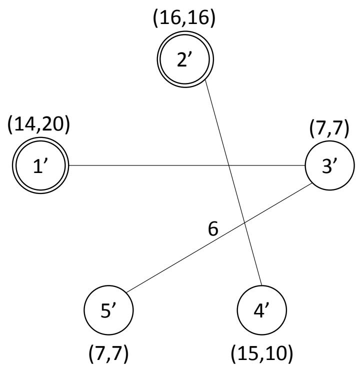
(a) M-CARP instance converted into an Eulerian graph with an augmenting edge shown in dash; depots are represented by double-lined vertices; the pair (Q, L) on each depot denotes the capacity Q and the maximum route cost L; the pair (d, c) on each edge denotes its demand d and cost c



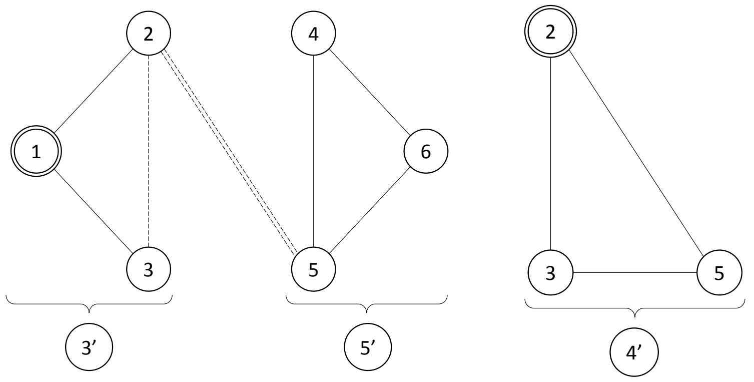
(b) Cycle decomposition of the Eulerian graph in (a)



(c) CMSTP instance derived from (a) and (b); the number next to each edge shows its cost (with zero cost omitted); the pair (Q, L) next to each depot shows its capacity Q and maximum cost L; the pair (d, c) next to each vertex shows its demand d and cost c

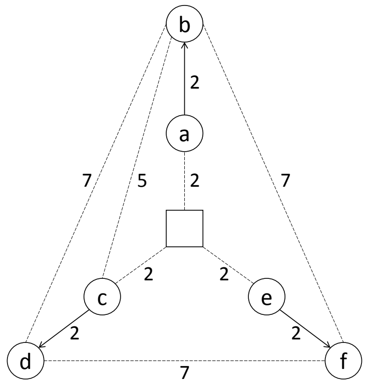


(d) A feasible solution to the CMSTP in (c)

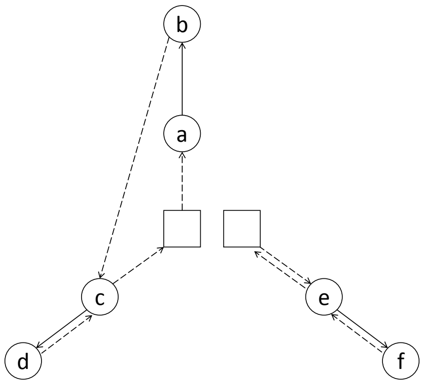


(e) M-CARP solution derived from the CMSTP solution in (d); solid lines represent traversals with service and dashed lines without service; the depot of each route is represented by a double-lined vertex

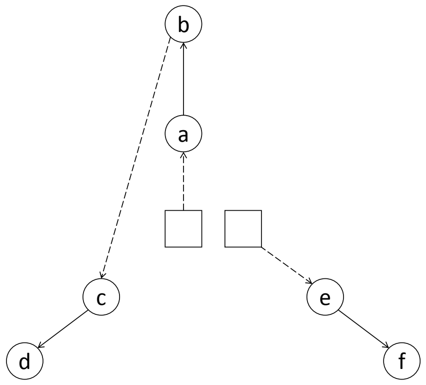
Figure 2.5: Transformation from M-CARP to CMSTP



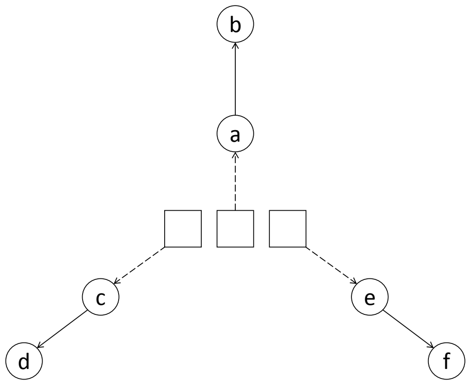
(a) Graph with edge costs; each task (solid line) must be serviced in a specified direction



(b) Optimal CARP solution



(c) OCARP solution obtained by removing a path after the last task in each route in (b) (cost = 15)



(d) Optimal OCARP solution (cost = 12)

Figure 2.6: Difference between optimal solutions for the CARP and the OCARP on the same instance; the depot is represented by a square; each task ab, cd, and ef has demand 1 and the capacity is 2

it takes a sequence of vertices (as opposed to a sequence of arcs) as an input. An initial population consists of both chromosomes generated by constructive heuristics and those generated randomly. To promote diversity in the population, having two chromosomes with the same fitness is not allowed. To produce new chromosomes, two parent chromosomes are selected by means of tournament selection, i.e. at least 2 chromosomes are selected randomly and one of them is kept as a parent chromosome. The process is repeated for the other parent chromosome. The parent chromosomes then undergo the Order Crossover (OX) (described in Section 2.6.2) to produce two offspring chromosomes. Then, one of the offspring chromosomes is randomly selected (and the other discarded). With a given probability, the selected offspring chromosome undergoes a local search procedure involving 2-opt, 1-1 swap, and single insertion moves (the last two types of moves are also called 1-1 exchange and 1-0 exchange moves in their original papers). Before the local search procedure, the split algorithm is used to convert the offspring chromosome into feasible routes. A solution given by the local search procedure is converted back into a chromosome by concatenating all routes. The split algorithm is then applied to this chromosome to determine its fitness, and an evaluation counter (which will be used to determine when to terminate the algorithm) increases by 1. If the offspring chromosome has better fitness than the worst one in the current population, one of the chromosomes in the worse half of the population is selected randomly to be replaced by the offspring chromosome. However, if this results in there being two chromosomes with the same fitness, the offspring chromosome is discarded. This process is repeated until the evaluation counter reaches a given maximum limit.

Open CARP without the depot There is an even more general open CARP, in which there is no depot, so vehicles can start and end their journeys at any of the vertices. Notice that this version of open CARP is trivial if the number of vehicles is a decision variable: an optimal solution can be obtained by using as many vehicles as tasks, letting each vehicle service one task, starting from one endpoint of the task and ending at the other endpoint (so the total distance is simply the sum of distances of all tasks). Thus, we can restrict our attention to non-trivial cases, where the number of vehicles is fixed and less than the number of tasks. For those cases, Usberti et al. (2011) proposed a constructive heuristic called the *reactive path scanning heuristic with ellipse rule*. It was adapted from the path scanning heuristic with ellipse rule proposed by Santos et al. (2009) (see Section 2.5), which tends to give an infeasible solution for this type of open CARP because it has no mechanism for limiting the number of vehicles and hence it tends to use more vehicles than there are available. Recall that the path scanning heuristic constructs one route at a time and each route is constructed by iteratively adding to the route a task that is nearest to the current endpoint of the route. To increase the likelihood of obeying the given number of vehicles, they amend the criteria for choosing the next task in order to bias towards tasks that are near the current endpoint *and* have large demands; this was motivated by the first fit decreasing heuristic for the bin packing problem. More precisely, let v_e be the current endpoint of the route, $\mathrm{SP}^{\max}$ the distance of the longest shortest path between any two vertices, and $d^{\max}$ the maximum demand, the next task to be added to the route is chosen from a set of tasks (i, j) (with demand d_{ij}) that minimises

$$\gamma \frac{D(v_e, i)}{\mathrm{SP}^{\max}} + (1-\gamma)\left(1 - \frac{d_{ij}}{d^{\max}}\right), \tag{2.22}$$

for some parameter $\gamma \in [0,1]$. Notice that $\gamma = 1$ gives the original criteria (i.e. restricting candidates tasks to those nearest to the current endpoint). The construction is repeated for a given number of times with γ initially set to 1 and later adapted depending on the feasibility of a solution: if the construction gives an infeasible solution, γ is decreased so that the algorithm focusses less on minimising the distance and more on the bin-packing nature of the problem, i.e. trying to use relatively few vehicles. Furthermore, the parameter β that controls when to use the ellipse rule is chosen randomly from a discrete set $\{0.0, 0.5, 1.0, 1.5, 2.0\}$ because it was found that no single value of β worked well for all instances (recall that the ellipse rule is used when the remaining capacity of the vehicle is less than or equal to $\beta \times \frac{\text{total demand}}{\text{number of tasks}}$). The probability of choosing a particular value of β is also amended after each construction: it increases when it gives a feasible solution and decreases otherwise.

Even though the open CARP can be viewed as the CARP without the depot constraint, it is still NP-hard: Usberti et al. (2011) proved that the CARP, which is NP-hard, can be reduced polynomially into the open CARP. The main idea in their proof for transforming the CARP into the open CARP is that, given the number of vehicles n_{veh}, introduce $2n_{\text{veh}}$ dummy vertices and $2n_{\text{veh}}$ dummy edges, each of which connects one dummy vertex to the depot and has some demand $\delta > 0$, and increase the capacity of each vehicles by 2δ. Each route in any feasible solution to this open CARP contains 2 dummy edges, so a solution to the CARP can be obtained by removing those dummy edges from the open CARP solution.

2.10 Dynamic CARPs

A dynamic CARP is the CARP in which some information changes over time, especially while vehicles are travelling around the graph. In this section, we review types of changes in dynamic CARPs that have been studied in the literature.

Dynamic Graph There are situations where the vertex set and/or the edge set of an underlying graph can change while vehicles are travelling. One example, often found in industry, is a problem of determining a minimum-distance path for cutting a plate into specified shapes (assuming that the shapes have already been arranged in the plate). This problem can be viewed as a dynamic rural postman problem (Moreira et al., 2007), where a graph $G = (V, E)$ represents possible movements of a cutter (equivalent to a "vehicle") within the plate. Assuming all the shapes are polygons, the vertex set V contains the polygons' vertices, and the edge set E contains the polygons' edges and straight lines between two vertices in V that do not pass through any polygons; this ensures that the specified shapes are not cut into smaller pieces. The dynamism arises from the fact that when part of the plate is completely cut out and falls to a container underneath, there is more empty space in which the cutter can move. Moreira et al. (2007) proposed a constructive heuristic which constructs a route by extending it with one edge at a time. It should be noted that an edge is forbidden if it cuts out a piece of the plate that contains smaller shapes also needing to be cut out. After each extension, the heuristic checks whether the edge leads to a shape being completely cut out (i.e. all of its edges have been traversed); if so, the graph is updated to reflect current possible movement of the cutter.

Another example of a dynamic CARP with a changing graph is when vehicles encounter blockages that are unknown when planning routes initially. In other words, some edges in the graph disappear, and so any route that involves such edges need to be amended. Yazici et al. (2014) proposed a constructive heuristic for updating a solution whenever a blockage is encountered. This constructive heuristic was obtained by adapting Ulusoy's Partitioning (described in section 2.5) for the CARP with heterogeneous vehicles (i.e. having different capacities) and multiple starting vertices.

Changes of Edge Costs Tagmouti et al. (2011) studied a dynamic CARP with the focus on winter gritting, where edge costs change according to weather report updates, which arrive at given regular time intervals. Notice that the changes do not make the solution infeasible but can affect the quality of the solution, for example, the solution cost could increase if the cost of some edge contained in the solution increases. A solution update is needed to maintain or possibly improve the quality of the solution after the changes. In their study, a solution is updated by a variable neighbourhood descent algorithm whenever a weather report update is received.

Broken-Down Vehicles When a vehicle is broken down, tasks that are currently assigned to that vehicle but have not yet been serviced must be reassigned to other vehicles. Referring to this problem as the *rescheduling arc routing problem*, Monroy-Licht et al. (2016) attempted to tackle this type of changes with the assumptions that initial routes are given, vehicles are uncapacitated (so tasks from broken-down vehicles can be reassigned to any other vehicle without restriction), and there are no extra vehicles. Their main goal is to ensure that all tasks are serviced, while minimising one of the following objective functions: the total distance, the so-called "disruption cost," which measures similarity between the original set of routes and a new one, and a combination of the previous two objectives. Using exact algorithms, they found that the total distance and the disruption cost could not be minimised simultaneously, whereas a combination of these two objectives could provide a trade-off between them. It should be noted that their study is restricted to a single "breakdown time," that is, only one solution update is needed, and the update occurs whenever the breakdown occurs.

Multiple Types of Changes The aforementioned studies concern dynamic CARP with one type of change. To the best of our knowledge, the study of Liu et al. (2014a,b) is the only existing study that attempts to tackle a dynamic CARP with multiple types of changes. More precisely, they considered broken-down vehicles, unavailability of edges (road blockages), changes of edge costs, and changes of edge demands. In particular, the changes of edge demands being considered include a slight change in the amount of demand on an existing task (demand changing from one positive integer to another), an appearance of a new task (demand changing from zero to a positive integer), and a disappearance of an existing task (demand changing from a positive integer to zero). In their study, a solution is updated by a memetic algorithm whenever changes occur.

Notice that in almost all of the aforementioned studies on dynamic CARPs (the only exception being Moreira et al., 2007), solutions are updated as soon as a change occurs. However, for some types of changes such as the appearance of new tasks, a solution need not be updated immediately. In fact, it might be possible that updating a solution at different times would lead to different solution qualities. To the best of our knowledge, this has not been investigated in the context of dynamic CARPs.

2.11 Summary

In this chapter, the CARP and other related routing problems were reviewed. Although some routing problems can be solved efficiently, it was found that the CARP is NP-hard, and thus it can take an infeasible amount of time to find an optimal solution. In practice, a reasonably good solution is usually acceptable as long as it can be obtained within a relatively small amount of time. This can be achieved by means of heuristic algorithms, many of which have been proposed for the CARP in the literature. These include constructive heuristics and metaheuristic algorithms. Several techniques for computing lower bounds have also been proposed; this helps estimate how close a heuristic solution is to optimality and can also be used to prove optimality of some solutions (when a lower bound is equal to the cost of a solution). Moreover, there exists some work on developing exact algorithms for the CARP, although computational results suggest that they require a substantial amount of time to find optimal solutions in some instances.

It was noticed that most of the existing work on the CARP deal with the static version of the problem, that is, it is assumed that all information of the problem is known and does not change after the vehicles depart from the depot. In real life, however, some information of the problem may change, which may lower the quality of the original solution or render it infeasible, hence the need to update the solution. With current communication technology, a route planner can track the current state of each vehicle and inform drivers of changes in their routes in a short time, if not immediately. A review of literature revealed that there has been relatively little work on a dynamic version of the CARP. This motivates further study in this area. Furthermore, computational results in existing work suggests that heuristic algorithms would be preferable for tackling dynamic CARP due to its ability to provide feasible solutions quickly compared with exact algorithms. This motivates the idea of developing heuristic algorithms for dynamic CARP.

By comparing existing metaheuristic algorithms for the CARP in the literature, it was found that single-solution-based metaheuristis algorithms, namely guided local search and tabu search, can achieve fairly good solutions within a small amount of computation time. In the next chapter, we opt to study how tabu search performs over a range of iterations; this metaheuristic algorithm requires relatively few parameters, which allows us to concentrate our attention when attempting to identify factors that encourage the algorithm to improve a solution quickly. A variant of a tabu search algorithm that improves a solution relatively quickly will then be selected to tackle the dynamic CARP in subsequent chapters.

Chapter 3

Metaheuristic Algorithms for the Static CARP

3.1 Introduction

Metaheuristic algorithms have been shown to be able to achieve near-optimal or even optimal solutions for the CARP. However, their performances were commonly reported based on certain stopping criteria such as a given number of iterations or a given number of consecutive iterations without improvement. This provides little, if any, information about their performances over the course of their execution, hence little information about what makes an algorithm find a good solution quickly. This information would be useful when designing an algorithm for a dynamic CARP, where a solution is updated while tasks are being serviced. Among algorithms that can achieve equally good solutions, it would be more preferable to use an algorithm that can improve a solution significantly within a few iterations than one that can eventually find an optimal solution but only improves a solution at a very slow rate.

In this chapter, the aim is to gain more insights about what would make an algorithm find a good CARP solution quickly. In particular, several variants of a tabu search algorithm will be compared based on a wide range of stopping criteria. Tabu search is a metaheuristic that requires a relatively small number of parameters; this allows us to concentrate our attention on certain aspects of the algorithm when investigating what would make an algorithm find a good solution quickly. A description of how tabu search works in general can be found in Section 2.6.1.

In addition, a novel operator for improving the tabu search algorithm is proposed; in fact, this operator can also be used not only with tabu search but also with other metaheuristic algorithms. This operator, called the deadheading cycle remover, works by detecting and removing unnecessary traversals that form so-called deadheading cycles in a given solution. It is intended to be a quick operator that helps tabu search improve a solution faster without too much additional computational effort.

Sections 3.2 and 3.3 describe components of the tabu search algorithm that will be investigated, i.e. neighbourhood moves and definitions of tabu moves, respectively. Variants of the tabu search algorithm with different ways of defining tabu moves will be compared and analysed in Section 3.4. Section 3.5 proposes a novel operator, namely the deadheading cycle remover and analyses its effect to the performance of the tabu search algorithm. Section 3.6 discusses a possible pitfall of using different tabu lists for different neighbourhood moves. The conclusions of this chapter are given in Section 3.7.

3.2 Neighbourhood Moves

Neighbourhood moves are ways of making small changes to a given solution in the tabu search methodology (as well as other types of local search in general). Recall that a CARP solution is a set of routes, and a route can be represented by a sequence of tasks with specified directions (preceded by its starting vertex and followed by the depot v_0). Let $R_1 = (v_{s_1}, a_1, \ldots, a_{n_1}, v_0)$ and $R_2 = (v_{s_2}, b_1, \ldots, b_{n_2}, v_0)$ be two routes, for some tasks $a_1, \ldots, a_{n_1}, b_1, \ldots, b_{n_2}$, some vertices v_{s_1}, v_{s_2}, and some positive integers n_1, n_2. For ease of notation, a_{n_1+1} and b_{n_2+1} are identified with the depot v_0 at the end of the routes. The tabu search algorithm that will be considered in this chapter uses the following neighbourhood moves:

- Single Insertion: remove a task from one route and insert it in another route. Given a removal index i $(1 \leq i \leq n_1)$ and an insertion index j $(1 \leq j \leq n_2 + 1)$, remove a_i from R_1 and insert it in front of b_j in R_2.
- Double Insertion: remove two tasks from one route and insert them in another route. Given two removal indices i_1, i_2 $(1 \leq i_1 < i_2 \leq n + 1)$ and two insertion indices j_1, j_2 $(1 \leq j_1, j_2 \leq n_2 + 1)$, remove a_{i_1} and a_{i_2} from R_1 and insert them in R_2 in front of b_{j_1} and b_{j_2}, respectively. If $j_1 = j_2$, i.e. a_{i_1}, a_{i_2} are inserted at the same position, both possible orders (a_{i_1} followed by a_{i_2} or vice versa) are considered.

- Swap: swap tasks between two routes (one task from each route). Given two removal indices i, j ($1 \leq i \leq n_1, 1 \leq j \leq n_2$) and two insertion indices i', j' ($1 \leq i' \leq n_1 + 1, 1 \leq j' \leq n_2 + 1$) such that $i \neq i'$ and $j \neq j'$, remove a_i from R_1 and place it in front of $b_{j'}$ in R_2; remove b_j from R_2 and place it in front of $a_{i'}$ in R_1.

- 2-Opt: cut two routes each into two subroutes and re-connect subroutes to obtain two new routes. Given two cutting indices i, j ($1 \leq i \leq n_1 + 1, 1 \leq j \leq n_2 + 1$), cut R_1 into two parts $(v_s, a_1, \ldots, a_{i-1})$ and $(a_i, \ldots, a_{n_1}, v_0)$[1] and cut R_2 in a similar fashion. Then, join one part of R_1 with one part of R_2 (and join the remaining parts together). Note that there are two possible ways to join the parts, as illustrated in Figure 3.1.

For Single Insertion, Double Insertion, and Swap, when inserting a task into a route, both directions of traversal on that task are tested.

These four types of neighbourhood moves have been used in the literature (see Section 2.6.1), although some of them are implemented in this thesis in a slightly different way: for Double Insertion, removed tasks do not need to be consecutive (i.e. i_2 is not necessarily equal to $i_1 + 1$) and they do not need to be inserted at the same position (i.e. j_2 is not necessarily equal to j_1). For 2-opt, it is allowed that all tasks in one route (e.g. when $i = 1$) is moved to another route. To avoid unnecessary computation, 2-opt moves that has essentially no effect to the solution are omitted (e.g. $i = j = 1$ with a particular way of joining routes simply results in renumbering routes). The 2-opt moves that resemble other neighbourhood moves are also omitted (e.g. $i = n_1$ and $j = n_2$ with a particular way of joining routes resembles Swap).

A neighbourhood move is said to be *admissible* if either it is non-tabu or it is tabu but leads to the solution that is better that the current best solution. In each iteration of the tabu search algorithm, all neighbourhood moves that are both admissible and feasible with respect to the capacity constraint are considered. Among those moves, the best move (i.e. it leads to the lowest total distance) is selected and applied to the current solution; if there is more than one best move, one of them is selected randomly. The next section describes how the tabu status of neighbourhood moves are determined.

[1]If $i = 1$, then the route R_1 is divided into (v_s) and $(a_1, \ldots, v_0)$. If $i = n_1 + 1$, then the route is divided into $(v_s, \ldots, a_{n_1})$ and (v_0).

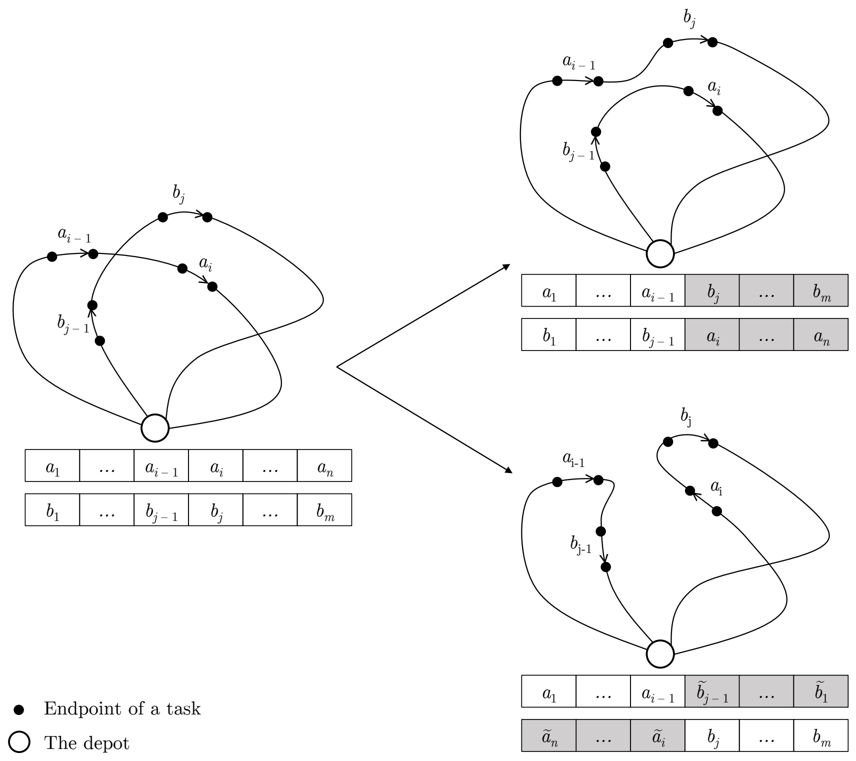

a_{i-1}
b_j
a_i
b_{j-1}
a_1 … a_{i-1} a_i … a_n
b_1 … b_{j-1} b_j … b_m
a_1 … a_{i-1} b_j … b_m
b_1 … b_{j-1} a_i … a_n
a_1 … a_{i-1} $\tilde{b}_{j-1}$ … $\tilde{b}_1$
$\tilde{a}_n$ … $\tilde{a}_i$ b_j … b_m
Endpoint of a task
The depot

Figure 3.1: Two possible ways of joining parts of routes as a result of a 2-opt move; tasks that are removed from their original routes are highlighted; $\tilde{a}$ denotes the opposite direction of traversal on task a

3.3 Solution Attributes and Tabu Moves

One key idea of tabu search is to avoid getting stuck at local optima by forbidding neighbourhood moves that lead to previously visited solutions. In theory, this can be achieved by recording complete descriptions of all solutions that have been visited. However, checking the whole solutions can be very inefficient as more and more solutions are visited and recorded. An alternative approach is to consider solution attributes, i.e. partial descriptions of solutions, in which case a move is regarded as *tabu* if it leads to a solution that contains certain solution attributes. Types of solution attributes that have been considered in existing tabu search algorithms for arc routing and vehicle routing include:

- *Task-in-a-route* attributes. This concerns the fact that a task is serviced in a certain route. When a task is removed from a route, it is prohibited from returning to that route for a given number of iterations. This type of solution attributes was used by Hertz et al. (2000) and Greistorfer (2003). A similar idea was also used by Chiang and Russell (1997), Ho and Haugland (2004), and Brandão (2009) for vehicle routing, i.e. customers (demands on vertices) are considered instead of tasks.
- *2-customer* attributes. This concerns an edge joining two customers in the same route for vehicle routing, or equivalently, the fact that two customers are serviced consecutively in the same route. When an edge is removed, it is prohibited from re-appearing for a given number of iterations. This type of solution attributes was used by Ho and Haugland (2004), Montané and Galvao (2006), and Holborn et al. (2012).

In our tabu search algorithm, after the best neighbourhood move in each iteration is selected, some solution attributes corresponding to the move are recorded in a so-called tabu list (which is initially empty). Each solution attribute that is recorded in the tabu list remains in the list for some iterations, the number of which is to be specified; this number is called a *tabu tenure*. To determine whether each neighbourhood move is tabu, solution attributes that would arise as a result of the move are checked, and the move is regarded as tabu if all of those attributes are currently in the tabu list. There are a variety of solution attributes that can be considered. Here, we will consider and compare four types of solution attributes as follows:

- *Task-in-a-route* attributes.

- *2-task* attributes. This is similar to the 2-customer attributes described above but tasks are considered instead of customers. In other words, this concerns the fact that two tasks are serviced consecutively in the same route.
- *2-task-in-a-route* attributes. This new attribute combines the concepts of the above two attributes. In other words, it concerns the fact that two tasks are serviced consecutively in a certain route.
- *3-task* attributes. This new attribute concerns the fact that three tasks are serviced consecutively in the same route.

Notice that 2-task-in-a-route attributes contain more information than task-in-a-route and 2-task attributes, and 3-task attributes contain more information than 2-task attributes. A type of attribute that contains more information is expected to make fewer moves tabu, and so the corresponding tabu search algorithm is expected to be less restrictive (i.e. more admissible moves in each iteration). It will be investigated whether this necessarily leads to a better tabu search algorithm. Each type of solution attribute will be used separately (unless otherwise stated), giving four different variants of the tabu search algorithm. For clarity, the rest of this section describes what exactly is recorded in the tabu list and how to determine whether each neighbourhood move is tabu based on each type of solution attributes.

3.3.1 Task-in-a-Route Attributes

Recording solution attributes: When a task t is removed from a route R by either a Single Insertion, Double Insertion, or Swap move, a pair (t, R) is recorded in the tabu list. Notice that a Single Insertion move corresponds to one pair, whereas a Double Insertion or Swap move corresponds to two pairs.

For 2-opt, notice that this type of move generally transfers many tasks between routes (especially when a route with many tasks is cut near the middle of the route), so it is expected that determining whether a 2-opt move is tabu based on the task-in-a-route attributes would involve recording and checking a large number of pairs. For this reason, we opt to use a separate tabu list for 2-opt moves based on 2-task attributes (see Section 3.3.2).

Determining tabu status of moves: A Single Insertion move that inserts task t into route R is tabu if the pair (t, R) is currently in the tabu list. A Double Insertion or Swap move that inserts task t_1 into route R_1 and task t_2 into route R_2 ($R_2 = R_1$ for Double Insertion) is tabu if both pairs $(t_1, R_1), (t_2, R_2)$ are currently in the tabu list.

3.3.2 2-Task Attributes

Recording solution attributes: Let $R = (v_s, a_1, \ldots, a_n, v_0)$ be a route. When task a_i is removed from route R by either a Single Insertion, Double Insertion, or Swap move, two pairs (a_{i-1}, a_i) and (a_i, a_{i+1}) are recorded in the tabu list (that is, each pair of consecutive tasks involving the move task is recorded). For ease of notation, let $a_0 = v_s$ (the starting vertex) and $a_{n+1} = v_0$ (the depot at the end of the route).

For 2-opt, the pair of tasks next to the cutting position in each route is recorded in the tabu list. More precisely, if a 2-opt move cuts a route $R = (v_s, a_1, \ldots, a_n, v_0)$ into two parts $(v_s, a_1, \ldots, a_{i-1})$ and $(a_i, \ldots, a_n, v_0)$, then a pair (a_{i-1}, a_i) is recorded.

Let $\tilde{a}$ denote the opposite direction of traversal on task a. Due to symmetry of routes in an undirected graph, for any tasks a and b, the pair $(\tilde{a}, \tilde{b})$ is regarded as identical to the pair (b, a).

Determining tabu status of moves: A Single Insertion move that inserts task t into a route $(v_s, b_1, \ldots, b_n, v_0)$ between tasks b_{j-1} and b_j is tabu if both pairs (b_{j-1}, t) and (t, b_j) are currently in the tabu list. For a Double Insertion or Swap move, two pairs corresponding to each inserted task are checked, and the move is tabu if all the pairs are currently in the tabu list.

A 2-opt move is tabu if the pairs of tasks next to the joining positions in both routes are currently in the tabu list. More precisely, consider a 2-opt move that modifies two routes $(v_{s_1}, a_1, \ldots, a_{n_1}, v_0)$ and $(v_{s_2}, b_1, \ldots, b_{n_2}, v_0)$ and gives two new routes, namely $(v_{s_1}, \ldots, a_{i-1}, b_j, \ldots, v_0)$ and $(v_{s_2}, \ldots, b_{j-1}, a_i, \ldots, v_0)$. This 2-opt move is tabu if both (a_{i-1}, b_j) and (b_{j-1}, a_i) are currently in the tabu list.

3.3.3 2-Task-in-a-Route Attributes

Recording solution attributes: The procedure is the same as the 2-task attributes except that an appropriate route is also recorded. In other words, for any pair of tasks (a, b) that would be recorded in the case of the 2-task attributes, we instead record a triple (a, b, R), where R is the route to which both tasks belong together before the solution is amended by a selected move.

Determining tabu status of moves: A Single Insertion move that inserts task t into a route $R = (v_s, b_1, \dots, b_n, v_0)$ between tasks b_{j-1} and b_j is tabu if both triples (b_{j-1}, t, R) and (t, b_j, R) are currently in the tabu list. For a Double Insertion or Swap move, two triples corresponding to each inserted task are checked, and the move is tabu if all the pairs are currently in the tabu list.

A 2-opt move is tabu if the triples corresponding to tasks next to the joining positions in both routes are currently in the tabu list. More precisely, consider a 2-opt move that would amend routes $R_1 = (v_{s_1}, a_1, \dots, a_{n_1}, v_0)$ and $R_2 = (v_{s_2}, b_1, \dots, b_{n_2}, v_0)$ and give $R_1 = (v_{s_1}, \dots, a_{i-1}, b_j, \dots, v_0)$ and $R_2 = (v_{s_2}, \dots, b_{j-1}, a_i, \dots, v_0)$. This 2-opt move is tabu if both (a_{i-1}, b_j, R_1) and (b_{j-1}, a_i, R_2) are currently in the tabu list.

3.3.4 3-Task Attributes

Recording solution attributes: Let $R = (v_s, a_1, \dots, a_n, v_0)$ be a route. When task a_i is removed from route R by either a Single Insertion, Double Insertion, or Swap move, the triple (a_{i-1}, a_i, a_{i+1}) is recorded in the tabu list. As before, let $a_0 = v_s$ and $a_{n+1} = v_0$ for ease of notation.

For 2-opt, each task next to the cutting point in each route is regarded as a middle task in a triple, and a similar procedure for recoding attributes applies. More precisely, if a 2-opt move cuts a route $R = (v_s, a_1, \dots, a_n, v_0)$ into two parts $(v_s, a_1, \dots, a_{i-1})$ and $(a_i, \dots, a_n, v_0)$, then the triples (a_{i-2}, a_{i-1}, a_i) and (a_{i-1}, a_i, a_{i+1}) are recorded in the tabu list. If $i = 1$, a_{i-2} is undefined and so the triple (a_{i-2}, a_{i-1}, a_i) is omitted. The other exception case (i.e. when $i = n + 1$) is dealt with in the same way.

Due to symmetry of routes in an undirected graph, for any tasks a, b, and c, the triple $(\tilde{a}, \tilde{b}, \tilde{c})$ is regarded as identical to the triple (c, b, a).

Determining tabu status of moves: A Single Insertion move that inserts task t into a route $(v_s, b_1, \dots, b_n, v_0)$ between tasks b_{j-1} and b_j is tabu if the triple (b_{j-1}, t, b_j) is currently in the tabu list. For a Double Insertion or Swap move, the triple corresponding to each inserted task is checked, and the move is tabu if both the triples are currently in the tabu list.

A 2-opt move is tabu if all "new" sequences of 3 consecutive tasks that arise as a result of the move are currently in the tabu list ("new" here means that the sequences are not part of the solution before the move). More precisely, consider a 2-opt move that would amend routes $R_1 = (v_{s_1}, a_1, \ldots, a_{n_1}, v_0)$ and $R_2 = (v_{s_2}, b_1, \ldots, b_{n_2}, v_0)$ and give $R_1 = (v_{s_1}, \ldots, a_{i-1}, b_j, \ldots, v_0)$ and $R_2 = (v_{s_2}, \ldots, b_{j-1}, a_i, \ldots, v_0)$. This 2-opt move is tabu if the following triples are currently in the tabu list: (a_{i-2}, a_{i-1}, b_j), (a_{i-1}, b_j, b_{j+1}), (b_{j-2}, b_{j-1}, a_i), and (b_{j-1}, a_i, a_{i+1}).

3.4 Comparison of Tabu Attributes

In this section, variants of the tabu search algorithm with different types of tabu attributes (described in Section 3.3) are compared. All variants construct an initial solution by the Path Scanning algorithm (see Section 2.5) and use the neighbourhood moves described in Section 3.2. Each variant is run for a maximum of $100n_t$ iterations, where n_t is the number of tasks. Due to the stochastic nature of the algorithm, its performance on a given instance is assessed based on average results over 20 runs. The instances used here are taken from the BMCV dataset (Beullens et al., 2003), and their details are shown in Table 3.1 (these instances are taken from `http://logistik.bwl.uni-mainz.de/benchmarks.php`, last accessed 29 September 2017). The tabu search algorithm was coded using C++, and the experiment was performed on an Intel Core i3-2120 3.30GHz CPU with 8GB RAM.

The performance of the tabu search algorithm is measured based on percentage deviations from optimality, i.e.

$$\text{percentage deviation} = \left(\frac{\text{solution cost} - \text{optimal cost}}{\text{optimal cost}} \right) \times 100. \tag{3.1}$$

Notice that an optimal solution (a set of routes) on a given instance remains optimal after multiplying each edge cost by a constant since all solution costs are scaled up or down by the same constant factor. The percentage deviation is invariant under such multiplication and therefore helps avoid bias towards results on a particular instance, especially when relative sizes of unit costs on different instances are unclear.

As different types of tabu attributes may work well with different tabu tenures, each of them is tested in preliminary experiments with a range of tabu tenures $l \times n_t$ for $l = 0.25, 0.5, 1, 2, 4, 8, 16, 32$. The results (shown in Figures B.1 to B.4 in Appendix B)

Table 3.1: Characteristics of instances from the BMCV dataset in an ascending order of the number of tasks; n_{veh} is the least number of vehicles needed (total demand divided by capacity, rounded up to the nearest integer)

Instance	Number of vertices	Number of edges	Number of tasks	Optimal cost	n_{veh}
E25	26	35	28	1615	4
C16	32	42	32	1475	3
E17	38	50	36	2740	5
C25	37	50	38	2310	5
C17	43	56	42	3555	7
C22	56	76	43	2245	4
E06	49	66	43	2055	5
E22	54	73	44	2470	5
E03	46	61	47	2015	5
E10	56	76	49	3605	7
E07	73	94	50	4155	8
C03	46	64	51	2575	6
C06	38	55	51	2535	6
C07	54	70	52	4075	8
C13	40	60	52	2955	7
E13	49	73	52	3345	7
C02	48	66	53	3135	7
C20	45	64	53	2120	5
E16	60	80	54	3775	7
C10	60	82	55	4700	9
E14	53	72	55	4115	8
C14	58	79	57	4030	8
E02	58	81	58	3990	8
E08	74	98	59	4710	9
C19	62	84	61	3115	6
E05	68	94	61	4585	9
C08	66	88	63	4090	8
E20	56	80	63	2825	7
C05	56	79	65	5365	10
E19	77	103	66	3235	6

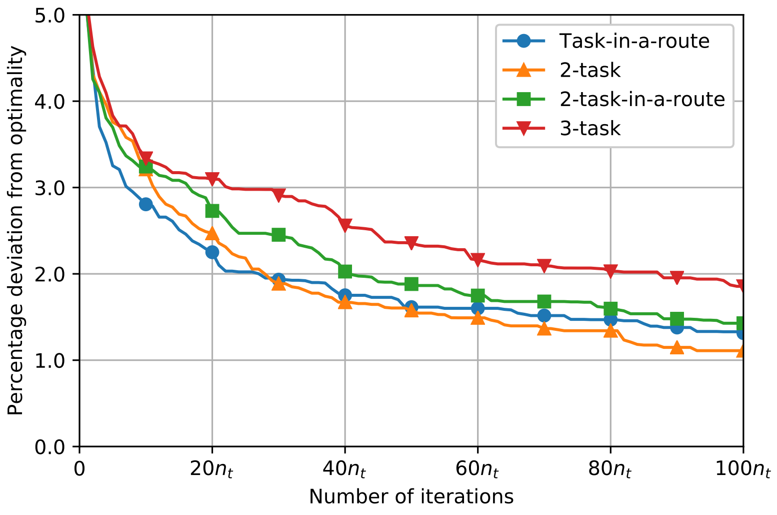


Figure 3.2: Medians of percentage deviations from optimality over the set of instances in Table 3.1 given by the tabu search algorithm corresponding to different types of tabu attributes; n_t denotes the number of tasks

suggested that the tabu search algorithm performs relatively well with the following tabu tenures: $2n_t$ for the task-in-a-route attributes, $16n_t$ for the 2-task attributes, $32n_t$ for the 2-task-in-a-route attributes, and $32n_t$ for the 3-task attributes. The solution attributes will be compared based on these tabu tenures.

Figure 3.2 shows medians of percentage deviations from optimality observed over the course of the tabu search algorithm with different types of tabu attributes (the percentage deviations on each instance are averaged over 20 runs, and the medians shown in Figure 3.2 are taken from the results over 30 instances). It can be seen that different attributes lead to different rates of improving the solution. Generally, the task-in-a-route and the 2-task attributes improve the solution at a faster rate than the 2-task-in-a-route, which in turn improves the solution at a faster rate than the 3-task attribute.

A two-tailed Wilcoxon signed-rank test (with the significance level of 0.05) is performed to analyse the significance of the differences between the results given by the four attributes at various points throughout the process of tabu search ($k \times n_t$ for $k = 10, 20, \dots, 100$). As shown in Table 3.2, it was found that both the task-in-a-route and the 2-task attributes are significantly better than the 3-task attribute at all the iteration numbers considered (i.e. $10n_t$, $20n_t$, $\dots$, $100n_t$). Furthermore, the task-in-a-route attribute is significantly better than the 2-task-in-a-route attribute at the iteration numbers $10n_t$ up to $50n_t$. This further supports that the task-in-a-route and the 2-task attributes could improve the solution at the fastest rate among the tabu attributes considered. It is also interesting to note that the 2-task-in-a-route

Table 3.2: Medians of percentage deviations from optimality (rounded to 2 decimal places) given by the tabu search algorithm with different types of tabu attributes; n_t denotes the number of tasks

Number of iterations	Type of tabu attribute			
	Task-in-a-route	2-task	2-task-in-a-route	3-Task
$10n_t$	$2.81^{*\dagger}$	$3.21^{\dagger}$	3.24	3.34
$20n_t$	$2.25^{*\dagger}$	$2.47^{\dagger}$	$2.73^{\dagger}$	3.10
$30n_t$	$1.93^{*\dagger}$	$1.89^{*\dagger}$	$2.45^{\dagger}$	2.91
$40n_t$	$1.75^{*\dagger}$	$1.67^{\dagger}$	$2.03^{\dagger}$	2.56
$50n_t$	$1.61^{*\dagger}$	$1.58^{\dagger}$	$1.88^{\dagger}$	2.36
$60n_t$	$1.60^{\dagger}$	$1.49^{\dagger}$	$1.75^{\dagger}$	2.16
$70n_t$	$1.52^{\dagger}$	$1.37^{\dagger}$	$1.68^{\dagger}$	2.09
$80n_t$	$1.47^{\dagger}$	$1.34^{\dagger}$	$1.60^{\dagger}$	2.03
$90n_t$	$1.38^{\dagger}$	$1.15^{\dagger}$	$1.48^{\dagger}$	1.95
$100n_t$	$1.31^{\dagger}$	$1.11^{\dagger}$	$1.43^{\dagger}$	1.86

* significantly better than the 2-task-in-a-route attribute
† significantly better than the 3-task attribute
based on a two-tailed Wilcoxon signed-rank test with a Bonferroni correction (for 6 pairwise comparisons), resulting in a significance level of $0.05/6 \approx 0.0083$

attribute is significantly better than the 3-task attribute from the iteration number $20n_t$ onwards.

Regarding elapsed time for executing the tabu search algorithm, Figure 3.3 shows that the task-in-a-route type generally uses less computation time than the other types. This suggests that when actual computation time is considered instead of the number of iterations, the task-in-a-route type is still among those that improve the solution at the fastest rate.

To better understand the behaviours of the tabu search algorithm with different types of tabu attributes, let us look at the results on a particular instance. Figure 3.4 shows total distances of current solutions over the course of the tabu search algorithm from 10 sample runs on the E17 instance (the other 10 runs are shown in Figure B.5 in Appendix B). The total distances corresponding to the task-in-a-route type tend to fluctuate more than the other types of attributes. In contrast, the total distances corresponding to the 3-task type fluctuate noticeably less than the other types of attributes; in fact, the total distances remain constant for a majority of iterations in some runs.

To see how the total distance could remain constant, notice that there can be many CARP solutions with the same total distance because tasks can be transferred

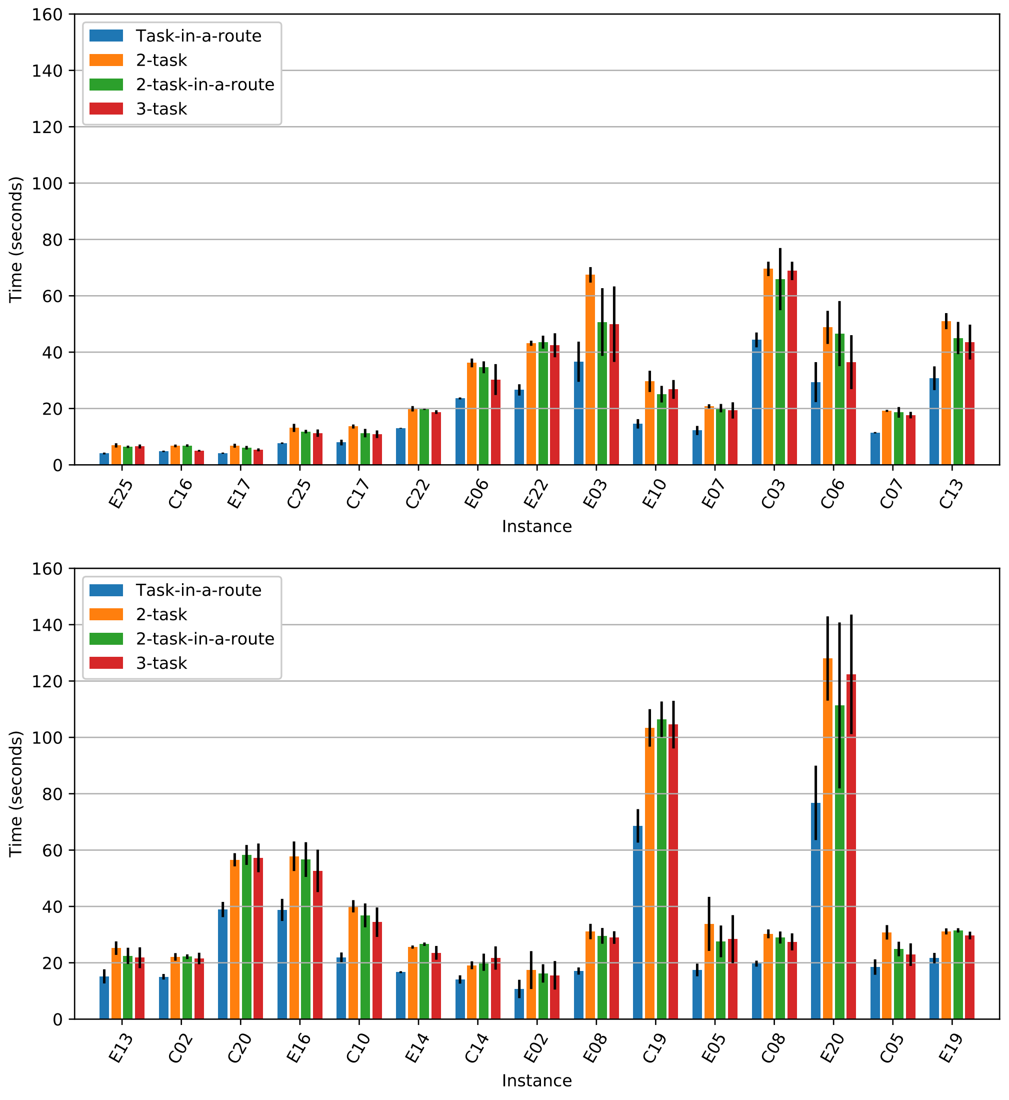
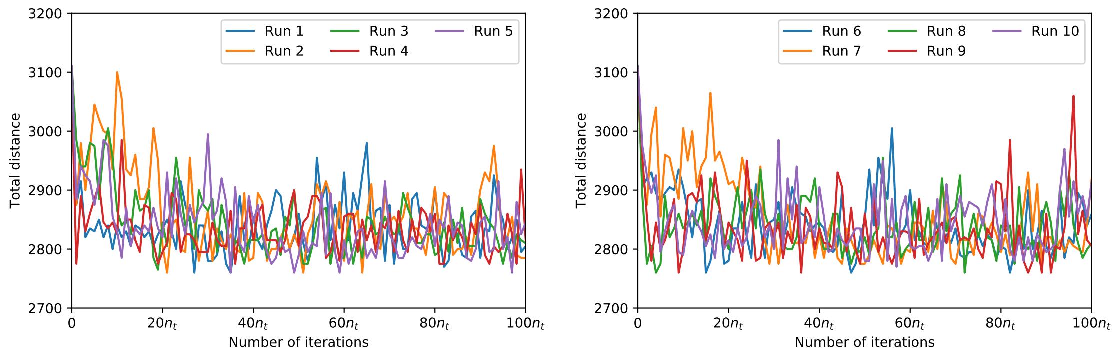
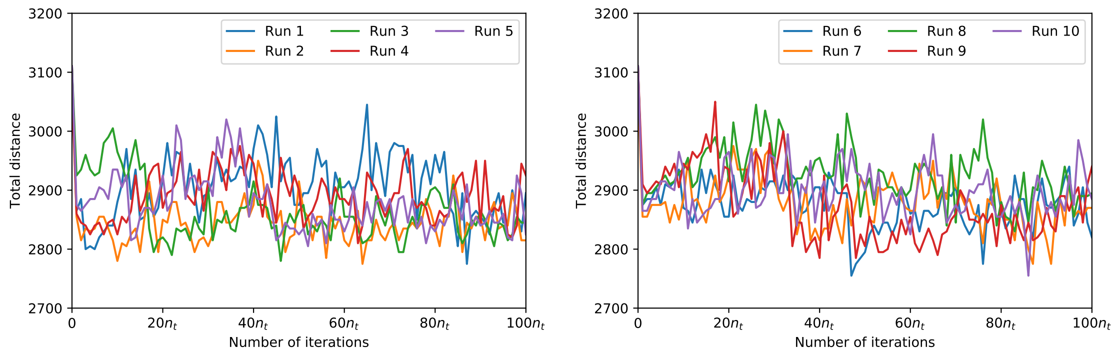


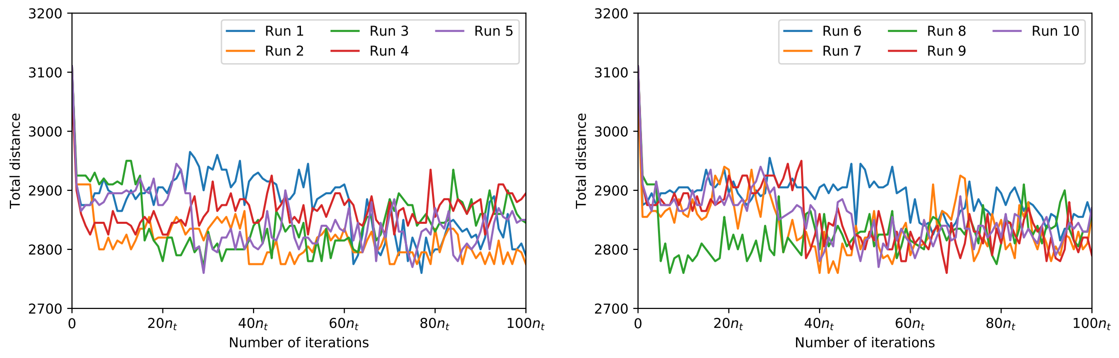
Figure 3.3: Average elapsed time for executing the tabu search algorithm with different tabu attributes for $100n_t$ iterations, where n_t is the number of tasks (averaged over 20 runs); black vertical lines show one standard deviation from each side of the averages



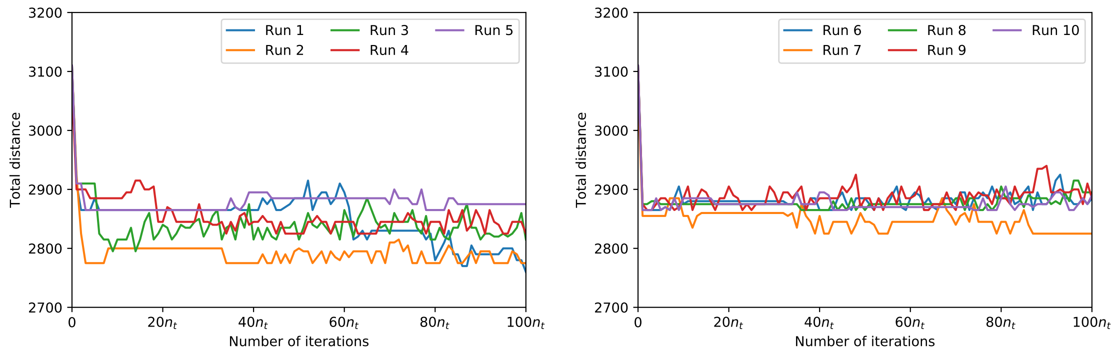
(a) Task-in-a-route



(b) 2-task

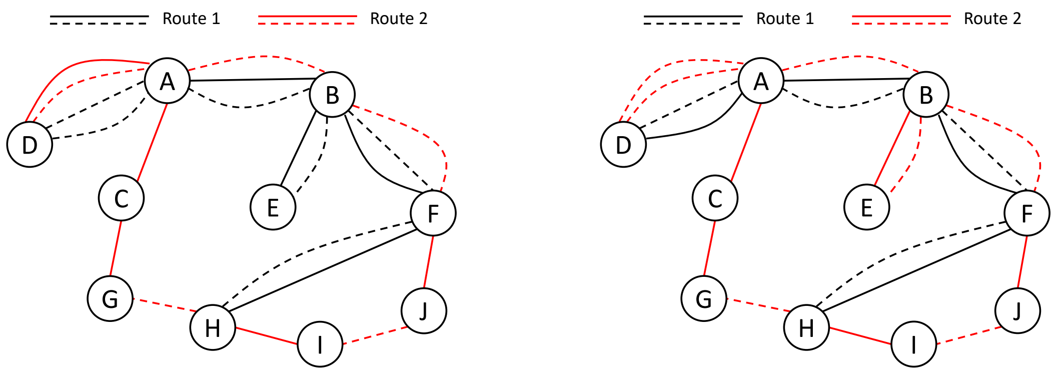


(c) 2-task-in-a-route



(d) 3-task

Figure 3.4: Total distances of current solutions over the course of the tabu search algorithm from 10 sample runs ("Runs 1-10") on the E17 instance for each type of attribute; n_t denotes the number of tasks



(a) Routes $(AB, \underline{BE}, BF, FH)$ and $(\underline{DA}, AC, CG, HI, JF)$

(b) Routes $(\underline{DA}, AB, BF, FH)$ and $(AC, CG, HI, JF, \underline{BE})$

Figure 3.5: Different sets of routes with the same total distance. Each set contains two routes shown in different colours (black and red). The depot is at vertex D. Dashed lines represent traversals without service and solid lines represent traversals with service. The routes in (a) are transformed into those in (b) by a swap move; the tasks that are affected by the move (BE and DA) are underlined.

between routes without changing the total number of times each edge is traversed. This is illustrated by Figure 3.5. Two routes in Figure 3.5(a) (one shown in black and the other in red) are transformed to two new routes in Figure 3.5(b) by swapping tasks BE and DA between their respective routes. Notice that the swap move does not affect the total number of traversals on each edge; it only transfers the service on some edge from one traversal to another, so the routes in Figure 3.5(a) and Figure 3.5(b) have the same total distance. What remains to be seen now is why the tabu search algorithm with the 3-task attribute was susceptible to stagnation of total distances.

It is worth reminding ourselves at this stage that the aspiration criterion was implemented in the tabu search algorithm. Table 3.3 shows the number of iterations in which a neighbourhood move is selected as a result of the aspiration criterion (i.e. it is tabu but leads to a solution better than the best one found so far). It turns out that the number of such iterations is very low compared with the number of all iterations for which the tabu search is executed (the maximum number is 9 out of 3,600 iterations). This means that the aspiration criterion plays a part in relatively few iterations in the results shown in Figure 3.4. It follows that the performance of each variant of the tabu search algorithm is largely affected by the determination of tabu/non-tabu moves, which is dependent on the selected type of attributes.

It is suspected that the reason why the 3-task type is more susceptible to stagnation of total distances than the other types of tabu attributes is because of its relatively

Table 3.3: The number of iterations (out of 3,600 iterations) in which a neighbourhood move was selected as a result of the aspiration criteria from 20 runs on the E17 instance

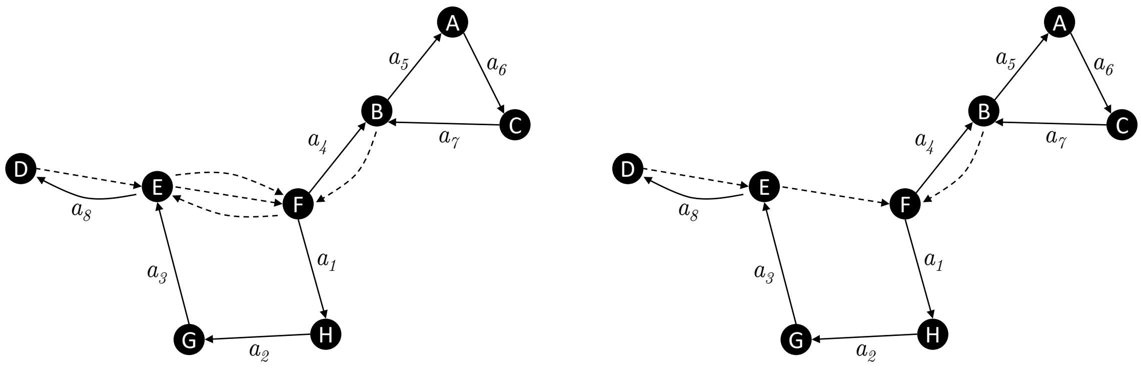
Attribute type	Average	Max	Min
Task-in-a-route	5.15	9	1
2-task	3.95	8	1
2-task-in-a-route	2.25	6	0
3-task	0.95	4	0

low "restrictiveness," i.e. a neighbourhood move is less likely to be tabu based on the 3-task type than the other types of attributes. For example, a 3-task attribute (s, t, u) prohibits the insertion of task t between tasks s and u. If tasks s and u are serviced consecutively in the same route R, then the 3-task attribute (s, t, u) prohibits only one insertion position in the route R (i.e. between s and u). Otherwise, the 3-task attribute (s, t, u) plays no role in the determination of tabu moves. In contrast, a task-route attribute (t, R) prohibits the insertion of task t into route R, regardless of the insertion position. This means the task-route type of attribute tends to make a large proportion of moves tabu, thereby potentially ruling out all moves with cost 0 and forcing the algorithm to visit solutions in further reaches of the solution space. Since there can be many CARP solutions with the same total distance, the abilities to move away from a certain total distance and to explore a wide variety of solutions seem important for achieving a high-quality solution over the course of execution of the tabu search algorithm.

The next section proposes an operator that attempts to improve the tabu search algorithm by detecting and removing certain undesirable features in a solution.

3.5 Deadheading Cycles

In practice, a route can contain both edges that are traversed with service and those that are traversed without service; the latter exist when tasks that are serviced consecutively in the same route are not physically adjacent to each other (or when the first/last task is not adjacent to the depot). An edge that is traversed without service is also referred to as a *deadheading edge*. It is possible that several deadheading edges in the same route form a cycle, called a *deadheading cycle*. If a deadheading cycle exists in some route in a given solution, it can be removed without affecting feasibility of the solution as long as it does not disconnect the route. The reasons



(a) Route $(D, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, D)$ (b) Route $(D, a_4, a_5, a_6, a_7, a_1, a_2, a_3, a_8, D)$

Figure 3.6: A route that contains a deadheading cycle (a) and an improved route after removing the deadheading cycle (b); the depot is at vertex D; a solid line represents a traversal with service, while a dashed line represents a traversal without service

why the feasibility of a solution is maintained after removing a deadheading cycle are that: (i) a deadheading cycle does not involve any services, so all tasks are still serviced and the capacity constraint remains satisfied, and (ii) with a route regarded as an Eulerian multigraph, removing a cycle preserves the parity of the degree of each vertex (since there are an even number of edges that are incident to each vertex in a cycle), and therefore the multigraph remains Eulerian after the removal of the cycle.

Provided that all edge costs are positive, removing a deadheading cycle always leads to a better solution. Figure 3.6 shows an example of a deadheading cycle and how the solution can be affected by the removal of the deadheading cycle. A route in Figure 3.6(a) contains a deadheading cycle (E, F, E). Removing this cycle results in the route shown in Figure 3.6(b). Notice that removing a deadheading cycle can affect the order of tasks that are serviced in the route: as can be seen in Figure 3.6, removing the deadheading cycle (E, F, E) transforms the route $(D, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, D)$ into $(D, a_4, a_5, a_6, a_7, a_1, a_2, a_3, a_8, D)$. In other words, the subsequence of tasks (a_4, a_5, a_6, a_7) was moved to the front of a_1. If the tabu search algorithm relies only on "small-step" neighbourhood moves (i.e. making changes to a small number of tasks) such as those defined in Section 3.2, it would require several neighbourhood moves to obtain the same change.

In order to detect deadheading cycles in a given route, we need to know the number of times each edge is traversed in total, counting traversals in any direction, both with and without services. To put it another way, a route should be viewed as an Eulerian multigraph in which the multiplicity of each edge is equal to the number of traversals on that edge. However, in the tabu search algorithm described in previous sections (as well as some other existing metaheuristic algorithms for the CARP in the literature), a route is viewed as a sequence of tasks. Although this provides a concise representation of a route to work with neighbourhood moves (since a neighbourhood move is essentially a means to move tasks within a solution), it omits information about deadheading edges, and thus it gives no indication of the existence of deadheading cycles. Given a route as a sequence of tasks $R = (v_0, a_1, a_2, \ldots, a_n, v_0)$, we can determine deadheading edges in the route by finding shortest paths between endpoints of consecutive tasks (and between the first/last task and the depot). Shortest paths between any two vertices can be found by a polynomial-time algorithm such as that of Dijkstra (1959).

Let G_{mult} be a multigraph such that the multiplicity of each edge in G_{mult} is equal to the number of times the edge is traversed in the route R. If an edge is traversed more than twice, at least two such traversals are deadheading because an edge can only be serviced at most once. Every two deadheading traversals on the same edge form a cycle in G_{mult}. Thus, for an edge that is traversed more than twice, there is guaranteed to be a deadheading cycle on that edge that can be removed without disconnecting the route. In contrast, for an edge that is traversed twice and both traversals are deadheading, careful consideration is needed because the removal of the deadheading cycle in this case may or may not disconnect the route; this is illustrated in Figure 3.7. A deadheading cycle in Figure 3.7(a) is removable, whereas a deadheading cycle in Figure 3.7(b) is not removable because removing it would disconnect the route and result in an invalid solution (i.e. the removal splits the route into two smaller routes, one of which does not contain the depot). Therefore, to ensure continuity of a route, here we opt to remove only deadheading cycles on edges that are traversed at least three times. In other words, we remove a deadheading cycle on each edge until the number of traversals on that edge reduces to either 1 or 2, depending on its parity.

After detecting and removing deadheading cycles, the consequent Eulerian multigraph G_{mult} can be converted back into a sequence of tasks by determining an Eulerian cycle in G_{mult}. If an edge corresponding to a task is traversed more than once (i.e. its multiplicity in G_{mult} is greater than one), it is assumed that the task is serviced on its first traversal.

For clarity, a pseudocode of the proposed procedure, which will also be referred to as the *deadheading cycle remover* (DCR), is given as Algorithm 3.

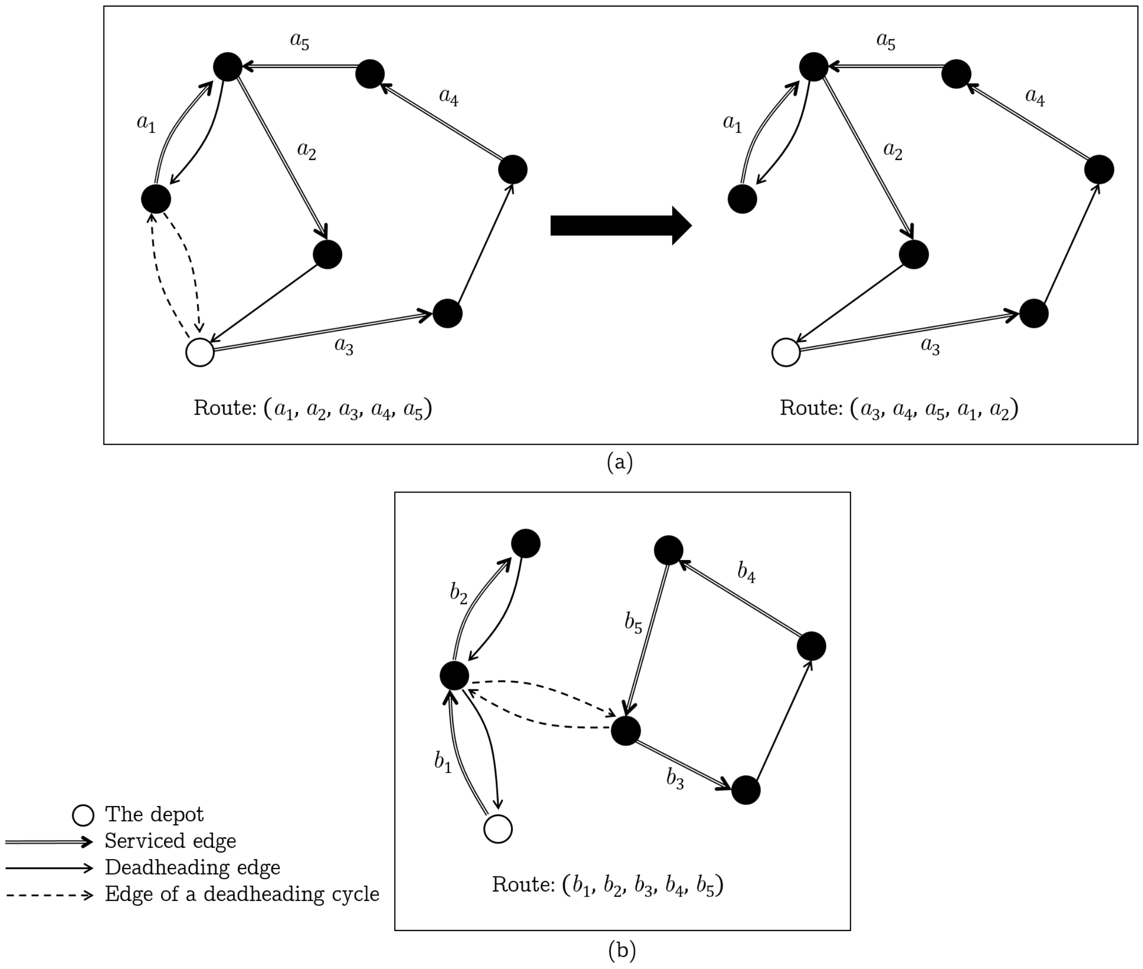

a_5
a_4
a_1
a_2
a_3
Route: $(a_1, a_2, a_3, a_4, a_5)$
Route: $(a_3, a_4, a_5, a_1, a_2)$
(a)
b_2
b_4
b_5
b_1
b_3
Route: $(b_1, b_2, b_3, b_4, b_5)$
(b)
The depot
Serviced edge
Deadheading edge
Edge of a deadheading cycle

Figure 3.7: A deadheading cycle on an edge that is traversed twice may be removable (a) or not removable (b)

Algorithm 3 The deadheading cycle remover

```
1: given a graph $G = (V, E)$ and a route as a sequence of tasks $(v_0, a_1, a_2, \ldots, a_n, v_0)$
2: /* step 1: count the number of traversals $n_t(e)$ on each edge $e \in E$ */
3: initialise $n_t(e) = 0$ for all edges $e \in E$
4: for $i = 0$ to $n$ do
5:     find a shortest path $P$ between $h(a_i)$ and $t(a_{i+1})$
6:     (where, for ease of notation, $a_0 = (v_0, v_0) = a_{n+1}$)
7:     for each edge $e$ in $P$ do
8:         increase $n_t(e)$ by 1                    ▷ count deadheading traversals
9:     if $i > 0$ then
10:        let $e$ be the edge corresponding to task $a_i$
11:        increase $n_t(e)$ by 1                    ▷ count traversals with services
12: /* step 2: remove deadheading cycles while keeping the route connected */
13: for each edge $e \in E$ do
14:     if $n_t(e) > 2$ then
15:         if $n_t(e)$ is odd then
16:             set $n_t(e) = 1$
17:         else
18:             set $n_t(e) = 2$
19: /* step 3: obtain a new route after removing deadheading cycles */
20: let $G_{\text{mult}} = (V, E)$ be a multigraph such that the multiplicity of each edge $e \in E$
    is equal to $n_t(e)$.
21: find an Eulerian cycle in $G_{\text{mult}}$, say $C = (v_0, e_1, v_1, e_2, v_2, \ldots, v_{m-1}, e_m, v_0)$
22: let $R' = (v_0)$                                 ▷ initialise a new route
23: for $i = 1$ to $m$ do          ▷ $m$ is the number of edges in the Eulerian cycle $C$
24:     if $e_i$ corresponds to some task $a_j$ and $a_j$ is not in $R'$ then
25:         append $a_j$ (with the appropriate direction, i.e. from $v_{i-1}$ to $v_i$) to $R'$
    append $v_0$ to $R'$                            ▷ close the route properly
26: return $R'$
```

3.5.1 Computational Results

First we will investigate whether deadheading cycles actually exist in the solutions given by the best variant of the tabu search algorithm from Section 3.4, namely the one in which tabu moves are defined based on the task-in-a-route attributes and the tabu tenure is $2n_t$, where n_t is the number of tasks. Figure 3.8 shows the number of runs in which there exists a removeable deadheading cycle (i.e. some edges were traversed more than twice) in the best observed solution given by the aforementioned variant at various iteration numbers $k \times n_t$, $k = 0, 1, \ldots, 100$. Recall that the algorithm was repeated 20 times on each instance, resulting in 600 runs (30 instances $\times$ 20 runs) in total. Among initial solutions (iteration 0), removable deadheading cycles existed in about half the runs (318 out of 600 runs). As the number of iterations increased, the number of runs with removable deadheading cycles dropped dramatically to just over 100 runs and then decreased steadily, eventually reaching and seeming to stagnate around 60, which account for 10% of all runs in this experiment, despite increasing the number of iterations to $100n_t$. This shows that some deadheading cycles in early iterations were indirectly removed as the tabu search algorithm attempted to find better solutions by means of neigbourhood moves, but afterwards the tabu search algorithm was not always able to completely remove deadheading cycles on its own even if it was given a relatively large number of iterations. The inability to remove deadheading cycles could be because the rule for selecting the best neighbourhood move in each iteration is largely based on (changes of) total distances, not on whether it would remove deadheading cycles. Implementing the DCR after the final iteration of the tabu search algorithm as a post-optimisation step would further improve the solution by dealing with deadheading cycles that have been overlooked by the tabu search.

Instead of implementing the DCR as a post-optimisation step, it is also possible to implement the DCR after the best neighbourhood move is applied to the current solution in each iteration. Figure 3.9 compares two ways of implementing the DCR: (i) after the final iteration, and (ii) in every iteration after the best neighbourhood move. More precisely, Figure 3.9 shows medians[2] of percentage deviations from optimality given by these two variants on a range of stopping criteria (that is, the number of iterations = $k \times n_t$, where $k = 0, 1, \ldots, 100$). For ease of reference, Figure 3.9 also displays the result corresponding to the algorithm without the DCR (the blue line with circle markers in Figure 3.2). The computational results suggest that implementing the DCR in either way could help the tabu search algorithm find

[2]As before, the percentage deviations on each instance are averaged over 20 runs, and the medians are taken from those averages over 30 instances.

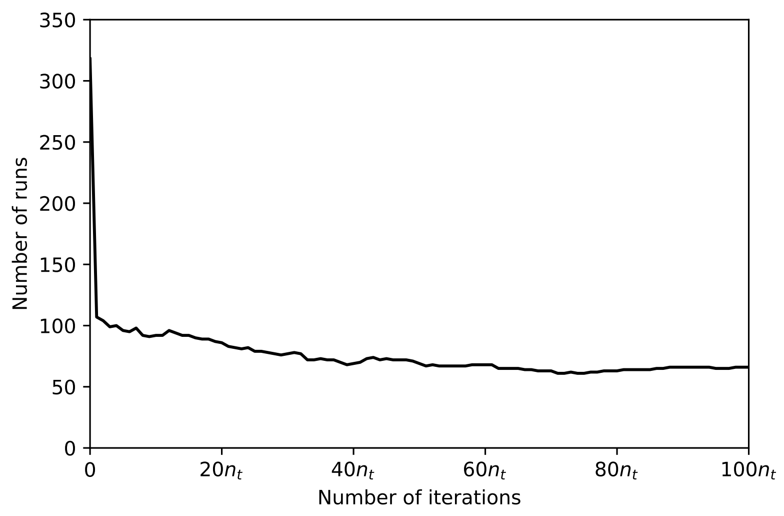


Figure 3.8: The number of runs (out of 600 runs) in which the best solution at various iteration numbers kn_t has removable deadheading cycles, where n_t is the number of tasks, and $k = 0, 1, \ldots, 100$

a better solution. Furthermore, implementing the DCR in every iteration generally results in lower total distance than implementing it only once after the final iteration over a wide range of stopping criteria.

A two-tailed Wilcoxon signed-rank test is conducted at various iteration numbers throughout the execution of tabu search ($k \times n_t$, where $k = 10, 20, \ldots, 100$). As shown in Table 3.4, it was found that implementing the DCR in either way leads to statistically significant improvement. Moreover, the difference between two ways of implmenting the DCR is statistically significant at early iterations ($10n_t$ and $20n_t$). This suggests that implementing the DCR in every iteration could help the tabu search algorithm improve the solution more effectively especially during early iterations. In other words, not only can the DCR be used to improve the solution as a post-optimisation step, but it can also be integrated into the tabu search algorithm and help the algorithm improve the solution at a faster rate.

Figure 3.10 shows average time taken on each instance by the tabu search algorithm without and with the DCR in every iteration (the results were obtained from running the algorithm on an Intel Core i5-4690 3.50GHz CPU with 8GB RAM). It can be seen that the computation times in both cases are very similar on all instances. This confirms that the DCR requires relatively little computation effort.

It should also be noted that the computation time taken by the tabu search with DCR is sometimes less than that without DCR. This is possible because implementing the DCR generally causes the tabu search algorithm to visit different solutions from

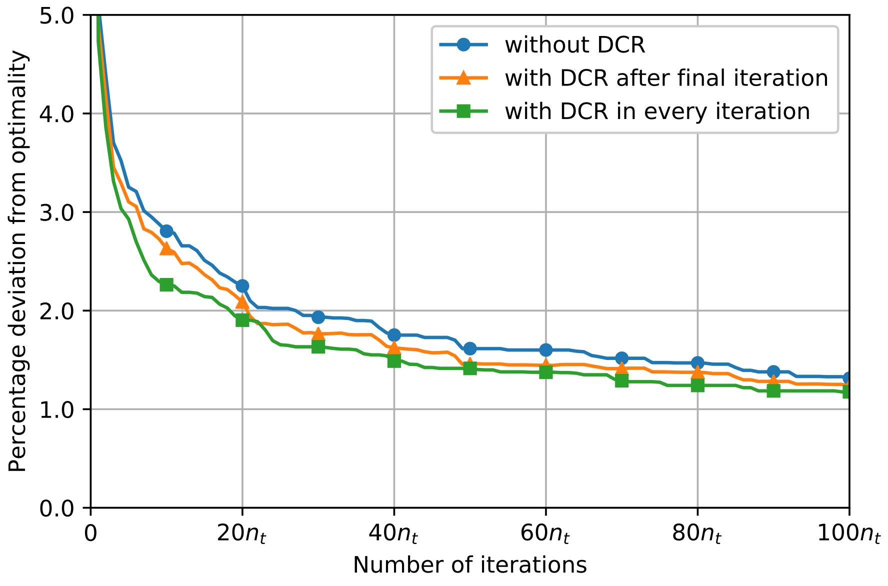


Figure 3.9: Medians of percentage deviations from optimality over a range of iteration numbers given by the tabu search algorithm with different ways of implementing the deadheading cycle remover, namely without the deadheading cycle remover (“without DCR”), with the DCR being implemented after the final iteration (“with DCR after final iteration”), and with the DCR being implemented in every iteration (“with DCR at every iteration”)

Table 3.4: Medians of percentage deviations from optimality given by the tabu search algorithm with different ways of implementing the deadheading cycle remover (“DCR”); n_t denotes the number of tasks

Number of iterations	Ways of implementing the DCR		
	Without DCR	With DCR after final iteration	With DCR in every iteration
$10n_t$	2.81	2.69*	2.26*†
$20n_t$	2.25	2.08*	1.90*†
$30n_t$	1.93	1.74*	1.63*
$40n_t$	1.75	1.70*	1.49*
$50n_t$	1.61	1.45*	1.41*
$60n_t$	1.60	1.42*	1.37*
$70n_t$	1.52	1.39*	1.29*
$80n_t$	1.47	1.32*	1.24
$90n_t$	1.38	1.29*	1.19*
$100n_t$	1.31	1.21*	1.18

* significantly better than “without DCR”

† significantly better than “with DCR after final iteration”

based on a two-tailed Wilcoxon signed-rank test with a Bonferroni correction (for 3 pairwise comparisons), resulting in a significance level of $0.05/3 \approx 0.017$

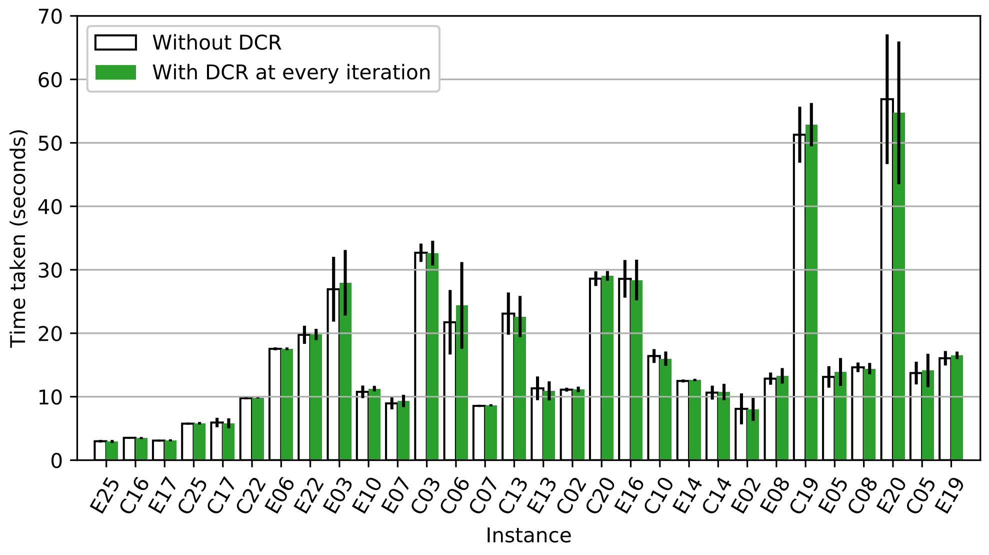


Figure 3.10: Average elapsed time for executing the tabu search algorithm on each instance for $100n_t$ iterations taken by the tabu search algorithm without the deadheading cycle remover ("without DCR") and with the DCR being implemented in every iteration ("with DCR at every iteration"); black vertical lines show one standard deviation from each side of the averages

those that the algorithm would without DCR, and different solutions generally have different numbers of feasible neighbour solutions. It could be the case that a solution visited by tabu search without DCR has a smaller number of feasible neighbour solutions, hence a smaller total amount of time taken to explore neighbourhood moves (even if the computation time taken by the DCR is included).

3.6 Notes on the Use of Multiple Tabu Lists

Recall that the variant of the tabu search algorithm that is based on the task-in-a-route attribute uses two separate tabu lists, one for Single Insertion, Double Insertion, and Swap, and the other for 2-opt moves; the latter is based on the 2-task attribute. Even though we saw in Section 3.4 that this variant showed the best performance among all four types of solution attributes being tested, it was later found that using two separate tabu lists could inadvertently allow the algorithm to revisit previous solutions, even if the corresponding attributes are currently tabu. An example of this phenomenon is given in Figure 3.11. The last move in Figure 3.11 is not tabu according to the task-in-a-route attribute: a_5 could be inserted into route A and b_4 into route B because both insertions (corresponding to the attributes $(a_5, \text{Route } A)$ and $(b_4, \text{Route } B)$) are not currently in the tabu list. Notice that a_5 was removed

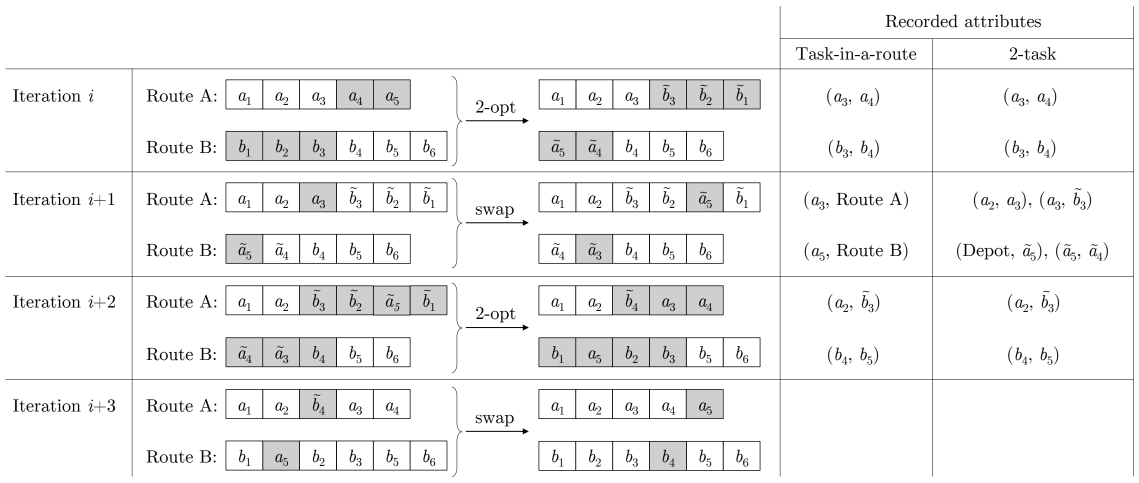

Recorded attributes
Task-in-a-route
2-task
Iteration i
Route A: a_1 a_2 a_3 a_4 a_5
Route B: b_1 b_2 b_3 b_4 b_5 b_6
2-opt
a_1 a_2 a_3 $\tilde{b}_3$ $\tilde{b}_2$ $\tilde{b}_1$
$\tilde{a}_5$ $\tilde{a}_4$ b_4 b_5 b_6
(a_3, a_4)
(b_3, b_4)
(a_3, a_4)
(b_3, b_4)
Iteration i+1
Route A: a_1 a_2 a_3 $\tilde{b}_3$ $\tilde{b}_2$ $\tilde{b}_1$
Route B: $\tilde{a}_5$ $\tilde{a}_4$ b_4 b_5 b_6
swap
a_1 a_2 $\tilde{b}_3$ $\tilde{b}_2$ $\tilde{a}_5$ $\tilde{b}_1$
$\tilde{a}_4$ $\tilde{a}_3$ b_4 b_5 b_6
$(a_3$, Route A$)$
$(a_5$, Route B$)$
(a_2, a_3), $(a_3, \tilde{b}_3)$
(Depot, $\tilde{a}_5$), $(\tilde{a}_5, \tilde{a}_4)$
Iteration i+2
Route A: a_1 a_2 $\tilde{b}_3$ $\tilde{b}_2$ $\tilde{a}_5$ $\tilde{b}_1$
Route B: $\tilde{a}_4$ $\tilde{a}_3$ b_4 b_5 b_6
2-opt
a_1 a_2 $\tilde{b}_4$ a_3 a_4
b_1 a_5 b_2 b_3 b_5 b_6
$(a_2, \tilde{b}_3)$
(b_4, b_5)
$(a_2, \tilde{b}_3)$
(b_4, b_5)
Iteration i+3
Route A: a_1 a_2 $\tilde{b}_4$ a_3 a_4
Route B: b_1 a_5 b_2 b_3 b_5 b_6
swap
a_1 a_2 a_3 a_4 a_5
b_1 b_2 b_3 b_4 b_5 b_6

Figure 3.11: An example of how a solution can be revisited; each a_i and b_j represent tasks with specified directions; $\tilde{a}$ denotes the opposite direction of a

from Route A by 2-opt moves (see Iterations i and $i+2$); if it were removed by a swap move, the attribute $(a_5, \text{Route } A)$ would have been recorded, but it was not because a different tabu list was used for 2-opt moves. In contrast, the same move would be tabu and thus forbidden (unless the aspiration criterion is satisfied) according to the 2-task attribute because all pairs corresponding to the swap move (namely, $(a_4, a_5), (a_5, \text{Depot}), (b_3, b_4)$, and (b_4, b_5)) are currently in the tabu list.

Another look at Figure 3.2 reveals that among all the types of attributes considered here excluding the task-in-route-attribute, it is the 2-task attribute that generally improves the solution at the fastest rate. We thus opt to investigate a way of solving a dynamic CARP in the next chapter based on the tabu search algorithm with the 2-task attribute.

3.7 Conclusions

This chapter presents a novel analysis of the performance of tabu search for the static CARP with different ways of defining tabu moves. In particular, we compare their performance over different iteration limits (instead of looking at their results based on a single stopping criteria) and attempt to identify key features of the algorithm that can enhance or inhibit its performance. The goal of this chapter is to see whether the best variant would vary according to the number of iterations for which the tabu search was allowed to run. This will help us decide which algorithm to use in a dynamic CARP, where the amount of time available for updating the solution is often limited. In particular, a variant that could improve the solution relatively fast would be preferable to one that could find a very good solution after long computation time but improves the solution at a much slower rate. In the tabu search algorithm considered in this chapter, the tabu status of neighbourhood moves was determined in the following way: solution attributes corresponding to the selected move in each iteration are recorded in the tabu list, and a neighbourhood move is said to be tabu if solution attributes that arise as a result of the move are currently in the tabu list. A comparison was made between four types of solution attributes, namely task-in-a-route, 2-task, 2-task-in-a-route, and 3-task attributes.

Computational results suggested that the tabu search algorithm based on the task-in-a-route attribute generally gave the best solutions across a wide range of numbers of iterations; this type of attribute also used the least amount of computation time on average. The worst type of attribute, namely the 3-task attribute, was seen to suffer from stagnation of total distance, whereas the other types of attributes did not encounter the same issue as much. This was suspected to be caused by its relatively low restrictiveness, i.e. a neighbourhood move is less likely to be tabu based on the 3-task attribute than the other types of attributes. As the CARP can have many solutions with the same total distance, a type of attribute that is sufficiently restrictive would make many moves tabu and encourage the algorithm to visit solutions that are different from previously visited solution, thereby helping the algorithm to avoid getting stuck with a group of solutions with certain total distance. Thus, the notion of "tabu moves" for the CARP in particular is not only for avoiding previously visited solutions, but it can also help avoid getting stuck in a set of solutions with the same total distance when defined appropriately. This seems to be one factor that helps tabu search improve the solution at a relatively fast rate.

A novel operator named the deadheading cycle remover (DCR) was proposed to improve the performance of tabu search. It was motivated by the fact that a cycle composed of deadheading cycles, i.e. traversals without services, can be removed from a route without affecting the feasibility of a solution as long as the route remains connected after the removal. Furthermore, empirical results showed that the tabu search algorithm could not always completely remove deadheading cycles on its own. It was found that implementing the DCR indeed significantly improved the solutions returned by the tabu search algorithm. In fact, implementing the DCR in every iteration leads to significant improvement at early iterations compared with implementing the DCR only once after the final iteration (i.e. using it as a post-optimisation step); in other words, the DCR can improve tabu search's speed of improving CARP solutions. It is also worth noting that the DCR requires only little additional computational effort.

Recall that the variant of our tabu search algorithm that is based on the task-in-a-route attribute actually involves another attribute: a 2-task attribute is considered instead when a 2-opt move is selected because using the task-in-a-route attribute would involve recording and checking a large number of pairs. Even though empirical results suggest that this variant is among those that improve a CARP solution at the fastest rates, it was found that defining tabu moves based on multiple attributes could fail to prevent tabu search from revisiting previous solutions. Among the other variants of tabu search considered in this chapter, the 2-task attribute yields the best performance. For this reason, the tabu search algorithm in which the definition of tabu moves is based on the 2-task attribute will be used to tackle a dynamic CARP in subsequent chapters.

In the next chapter, we will describe in greater detail how the dynamic CARP can be tackled and will investigate how the dynamic CARP solution can be affected by the frequency of updating the solution.

Chapter 4

Dynamic Capacitated Arc Routing Problem

4.1 Introduction

A dynamic CARP is an extension of the CARP in which some information in the problem changes while vehicles are travelling and servicing tasks. Those changes may cause a set of routes that are planned before the changes to have lower quality with respect to total distance (or some other quantity that is being optimised) or even become infeasible, hence the need to update the routes accordingly. There are many types of changes that can be considered in a dynamic CARP as we saw in Section 2.10. The type of change that we focus on in this thesis is the appearance of new demands.

In practice, route planning and amending need to be performed in an indefinite period of time as long as new tasks appear (Psaraftis, 1980). For the purposes of this thesis, however, we restrict our attention to a finite period of time, which will also be referred to as a *planning horizon.* It is assumed that all demands that appear within the planning horizon cannot be rejected and must be serviced.[1] In other words, those demands must be included in the solution by the end of the planning horizon. Nevertheless, they do not need to be added to the solution immediately when they appear. This means that a route planner can decide when to update the solution.

[1]In practice, demands that appear after the end of the planning horizon would be dealt with in the next planning horizon, e.g. the next working day.

In an extreme case, a route planner may decide to update the solution just once at the end of the planning horizon (while keeping all vehicles idle at the depot until the end of the planning horizon), which would allow all tasks to be added to the solution at the same time. This would be an ideal approach if the total distance is the only quantity that needs to be minimised. However, this can greatly delay the completion of the service since those new tasks could be serviced only after the solution is updated. Therefore, in this chapter, we consider performing a number of solution updates over the planning horizon as new tasks appear while vehicles are travelling and servicing tasks. We are interested in finding a way to amend a solution as new tasks appear while ensuring that both total distance and service completion time do not increase excessively.

Notice that updating a solution infrequently would allow the route planner to deal with many tasks at the same update. However, a large amount of time between updates means that a large proportion, if not the whole, of a route would be traversed (assuming that the corresponding vehicle travels without stopping from leaving the depot until returning to the depot) and thus could no longer be amended. This reduces the number of possible ways in which a solution can be changed. In contrast, updating the solution more frequently would give more flexibility in making changes to the solution, although a route planner would have less information about new tasks to exploit in each update. This illustrates the need to identify a solution update frequency that facilitates effective route planning for the dynamic CARP (and dynamic routing problems in general).

This chapter describes how the dynamic CARP will be tackled and investigates how the frequency of updating the solution can affect solution quality with respect to both total distance and service completion time, i.e. the time at which all vehicles return to the depot after servicing all tasks. Apart from the frequency of updating the solution, a comparison will also be made between different ways of integrating new tasks into an existing set of routes.

Section 4.2 describes components of a dynamic CARP solver. Section 4.3 explains a way of generating instances for the dynamic CARP with new tasks. Computational results given by variants of a dynamic CARP solver are shown and discussed in Section 4.4. Section 4.5 compares different ways of integrating new tasks into a solution. The conclusions of this chapter are given in Section 4.6.

4.2 Components of a Dynamic CARP Solver

Our process of finding a solution for the dynamic CARP can be divided into three main components: deciding when to update the solution, determining the current state of the problem at each update, and amending the solution subject to the current state of the problem. We now consider these in turn.

4.2.1 Solution Update Schedules

Here we shall focus on a *regular update schedule*: that is, the solution is updated at regular intervals. More precisely, let T be the length of the planning horizon and N the number of updates, which is to be specified. Without loss of generality, let the planning horizon start at time 0 and end at time T. Solution updates then take place at time $\frac{kT}{N}$ for $k = 1, \dots, N$. Intuitively, fewer updates mean more time to collect information about new tasks that appear in the interval prior to each update, while more updates means new tasks can be dealt with more promptly. A regular update schedule has been used in dynamic vehicle routing (Montemanni et al., 2005; Chen and Xu, 2006). In particular, computational results given by Montemanni et al. (2005) showed that the best solution (with respect to total distance) could be achieved when the number of updates was neither too high nor too low, although it was not explicitly investigated how (or whether) such "promising" number of updates would vary with the rate at which new tasks appear. Later in this chapter (Section 4.4), several numbers of updates will be tested on dynamic CARP instances with a range of rates at which new tasks appear.

4.2.2 Determining the Current State of the Problem

Before each solution update, the current state of the problem needs to be determined. This involves updating the set of tasks to be serviced, the vehicles' positions, and their remaining capacities. The current state of the problem depends on both changes in the problem itself (new tasks appear) and solutions from previous updates (vehicles travel around to service tasks). Also, notice that parts of the routes that have been traversed cannot be amended, so those parts should be clearly identified to ensure that changes made to the solution are feasible. For clarity, this section describes how the current state of the problem and the (non-)amendable parts of the routes are determined.

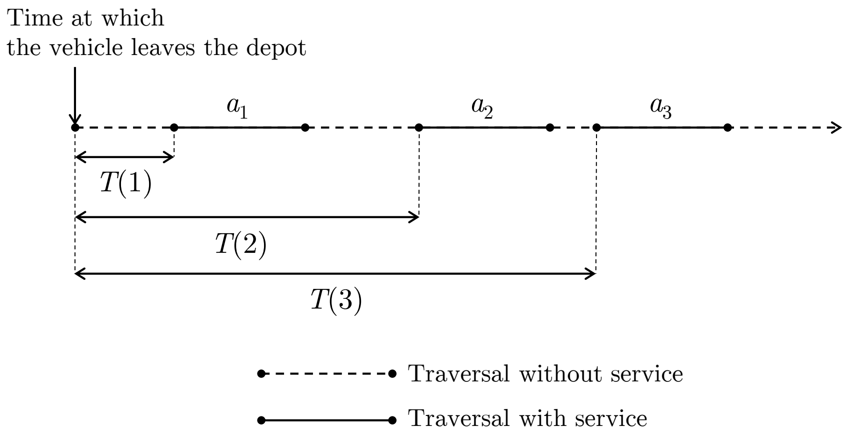


Figure 4.1: The time $T(i)$ at which a vehicle reaches the i^{th} task in its route.

Given the time at which the solution is updated, each route $R = (v_s, a_1, \dots, a_n, v_0)$ is divided into two parts, for some index $\tilde{i}$ and some vertex $\tilde{v}_s$:

$$(v_s, a_1, \dots, a_{\tilde{i}}, \tilde{v}_s); (\tilde{v}_s, a_{\tilde{i}+1}, \dots, a_n, v_0) \tag{4.1}$$

Here the first part cannot be amended (i.e. it is fixed), whereas the second part can still be amended. The *task fixing index* $\tilde{i}$ is the largest index i such that a_i has been reached by the vehicle corresponding to the route R. More precisely, let $c(a)$ denote the cost of an arc a, and $D(a, b)$ the shortest distance from the head of an arc a to the tail of another arc b. For ease of notation, let $a_0 = (v_s, v_s)$ and $c(a_0) = 0$. The task fixing index can be determined by finding the largest index i such that

$$T(i) = \frac{1}{\nu} \sum_{k=1}^{i} [c(a_{k-1}) + D(a_{k-1}, a_k)] \leq t_u - t_s(R), \tag{4.2}$$

where $T(i)$ is the time at which task a_i is reached (with $T(0) = 0$ by convention), t_u is the time of the update, $t_s(R)$ the start time of the route R (not all routes need to leave the depot at time 0; see Section 4.2.3), and ν the vehicle speed (distance per unit time); here it is assumed that ν is a constant given as part of the problem and that all vehicles travel at the same constant speed. Inequality (4.2) ensures that $a_{\tilde{i}}$ is included in the fixed part not only when it has been serviced but also when it is being serviced at the time of the update; see Figure 4.1 for the illustration. This agrees with the assumption that partial service is not allowed (see Section 2.3), from which it follows that if a vehicle is currently servicing $a_{\tilde{i}}$, it must continue the service until completion, and therefore it is certain that $a_{\tilde{i}}$ is serviced by this vehicle at its current order in the route.

The *starting vertex* $\tilde{v}_s$ in the expression (4.1) is the first vertex to be visited in the route after the given update. The way in which $\tilde{v}_s$ is determined depends on the current position of the vehicle: (i) if the vehicle is precisely at some vertex at the time of the update, then $\tilde{v}_s$ is simply that vertex; (ii) if the vehicle is in the middle of an edge, then $\tilde{v}_s$ is the endpoint of that edge which the vehicle is heading towards. That edge may be either a task that is being serviced by the vehicle or an edge in the middle of a deadheading path between two consecutive tasks a_i and a_{i+1} for some i (in fact, the index i here is the task fixing index $\tilde{i}$).

Once the task fixing index and the starting vertex of each route are determined, its remaining capacity can be updated accordingly by simply subtracting the total demand of tasks in the fixed part from the remaining capacity at the previous update. After that, its fixed part is archived (so at the end of the planning horizon, a complete journey of each vehicle is a concatenation of fixed parts of the corresponding route that are archived in all updates). The next step is to combine the amendable parts of the routes and the set of new tasks that have appeared since the previous update.

It should be noted that even though initially (at time 0) all vehicles have the same capacity and their routes start at the same vertex (the depot), they can service different amounts of demands and be at different locations at later time $t > 0$. This means that when updating the solution within the planning horizon, it is possible that a route planner encounters a more general version of the CARP, namely an *open* CARP with *heterogeneous* vehicles. Here, "open" refers to an open route, i.e. a vehicle's starting and ending vertices are not necessarily the same (it could be away from the depot at the time of the update), and "heterogeneous" means that different vehicles can have different (remaining) capacities.

4.2.3 Integrating New Tasks into the Solution

Once the current state of the problem is determined, new tasks need to be integrated into the solution to form a feasible solution subject to the current state of the problem. One way to do so is to reconstruct the solution from scratch. More precisely, all existing tasks are removed from the routes so the routes only contain their starting and ending vertices. Then, those tasks together with new tasks are added back to the routes using the Path Scanning algorithm (described in Section 2.5). Although the Path Scanning algorithm was originally designed for the standard CARP (where all vehicles have the same capacity and start their routes from the same vertex), it can be easily adapted for an open CARP with heterogeneous vehicles. Each route is reconstructed by adding one task at a time to the end of the route. The task to be added is the one that is nearest to the end vertex of the last task (or nearest to the starting vertex of the route if it is the first task). When there is more than one nearest task, one of them is chosen according to a given tie-breaking rule. Golden et al. (1983) proposed 5 tie-breaking rules: (i) maximise the distance from the endpoint to the depot; (ii) minimise the distance from the endpoint to the depot; (iii) use rule (i) if the vehicle's remaining capacity is at least half the whole capacity, and use rule (ii) otherwise; (iv) maximise the ratio demand/cost; and (v) minimise the ratio demand/cost. Tasks are iteratively added to the routes until no more tasks can be added due to the capacity. The other routes are constructed in the same way until all tasks are added to the routes. If the existing routes are not sufficient to service all the tasks, then a new route is constructed with the starting vertex being the depot (so this route leaves the depot after time 0). Each tie-breaking rule gives one solution, and the best among those solutions is chosen as the output of the algorithm. If there is more than one best solution, one of them is chosen randomly.

Obviously, reconstructing the solution from scratch is not the only way to integrate new tasks into the solution. Later on in this chapter (Section 4.5), an alternative way of integrating new tasks will be considered.

After new tasks are integrated into the solution, an attempt is then made to improve the solution by means of a variant of the tabu search algorithm from Chapter 3, namely the one in which the definition of tabu moves is based on the 2-task attribute.

A pseudocode for the whole dynamic CARP solver is given in Algorithm 4.

4.3 Generation of Dynamic CARP Instances

To test variants of the dynamic CARP solver, a number of dynamic CARP instances are required. There exists a benchmark instance generator for dynamic CARPs with various types of changes in the literature (see Liu et al., 2014a). However, this generator explicitly specifies the number of updates, the time of each update, and a set of changes that need to be considered at each update. In contrast, for the dynamic CARP being considered here, the appearance time of each new task is given in the problem, whereas the number of updates and the time of each update are variables to be decided by a route planner; a set of new tasks in each update would depend on these decisions. To the best of our knowledge, there are no instances for this type of dynamic CARP at the time of writing. Consequently, this section

Algorithm 4 Configuration of the dynamic CARP solver

```
1: given max_time (i.e. the length of the planning horizon), and a rule for deciding
   when to perform solution updates (see Section 4.2.1)
2: construct an initial solution $S$ by the Path Scanning algorithm (see Section 2.5)
3: apply the tabu search algorithm (from Chapter 3) to $S$
4: set time = 0
5: while time < max_time do
6:     increase time by 1 unit
7:     if time is an update time then
8:         let $\mathcal{T}$ be the set of new tasks that appear since last update
9:         if there exist new tasks then
10:            identify the fixed part and the starting vertex of each route for the
               current update (see Section 4.2.2)
11:            remove tasks from the non-fixed part of each route and add them to
               the set of tasks $\mathcal{T}$
12:            reconstruct the solution $S$ with the set of tasks $\mathcal{T}$ by the Path Scanning
               algorithm
13:            apply the tabu search algorithm (from Chapter 3) to $S$
```

describes how new instances for this type of dynamic CARP were generated for experiments in this chapter.

Before introducing a way of generating dynamic CARP instances, it is useful to know how to measure "dynamism," which will help to classify those instances according to how changes occur. A task that appears after time 0 will be called a *dynamic task*. The *degree of dynamism* (DoD) of a dynamic CARP instance is defined as the ratio of the number of dynamic tasks to the number of all tasks, including those that appear at time 0. This follows the definition of the degree of dynamism for dynamic vehicle routing given by Lund et al (1996) (as cited in Larsen and Madsen, 2000).

Dynamic CARP instances in this chapter were generated based on existing static CARP instances. Notice that the only thing that needs to be done once given a static CARP instance is to specify the time at which each task appears. Given a static CARP instance, the appearance time of each task is determined (provisionally) by randomly choosing an integer[2] $t \in \{1, 2, \ldots, T\}$ with the uniform distribution (recall that T is the length of the planning horizon). The tasks are then arranged in a sequence in a random order. Let n_t denote the number of tasks in the given static CARP. For each degree of dynamism $\delta = 0.1, 0.2, \ldots, 0.9$, the first $n_t \times \delta$ (rounded to the nearest integer) tasks in the sequence are regarded as dynamic tasks, while

[2]This discrete representation of time is designed to resemble a logging system that records the time at which each new task appears in the format "hour:minute" or "hour:minute:second," where the smallest unit of time (minutes or seconds) is discrete.

the rest are static tasks, i.e. their appearance times are reset to 0. One execution of this process gives 9 dynamic CARP instances (one for each degree of dynamism) per static CARP instance.

Notice that, among those 9 dynamic CARP instances from the same execution, a task that is dynamic on an instance with smaller δ is also dynamic and has the same appearance time on an instance with larger δ. This way of generating dynamic CARP instances allows us to meaningfully compare total distances on dynamic CARP instances with different degrees of dynamism: at the end of the planning horizon, a feasible solution on an instance with larger δ is also feasible on an instance with smaller δ. To see this, let $\mathcal{I}_1, \mathcal{I}_2$ be dynamic CARP instances that are generated in the same execution with the DoD on $\mathcal{I}_1$ smaller than that on $\mathcal{I}_2$. Let S_1, S_2 be feasible solutions at the end of the planning horizon on $\mathcal{I}_1, \mathcal{I}_2$, respectively. Because the appearance time of any task, say t, in $\mathcal{I}_2$ is no less than the appearance time of the same task t in $\mathcal{I}_1$, the service on t in S_2 starts no earlier than the appearance time of t in S_1 (notice that the time at which the service on a given task starts must be greater than or equal to the appearance time of the same task). Thus, the solution S_2 would also be feasible in the instance $\mathcal{I}_1$. It follows that the optimal total distance on an instance with larger DoD is no less than the optimal total distance on an instance with smaller DoD that is based on the same static CARP instance and the same appearance times.

For experiments in this chapter, a set of dynamic CARP instances has been generated based on 20 existing static CARP instances from the BMCV dataset (Beullens et al., 2003). Details of these static CARP instances are shown in Table 4.1. The best known lower and upper bounds on these instances are taken from `http://logistik.bwl.uni-mainz.de/benchmarks.php` (last accessed 23 March 2018). The numbers of tasks in the instances range from 66 to 107, and the numbers of vehicles that are needed (i.e. the total demand divided by the vehicle capacity, rounded up to the nearest integer) range from 6 to 12. The length of the planning horizon is set to 500, which is roughly equal to the average route cost in optimal solutions on the static CARP instances (the last column in Table 4.1); this helps prevent the planning horizon from being so long that a new task appears after all vehicles return to the depot. In total, 360 dynamic CARP instances were generated: 20 static CARP instances $\times$ 9 degrees of dynamism ($\delta = 0.1, 0.2, \ldots, 0.9$) $\times$ 2 rounds of generating dynamic CARP instances (with different sets of dynamic tasks and their appearance times in different rounds).

Table 4.1: Characteristics of static CARP instances on which a generation of dynamic CARP instances is based; LB and UB are the best known lower and upper bounds, respectively (LB is omitted when UB is optimal), and n_{veh} is the least number of vehicles needed (total demand divided by capacity, rounded up to the nearest integer); the capacity is 300 for all instances

Instance	Number of vertices	Number of edges	Number of tasks	LB	UB	Total demand	n_{veh}	$\frac{\text{UB}}{n_{\text{veh}}}$
C01	69	98	79		4150	2490	9	461.1
C04	60	84	72		3510	2170	8	438.8
C09	76	117	97	5245	5260	3440	12	438.3
C11	83	118	94	4615	4630	2825	10	463.0
C12	62	88	72		4240	2630	9	471.1
C15	97	140	107	4920	4940	3080	11	449.1
C21	60	84	76		3970	2245	8	496.3
C23	78	109	92	4075	4085	2395	8	510.6
C24	77	115	84		3400	2040	7	485.7
E01	73	105	85	4900	4910	2975	10	491.0
E04	70	99	77		4155	2545	9	461.7
E09	93	141	103	5805	5820	3585	12	485.0
E11	80	113	94		4650	2820	10	465.0
E12	74	103	67		4180	2485	9	464.4
E15	85	126	107		4205	2615	9	467.2
E18	78	110	88		3835	2225	8	479.4
E19	77	103	66		3235	1800	6	539.2
E21	57	82	72		3730	2025	7	532.9
E23	93	130	89		3710	2280	8	463.8
E24	97	142	86		4020	2235	8	502.5

4.4 Comparison of Variants of the Dynamic CARP Solver

This section presents a novel analysis of how the dynamic CARP solution can be affected by adjusting two key components of the dynamic CARP solver: the number of iterations of tabu search in each update, and the frequency of solution updates. Several variants of the dynamic CARP solver are tested on the instances generated in Section 4.3. Due to the stochastic nature of the algorithm, each variant is run 20 times on each dynamic CARP instance and its performance on that instance is assessed based on average results over 20 runs.

The algorithm performance on a given dynamic CARP instance will be measured in relation to *a posteriori lower bound*, which is the best known lower bound on the corresponding static CARP instance (in other words, when all tasks are treated as if they are all known in advance). More precisely, the algorithm performance will be reported in the form of percentage deviations from a posteriori lower bounds:

$$\text{percentage deviation} = \left(\frac{\text{solution cost} - \text{a posteriori lower bound}}{\text{a posteriori lower bound}}\right) \times 100. \quad (4.3)$$

In particular, if the optimal cost for a given instance is known, then the percentage deviation is calculated by putting the optimal cost in place of the lower bound in Equation (4.3).

4.4.1 The Number of Iterations of Tabu Search in Each Update

In the static CARP, executing tabu search for more iterations would give a better (or at least equally good) solution. In the dynamic CARP, however, it is not guaranteed that a better solution at one update leads to a better solution at a subsequence update (an example is given in Figure 1.2). Thus, it remains unclear whether executing tabu search for more iterations in each update would give a better solution (with respect to total distance) at the end of the planning horizon. This section presents a comparison between variants of the dynamic CARP solver that differ in how long tabu search is executed in each update.

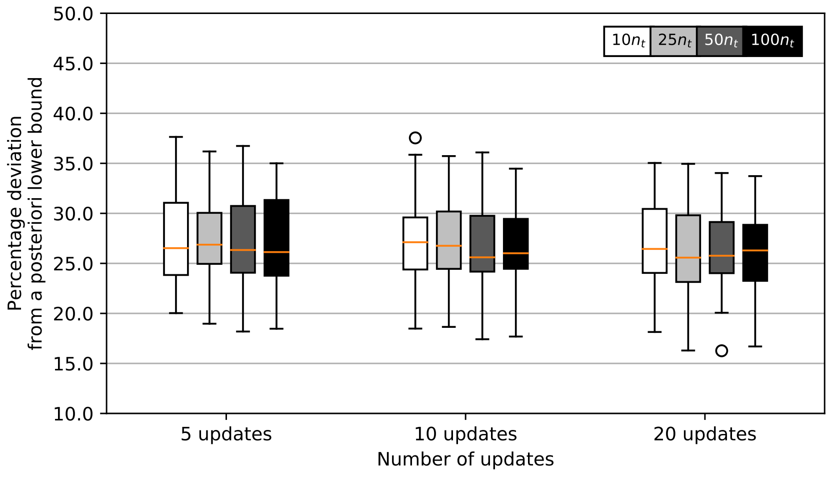
Notice that the number of tasks that remain to be serviced can vary over time as new tasks appear and existing tasks are serviced. It is anticipated that the number of iterations that tabu search would need to improve a solution varies with the number of tasks. For this reason, the maximum iteration limit for the tabu search algorithm in each update is given in the form of $k \times n_t$, where n_t is the number of tasks in the given update (that are not in fixed parts of the routes; see Section 4.2.2 for the description of a fixed part of a route) and k is a constant to be specified. In this section, four different maximum iteration limits (corresponding to $k = 10, 25, 50$, and 100) are tested with three regular update schedules (5, 10, and 20 updates), giving 12 variants of the dynamic CARP solver in total. Each variant is tested on three scenarios: "low," "moderate," and "high" dynamism, represented by dynamic CARP instances with degrees of dynamism $\delta = 0.2, 0.5$, and 0.8, respectively. The tabu tenure is set to half the number of tasks, following the choice of the tabu tenure chosen by Brandão and Eglese (2008).

Figure 4.2 shows distributions of percentage deviations over 40 instances given by each variant of the dynamic CARP solver; the percentage deviations are computed from solution costs (total distances) at the end of the planning horizon. The experiment results in Figure 4.2 show little improvement due to increasing the maximum iteration limit across different degrees of dynamism. Furthermore, a two-tailed Wilcoxon signed-rank test was conducted to make a comparison between each pair of maximum iteration limits (with a Bonferroni correction applied, resulting in a significance level of $0.05/6 \approx 0.0083$). Table 4.2 shows that a higher maximum iteration limit does not consistently lead to statistically significant improvement. This suggests that increasing the maximum iteration limit is not a reliable way to improve the performance of the dynamic CARP solver.

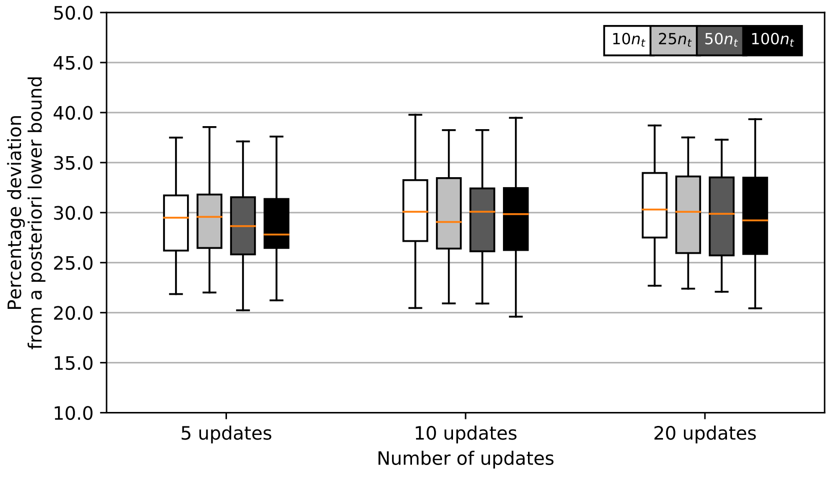
Figure 4.3 shows averages of elapsed time for executing each variant of the dynamic CARP solver in the whole planning horizon (i.e. accumulating computation time from all updates). It can be seen that a higher iteration limit indeed led to a greater amount of elapsed time on average, suggesting that the lack of significant improvement from the use of a higher iteration limit observed in Figure 4.2 is unlikely to be caused by premature termination (i.e. tabu search terminating before reaching the maximum iteration limit due to the absence of admissible[3] solutions).

It was also observed in Table 4.2 that the median of percentage deviations increased when using a higher maximum iteration limit in some cases. This illustrates a striking difference between the static CARP and the dynamic CARP: running an algorithm for the static CARP for more iterations would never lead to a worse solution. In

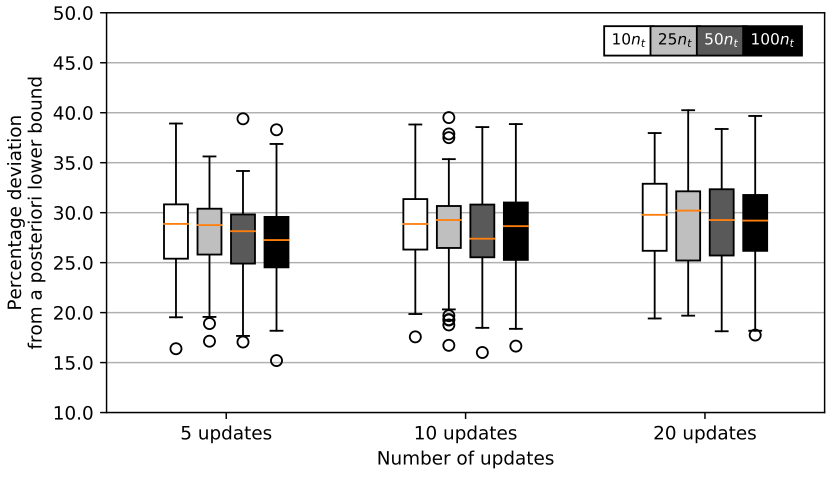
[3]Recall that a solution is said to be admissible if either it is non-tabu or it is tabu but leads to the solution that is better than the current best solution; see Section 3.2



(a) Degree of dynamism = 0.2

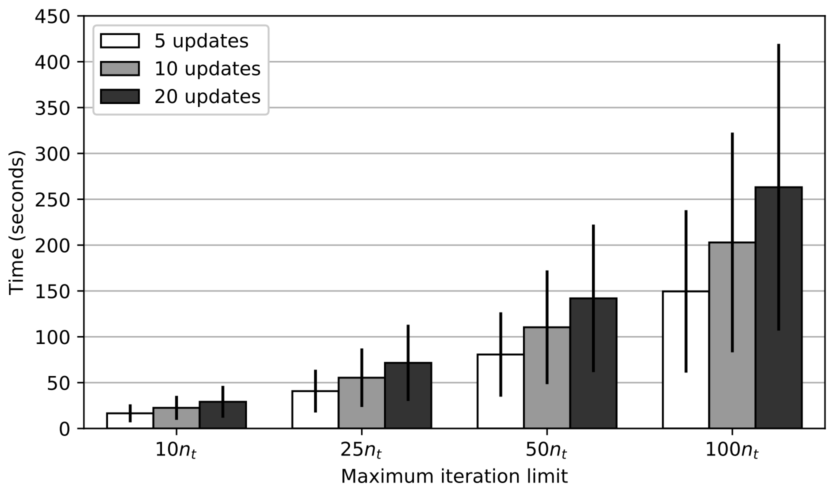


(b) Degree of dynamism = 0.5

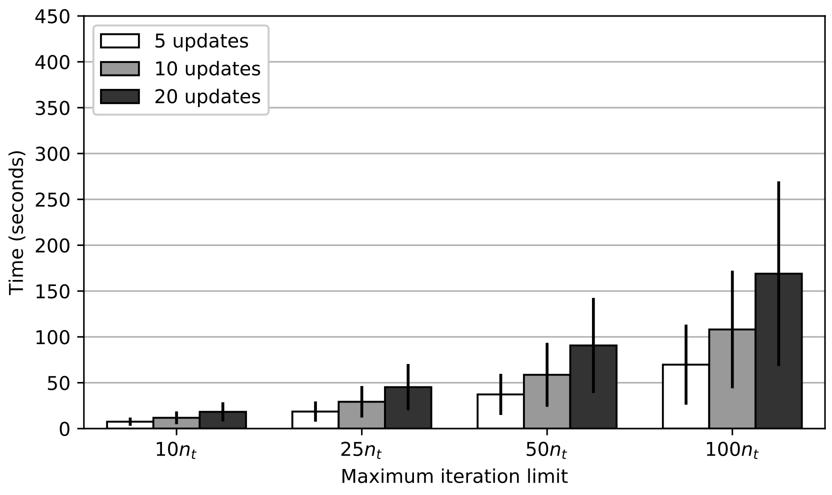


(c) Degree of dynamism = 0.8

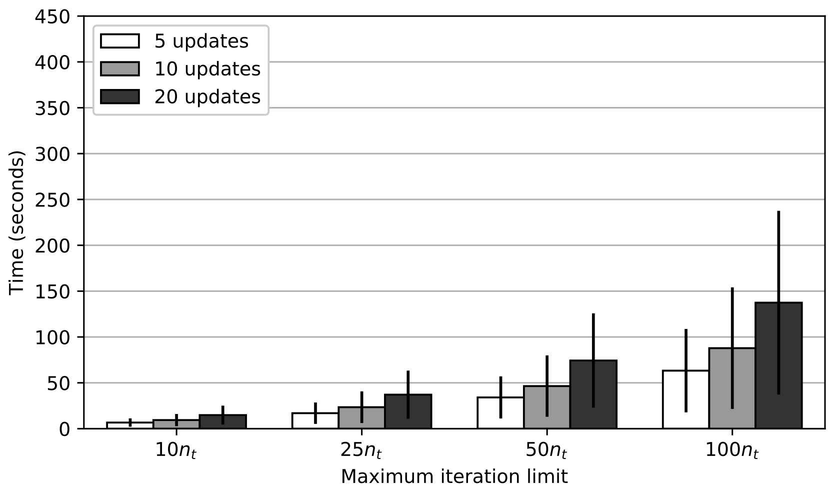
Figure 4.2: Distributions of average percentage deviations from a posteriori lower bounds with respect to total distances given by the dynamic CARP solver with different maximum iteration limits ($10n_t$, $25n_t$, $50n_t$, and $100n_t$, where n_t is the number of tasks) for various degrees of dynamism



(a) Degree of dynamism = 0.2



(b) Degree of dynamism = 0.5



(c) Degree of dynamism = 0.8

Figure 4.3: Average elapsed time taken by each variant of the dynamic CARP solver in the whole planning horizon; black vertical lines show one standard deviation from each side of the averages

Table 4.2: Medians of percentage deviations from a posteriori lower bounds given by the dynamic CARP solver with different maximum iteration limits

Degree of dynamism	Update schedule	Maximum iteration limit			
		$10n_t$	$25n_t$	$50n_t$	$100n_t$
0.2	5 updates	26.5	26.9	26.3	26.1[b]
	10 updates	27.1	26.8	25.6	26.0
	20 updates	26.4	25.6	25.8	26.3[a]
0.5	5 updates	29.5	29.6	28.7[a]	27.8[a]
	10 updates	30.1	29.1	30.1	29.9[a]
	20 updates	30.3	30.1	29.9[a]	29.2[a]
0.8	5 updates	28.9	28.8	28.1[a]	27.3[a,b]
	10 updates	28.9	29.3	27.4[a]	28.7[a,b]
	20 updates	29.8	30.2	29.3	29.2

[a] significantly better than the maximum iteration limit $10n_t$
[b] significantly better than the maximum iteration limit $25n_t$
based on a two-tailed Wilcoxon signed-rank test with a Bonferroni correction (for 6 pairwise comparisons), resulting in a significance level of $0.05/6 \approx 0.0083$

contrast, this can occur in the dynamic CARP due to the existence of changes in the problem. Moreover, as executing tabu search with different numbers of iterations generally leads to different solutions, this leads to different problem states (i.e. sets of tasks to be serviced, vehicles positions and capacities). This means that, even though they are based on the same tabu search algorithm, dynamic CARP solvers with different maximum iteration limits generally encounter different sequences of static CARPs over the planning horizon. This allows the relative performance of difference variants of the dynamic CARP solvers to vary and, in some cases, allows the variant with a higher iteration limit to return a worse solution. This further emphasises the need to devise a way to reliably improve the dynamic CARP solver other than increasing the maximum iteration limit.

4.4.2 Update Schedules

We now turn our attention to the effect of changing the update schedule. For a more comprehensive result, three regular update schedules (with 5, 10, and 20 updates) are tested on a wider range of degrees of dynamism $\delta = 0.1, 0.2, \ldots, 0.9$. The maximum number of iterations for the tabu search algorithm at each update is set to $50n_t$, where n_t is the number of tasks. The experiment results with respect to total distances given by different update schedules are shown in Figure 4.4(a).

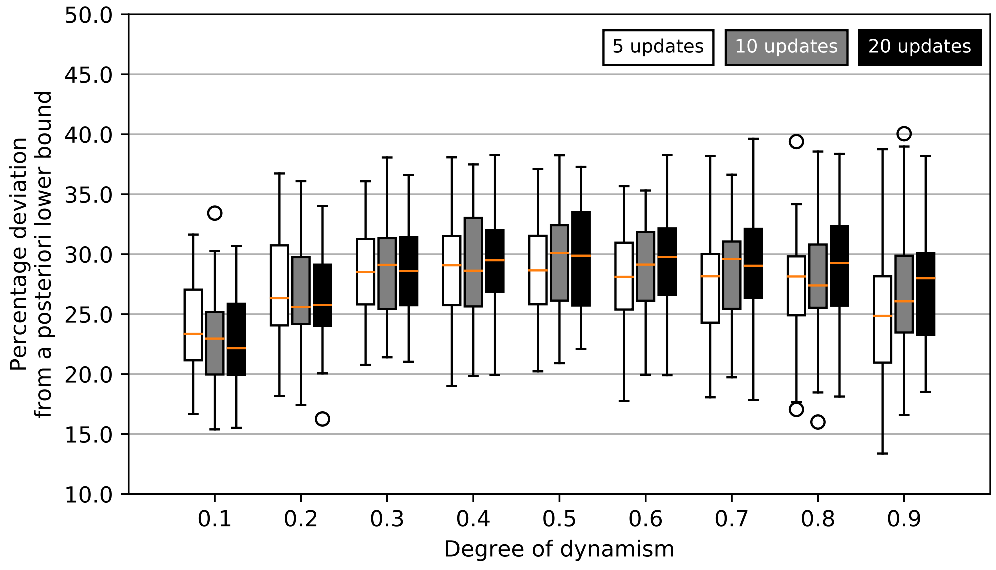
Table 4.3: Medians of percentage deviations from a posteriori lower bounds given by the dynamic CARP solver with different update schedules

Degree of dynamism	Update schedule		
	5 updates	10 updates	20 updates
0.1	23.4	23.0[a]	22.2[a]
0.2	26.3	25.6	25.8
0.3	28.5	29.1	28.6
0.4	29.1	28.6	29.5
0.5	28.7[c]	30.1	29.9
0.6	28.1[b,c]	29.1	29.8
0.7	28.2[b,c]	29.6	29.0
0.8	28.1	27.4[c]	29.3
0.9	24.9[b,c]	26.1	28.0

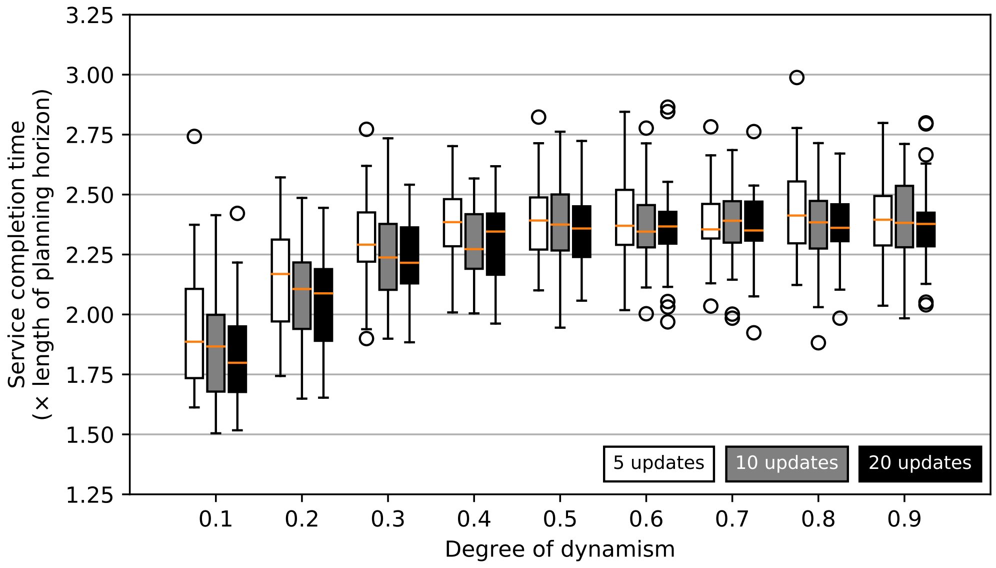
[a] significantly better than the 5-update schedule
[b] significantly better than the 10-update schedule
[c] significantly better than the 20-update schedule
based on a two-tailed Wilcoxon signed-rank test with a Bonferroni correction (for 3 pairwise comparisons), resulting in a significance level of $0.05/3 \approx 0.017$

Also, a two-tailed Wilcoxon signed-rank test was conducted to compare each pair of the update schedules (with a Bonferroni correction, resulting in a significance level of $0.05/3 \approx 0.017$) and the test results are shown in Table 4.3. It was found that the best update schedule with respect to total distance varies with the degree of dynamism. For relatively low degrees of dynamism, a more frequent update schedule tends to perform better. In particular, the 10-update and the 20-update schedules are significantly better than the 5-update schedule on the instances with the degree of dynamism $= 0.1$. As the degree of dynamism increases, the performance of different update schedules becomes more and more similar to each other. Then, for relatively high degrees of dynamism, a less frequent update schedule generally performs better; in some cases, the 5-update schedule is significantly better than the 10-update and the 20-update schedules.

The experiment results in terms of service completion times are shown in Figure 4.4(b). A similar Wilcoxon signed-rank test was also conducted to compare each pair of the update schedules in terms of service completion times, and the test results are shown in Table 4.4. The results suggest that a less frequent update schedule tends to result in a later service completion time, especially when the degree of dynamism is relatively low. In fact, the service completion time given by the 5-update schedule is significantly worse than the 10-update and the 20-update schedules on the instances with degrees of dynamism in the range 0.1 to 0.4. For relatively high



(a)



(b)

Figure 4.4: Distributions of percentage deviations from a posteriori lower bounds with respect to total distances (a) and distributions of service completion times (b) given by different update schedules over 40 instances for each degree of dynamism (0.1, 0.2, . . . , 0.9)

Table 4.4: Medians of service completion times (as multiples of the planning horizon length) given by the dynamic CARP solver with different update schedules

Degree of dynamism	Update schedule		
	5 updates	10 updates	20 updates
0.1	1.89	1.87[a]	1.80[a]
0.2	2.17	2.11[a]	2.09[a]
0.3	2.29	2.24[a]	2.22[a]
0.4	2.38	2.27[a]	2.35[a]
0.5	2.39	2.37	2.36
0.6	2.37	2.35	2.37
0.7	2.35	2.39	2.35
0.8	2.41	2.38[a]	2.36[a]
0.9	2.40	2.38	2.38

[a] significantly better than the 5-update schedule
based on a two-tailed Wilcoxon signed-rank test with a Bonferroni correction (for 3 pairwise comparisons), resulting in a significance level of $0.05/3 \approx 0.017$

degrees of dynamism (≥ 0.5), however, the service completion times for different update schedules are generally similar to each other, and no significant difference between the results of different update schedules were found in most cases.

As we have seen in Figure 4.4(a), updating the solution many times generally leads to relatively poor results for relatively high degrees of dynamism. A possible cause of this is the current way of integrating new tasks, that is, the solution is reconstructed from scratch before tabu search in each update. As a result, what the tabu search algorithm has learnt in each update about the problem (e.g. promising or unpromising sequences of tasks that should be serviced in the same route) is not transferred to future updates. In this case, a less frequent update would allow the solver to collect more information about new tasks for each update, which could then help the algorithm find better solutions. On the other hands, increasing the update frequency results in less information about new tasks in each update, and consequently the solver is more susceptible to making poor decisions when amending the solution in each update due to limited information.

To understand the effect of update schedules observed in the case of relatively low degrees of dynamism, it should be noted that for some of the dynamic CARP instances considered here, some updates are omitted due to the absence of new tasks, especially when the degree of dynamism is low. This is illustrated by Figure 4.5, which shows the number of actual updates, i.e. those in which there exist new tasks, under different update schedules on the dynamic CARP instances considered here

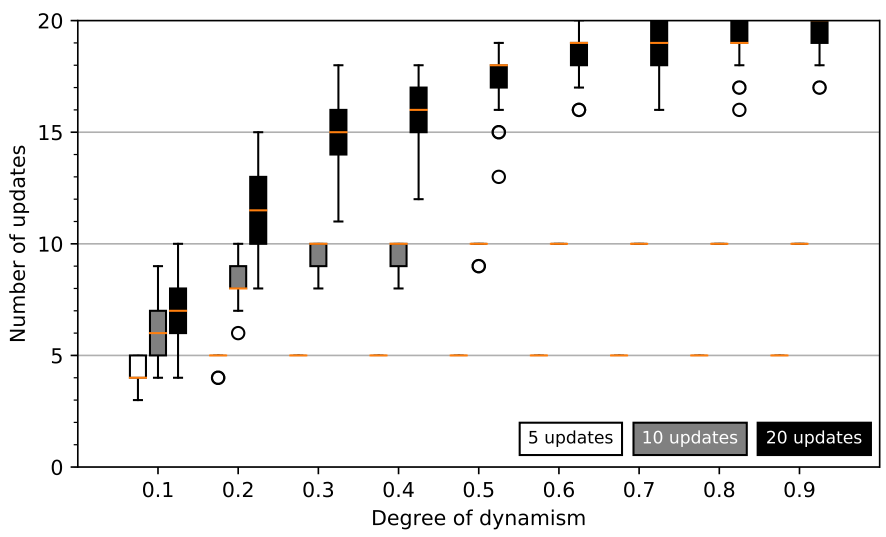


Figure 4.5: The number of updates in which new tasks exist on 40 dynamic CARP instances generated in Section 4.3 for each degree of dynamism (0.1, 0.2, . . . , 0.9)

across different degrees of dynamism. Notice that the difference between the numbers of actual updates for different update schedules is relatively small for low degrees of dynamism. Also notice that fewer and fewer updates are omitted as the degree of dynamism increases. In fact, no updates under the 5-update and the 10-update schedules are omitted for sufficiently large degrees of dynamism.

When the degree of dynamism is low, the solver does not suffer much from updating the solution too frequently even if a frequent update schedule is used; this is due to the absence of new tasks in some updates in the dynamic CARP instances considered here. In fact, a more frequent update schedule means that there are more updates arranged throughout the planning horizon, which allows new tasks to be added to the solution more promptly. The effect of this is particularly evident in Figure 4.4(b): the service tends to be completed at an earlier time under a more frequent update schedule. Furthermore, since more and more parts of the routes would be fixed as vehicles travel along their routes, adding new tasks to the solution at an earlier time would allow more possibilities to amend the solution, hence a greater chance of finding better solutions.

It is worth reminding ourselves at this stage that the above comparison of different update schedules is based on the dynamic CARP solver in which an initial solution in each update is reconstructed from scratch. The next section proposes another way of integrating new tasks to the dynamic CARP solution in each update and investigates whether it is beneficial to retain solutions from previous updates as opposed to solving the problem in each update from scratch.

4.5 An Alternative Method of Integrating New Tasks

In previous sections, as well as in existing work on the dynamic CARP in the literature (Liu et al., 2014b), the problem at each update is solved from scratch. This section proposes a novel concept that aims to retain vehicle routes that have been improved by the dynamic CARP solver throughout the planning horizon: in each update, instead of reconstructing a solution from scratch (the "Reconstruction method"), new tasks are inserted into existing routes one by one in a random order. This alternative method will be called the *Random Insertion* method. The motivation behind this idea is that the solution has been improved by tabu search in previous updates and so would be likely to contain some favourable characteristics (such as certain sequences of tasks), which would be lost if the solution was reconstructed from scratch. The Random Insertion method attempts to integrate new tasks into a solution in such a way that retains most characteristics of the solution given by the previous update.

The Random Insertion method works as follows. First, new tasks are arranged in a random order; they are then added to a given solution in that order by means of a greedy method: that is, each task is added to a given solution in a way that results in the least possible increase in the total distance. To find such a cheapest way to insert a task, let $D(a, b)$ denote the shortest distance from the head of task a to the tail of task b for any tasks a, b. For a route $R = (v_s, a_1, \dots, a_n, v_0)$ and a task a', the cost (or more precisely, change in the total distance of the solution) incurred by inserting task a' into route R between tasks a_j and a_{j+1} for some $j \in \{0, 1, \dots, n\}$ is equal to

$$D(a_j, a') + D(a', a_{j+1}) - D(a_j, a_{j+1}), \tag{4.4}$$

where, for ease of notation, $a_0 = (v_s, v_s)$ and $a_{n+1} = (v_0, v_0)$. For each new task, all routes with sufficient capacities are considered, and the task is inserted into the route that results in the cheapest insertion cost according to the expression (4.4)[4]. If there are many routes into which the task can be inserted with the cheapest cost, then one of them is chosen randomly. If there are no routes with sufficient capacities, then a new route is created for the task that is being considered. A pseudocode for the Random Insertion method is given in 5. The Random Insertion method will be

[4]Note that the cost of servicing task a' is omitted in the expression (4.4) because it is independent of the position of insertion and thus has no effect on the best insertion position.

implemented in place of the Reconstruction method in the dynamic CARP solver (see Lines 11 and 12 in Algorithm 4).

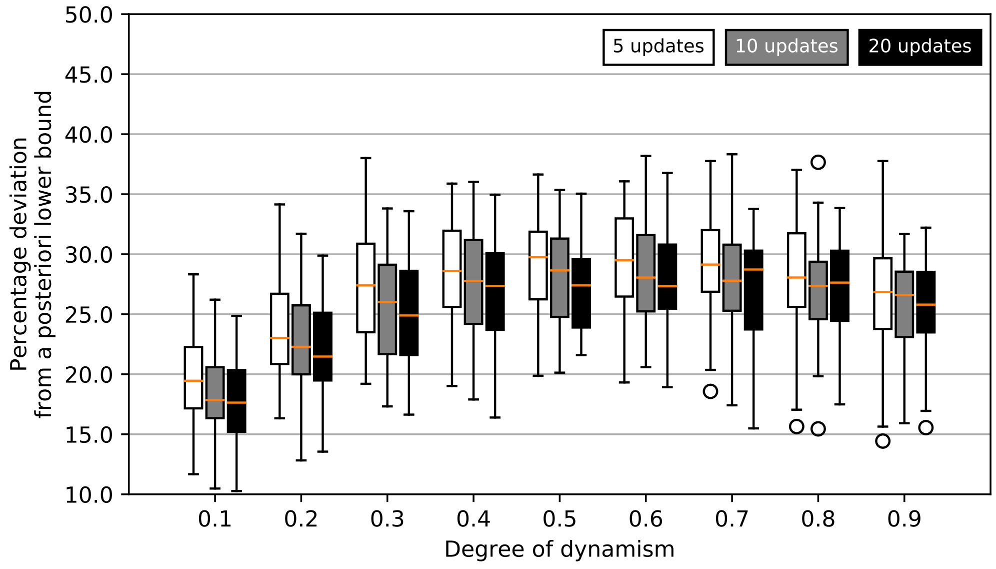
Algorithm 5 The Random Insertion method

```
1: **given** a set of new tasks $\mathcal{T}$, and a set of routes $S$
2: **for** each task $t \in \mathcal{T}$ (in a random order) **do**
3:     **if** there exists a route with sufficient remaining capacity to service $t$ **then**
4:         insert $t$ into one of the routes in $S$ that incurs the cheapest cost (if there
           is more than one such route, choose one of them randomly)
5:     **else**
6:         add $t$ to a new route
```

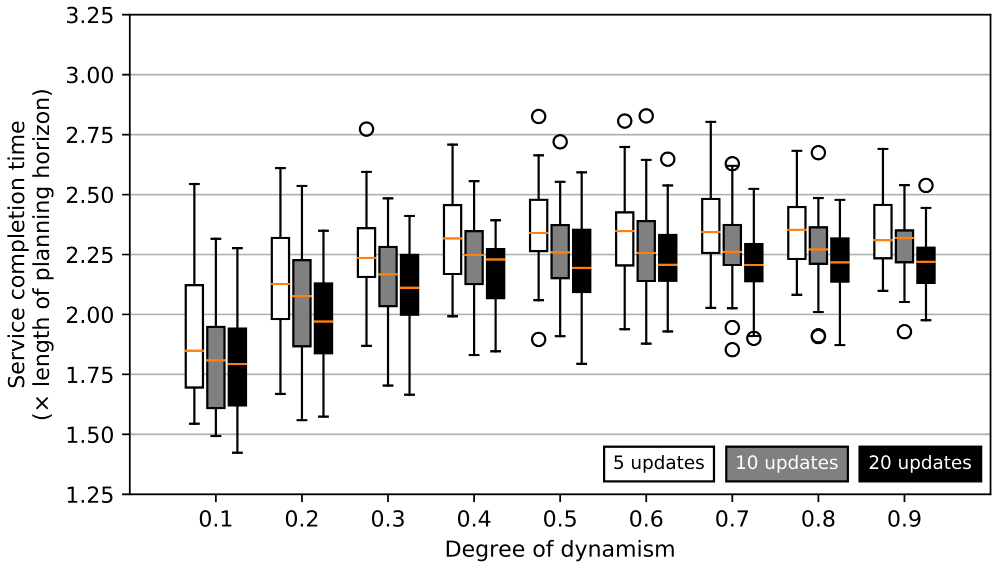
4.5.1 Computational Results

We now look at the performance of the dynamic CARP solver with different update schedules when the Random Insertion is implemented in place of the Reconstruction method. Figure 4.6(a) and Figure 4.6(b) show the performance of the dynamic CARP solver using the Random Insertion method with different update schedules based on two measures, namely total distances and service completion times, respectively. The difference between each pair of update schedules is also analysed by means of a two-tailed Wilcoxon signed-rank test with a Bonferroni correction applied, resulting in a significance level of $0.05/3 \approx 0.017$, and the test results with respect to total distances and service completion times are shown in Table 4.5 and Table 4.6, respectively.

The experiment results show that when the Random Insertion is implemented in place of the Reconstruction method, a more frequent update schedule generally leads to better solutions across different degrees of dynamism. This is different from the results with the Reconstruction method seen previously in Section 4.4.2, where the best update schedule varies with the degree of dynamism. A possible reason for this is that the Random Insertion method allows the solver to retain a solution that has been improved from previous updates. This means that an initial solution in each update has a higher chance of already containing promising features (such as certain sequences of tasks to be serviced in the same route). Consequently, there is not as much need to collect a lot of information about new tasks before each update in order to obtain high quality solutions. In this case, the benefit of accumulating information about new tasks in each update appears to be less prominent than the benefit of adding new tasks to the solution promptly. This is particularly evident when comparing the service completion times for the Reconstruction method and



(a)



(b)

Figure 4.6: Distributions of percentage deviations from a posteriori lower bounds with respect to total distances (a) and distributions of service completion times (b) given by different update schedules with the Random Insertion method over 40 instances for each degree of dynamism (0.1, 0.2, . . . , 0.9)

Table 4.5: Medians of percentage deviations from a posteriori lower bounds given by the dynamic CARP solver with the Random Insertion method and different update schedules

Degree of dynamism	Update schedule		
	5 updates	10 updates	20 updates
0.1	19.5	17.8[a]	17.6[a,b]
0.2	23.0	22.3[a]	21.5[a,b]
0.3	27.4	26.0[a]	24.9[a,b]
0.4	28.6	27.8	27.3[a,b]
0.5	29.7	28.7[a]	27.4[a]
0.6	29.5	28.1	27.3[a]
0.7	29.1	27.8[a]	28.7[a]
0.8	28.1	27.4[a]	27.6[a]
0.9	26.8	26.6	25.8

[a] significantly better than the 5-update schedule
[b] significantly better than the 10-update schedule
based on a two-tailed Wilcoxon signed-rank test with a Bonferroni correction (for 3 pairwise comparisons), resulting in a significance level of $0.05/3 \approx 0.017$

Table 4.6: Medians of service completion times (as multiples of the planning horizon length) given by the dynamic CARP solver with the Random Insertion method and different update schedules

Degree of dynamism	Update schedule		
	5 updates	10 updates	20 updates
0.1	1.85	1.81[a]	1.79[a]
0.2	2.13	2.08[a]	1.97[a,b]
0.3	2.24	2.17[a]	2.11[a,b]
0.4	2.32	2.25[a]	2.23[a,b]
0.5	2.34	2.26[a]	2.19[a,b]
0.6	2.35	2.26[a]	2.21[a,b]
0.7	2.34	2.26[a]	2.21[a,b]
0.8	2.35	2.27[a]	2.22[a,b]
0.9	2.31	2.32[a]	2.22[a,b]

[a] significantly better than the 5-update schedule
[b] significantly better than the 10-update schedule
based on a two-tailed Wilcoxon signed-rank test with a Bonferroni correction (for 3 pairwise comparisons), resulting in a significance level of $0.05/3 \approx 0.017$

the Random Insertion method; see Figure B.7.[5]

So far the update schedules have been compared based on the same method of integrating new tasks. It is also interesting to compare 6 variants of the dynamic CARP solver (3 update schedules $\times$ 2 methods of integrating new tasks) all together; in particular, this would allow us to clearly see how the performance of the dynamic CARP solver is affected by the method of integrating new tasks. For ease of reference, Figures B.6 and B.7 in Appendix B show the experiment results (with respect to total distances and service completion times, respectively) for both methods of integrating new tasks on the same plot. It was found that the Random Insertion generally gives better solutions than the Reconstruction method. In fact, using the Random Insertion with 20 updates is the most promising variant of the dynamic CARP solver; it is significantly better than the other variants in most cases (see Tables B.1 and B.2 in Appendix B). This further highlights the benefits of retaining the solution from previous updates compared with solving the problem at each update from scratch.

4.6 Conclusion

This chapter concerns a heuristic algorithm for solving the dynamic CARP. It begins by describing main components of a dynamic CARP solver, including an update schedule, how the current state of the problem can be determined, a method of integrating new tasks into the solution, and a heuristic algorithm for improving the solution, which in this thesis is based on tabu search. The purpose of this chapter is to investigate how the performance of a heuristic algorithm for solving the dynamic CARP can be affected by adjusting its configuration. In particular, here we consider adjusting 3 components of the solver: a maximum iteration limit for tabu search in each update, the frequency of solution updates, and a method of integrating new tasks to an existing solution. A novel analysis is conducted to compare several options of these components and to investigate how each of the components could affect the solution quality with respect to total distance and service completion time.

Regarding the maximum iteration limit, experiment results show that increasing the maximum iteration limit for tabu search in each update yields little improvement. Moreover, a larger maximum iteration limit could sometimes give worse results. This suggests that to consistently achieve a better solution in the dynamic CARP, it is not sufficient to rely solely on running the tabu search algorithm at each update for

[5]Figure B.7 contains the same information as Figure 4.4(b) and Figure 4.6(b) but is organised in such a way that it is easier to compare the results from different ways of integrating new tasks.

more iterations. This suggests the need to improve the dynamic CARP solver by other means.

Two ways of amending the dynamic CARP are then investigated: adjusting the frequency of solution updates and the way of integrating new tasks to the solution in each update. Here we consider 3 regular update schedules (with 5, 10, and 20 update) and 2 methods of integrating new tasks. In the existing literature, the problem at each update is usually solved from scratch, which we call here the Reconstruction method. Based on the intuition that solutions from previous updates are likely to contain some promising features as they have undergone the tabu search process, an alternative way of integrating new tasks is proposed, namely the Random Insertion method, which retains a solution from a preceding update and inserts new tasks to the solution in a greedy way before it is further improved by tabu search. Computational results show that the effect of adjusting the frequency of solution updates and the way of integrating new tasks to the solution in each update varies with the degree of dynamism, i.e. the ratio of the number of dynamic tasks (known after vehicles leave the depot at the beginning of the planning horizon) to the number of all tasks in the whole the planning horizon.

More precisely, for relatively low degrees of dynamism (up to 0.4 for the instances considered here), a more frequent update schedule tends to give better results, regardless of the method of integrating new tasks. Nevertheless, the Random Insertion method yields more promising results than the Reconstruction method. In contrast, for relatively high degrees of dynamism (at least 0.5 for the instances considered here), the performance of different update schedules depends on the method of integrating new tasks. With the Reconstruction method, a less frequent update schedule tends to give better results. In contrast, with the Random Insertion method, a more frequent update schedule tends to give better results.

Among all variants of the dynamic CARP solver considered, the Random Insertion method with 20 updates give the best results; it is significantly better than the other variants in many cases (see Tables B.1 and B.2). This highlights the benefit of retaining solutions from previous updates as opposed to solving the problem at each update from scratch.

In the next chapter, we will attempt to improve dynamic CARP solutions by means of waiting strategies. The goal will be to further reduce total distance while avoiding an overly large increase in service completion time.

Chapter 5

Waiting Strategies

5.1 Introduction

Recall that the dynamic CARP can be viewed as a sequence of static CARPs, each of which occurs at a certain time in the planning horizon. Given that the dynamic CARP involves the notion of time, focussing solely on developing an algorithm for finding a solution to the static CARP in each update would limit the capability of a route planner to amend vehicle routes for the overarching dynamic CARP. In Chapter 4, we attempted to exploit the notion of time in the dynamic CARP by means of varying the frequency of solution updates. This chapter further explores ways of exploiting the notion of time and anticipating changes that can occur at a later time to improve the dynamic CARP solution.

One way of exploiting the notion of time is to instruct vehicles to wait and stand by at certain locations, especially when they finish servicing all tasks that have been assigned to them. Without waiting instructions, the problem at each update would largely depend on the solution from the previous update. In particular, the positions of vehicles at a given update are solely determined by that solution. The dynamic CARP solver in Chapter 4 amends the solution in each update using only available information in that update, and thus it cannot be guaranteed that the positions of vehicles, or other solution features, will still be favourable in the current state of the problem. Utilising waiting instructions gives a route planner more control over the locations at which the vehicles will be, or the current state of the problem in general, at the upcoming update. This allows more possibilities of amending the solution and potentially improves effectiveness of route planning in the dynamic CARP.

Section 5.2 discusses possible undesirable features in the solution without waiting and proposes a waiting strategy that can prevent such features. Section 5.3 proposes a way to amend the waiting strategy by specifying a rule to decide which vehicle should wait based on its current capacity. Section 5.4 proposes another waiting strategy which is focussed specifically on the positions of vehicles. Section 5.5 introduces the idea of employing extra routes as a way of anticipating new tasks in the future. The conclusions of this chapter are given in Section 5.6.

5.2 Instructing Vehicles to Wait at the End of Last Tasks

In Chapter 4, it was implicitly assumed that once vehicles depart from the depot, they travel continuously - that is, without stops in the middle of their routes - until returning to the depot. This would be an appropriate assumption for the static CARP since letting vehicles wait or stand by somewhere would make no difference to the total distance of a solution. In contrast, for the dynamic CARP, instructing vehicles to wait could help prepare them for changes that may occur in the problem at a later time. For example, it was found that in some solutions given by the dynamic CARP solver in Chapter 4, there existed a vehicle that returned to the depot before the end of the planning horizon even though it still had some remaining capacity and thus could have serviced more demand. An example of a solution where this situation occurred is given in Figure 5.1, where black lines with crosses show the time at which each route starts and ends, and grey bars show the amount of demands serviced in each route. In this example, notice that Route 1 returned to the depot before the end of the planning horizon (slightly before time $0.6T$, where T is the length of the planning horizon), but the total demand serviced in this route was about 70% of the capacity, so it still had some remaining capacity to receive more demand. A waste of remaining capacity could have been prevented by instructing vehicles to wait and be on standby at the end vertices of their last tasks. This would allow a route planner to assign additional tasks to vehicles in an upcoming update instead of letting them head towards the depot straight away when they have no tasks to service. Reducing a waste of remaining capacity would decrease the amount of demands that later vehicles need to service and potentially reduce the need to employ more vehicles at a later time, thereby saving some distance travelled.

Besides a waste of remaining capacity, another possible drawback of vehicles returning

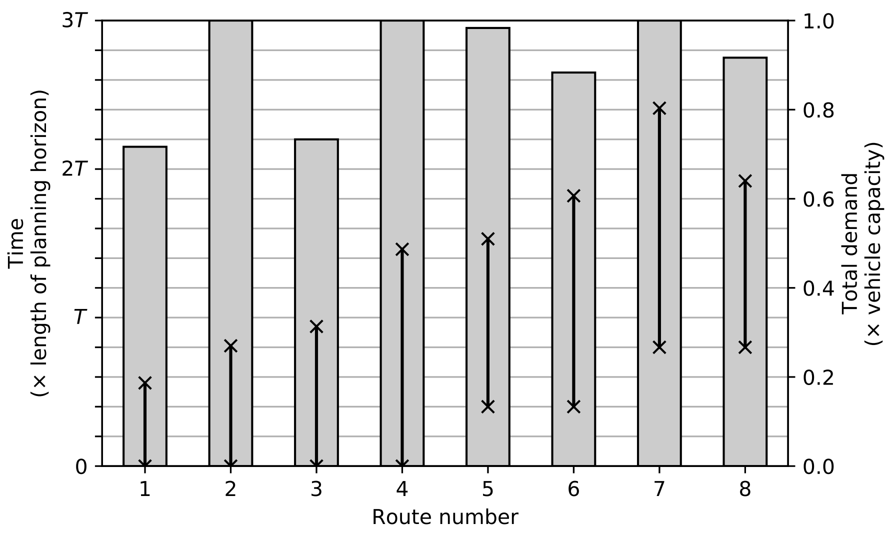


Figure 5.1: Start times, finish times, and total demands in each route in one solution for the C04-d50-1 instance; T denotes the length of the planning horizon; the lines with crosses show the period of time in which a vehicle corresponding to each route travels since leaving the depot until returning to the depot; the bars show the total demand of tasks serviced in each route.

to the depot immediately after their last tasks is that if they have sufficient remaining capacity, they could receive an additional task while heading towards the depot, which would force them to divert away from their originally planned paths. Instructing a vehicle to wait at the end vertex of the last serviced task would allow the vehicle to travel directly (i.e. along a shortest path) from the last serviced task to a newly assigned task, thereby saving some distance. A similar situation in which the waiting instruction would help save some distance is when a vehicle already returns to the depot but later departs from the depot again to service an additional task. These situations are illustrated by Figure 5.2.

Nevertheless, instructing a vehicle to wait means that the vehicle would be idle for some time, which might delay the completion of the overall service. In this section, we will investigate the trade-off between total distance and service completion time as a result of the waiting strategy. More precisely, the waiting strategy that will be considered works as follows: At each update (except at the end of the planning horizon), check whether each vehicle will be "idle" at the next update, i.e. it will finish servicing the last task by the upcoming update, according to the solution returned by the tabu search algorithm in the current update. For each idle vehicle, instruct it to wait at the end vertex of the last task after servicing it until the upcoming update. A pseudocode of the dynamic CARP solver with waiting instructions is shown in Algorithm 6; the solver makes a decision about waiting in Lines 4 and 15.

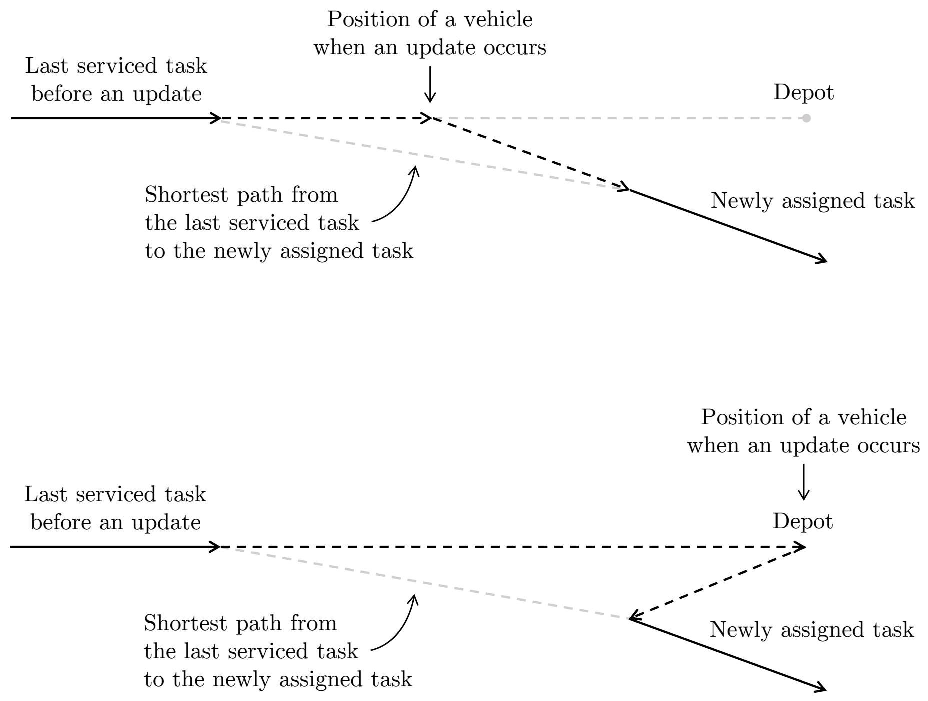

Position of a vehicle when an update occurs
Last serviced task before an update
Depot
Shortest path from the last serviced task to the newly assigned task
Newly assigned task
Position of a vehicle when an update occurs
Last serviced task before an update
Depot
Shortest path from the last serviced task to the newly assigned task
Newly assigned task

Figure 5.2: A diversion after the last serviced task as a result of assigning an additional task to a vehicle while it is heading towards the depot (see the top figure) or after it returns to the depot (see the bottom figure). A path along which the vehicle travels without waiting is shown in black. Alternative paths in which the vehicle could have travelled are shown in grey. Deadheading paths (i.e. travelling without servicing) are shown in dashed lines.

Algorithm 6 Configuration of the dynamic CARP solver with waiting

```
 1: given max_time (i.e. the length of the planning horizon), and a rule for deciding
    when to perform solution updates (see Section 4.2.1)
 2: construct an initial solution $S$ by the Path Scanning algorithm
 3: apply the tabu search algorithm (from Chapter 3) to $S$
 4: implement a waiting strategy (Algorithm 7)
 5: set time = 0
 6: while time < max_time do
 7:     increase time by 1 unit
 8:     if time is an update time then
 9:         let $\mathcal{T}$ be the set of new tasks that appear since last update
10:         if there exist new tasks then
11:             identify the fixed part and the starting vertex of each route for the
                current update (see Section 4.2.2)
12:             add the new tasks in $\mathcal{T}$ to $S$ by the Random Insertion method (see
                Algorithm 5)
13:             apply the tabu search algorithm (from Chapter 3) to $S$
14:         if this update is not the last one then
15:             implement a waiting strategy (Algorithm 7)
```

Algorithm 7 Instructing vehicles to wait at the end of last tasks

```
1: given a set of routes $S$, the time of the next update
2: for each route in $S$ do
3:     if the service on last task will finish before the next update then
4:         instruct the corresponding vehicle to wait at the end of its last task until
           the next update
```

5.2.1 Computational Results

The waiting strategy is tested with the dynamic CARP solver from Chapter 4 in which the method of integrating new tasks is the Random Insertion method and the maximum iteration limit for tabu search is $50n_t$, where n_t is the number of tasks at each update. All 3 regular update schedules (with 5, 10, and 20 updates) are considered. 40 dynamic CARP instances for each degree of dynamism that were generated and used in Chapter 4 are also used in this experiment (and the rest of this chapter); recall that in these instances, the length of the planning horizon is 500 units time, and the vehicle speed is one unit distance per unit time. Due to its stochastic nature, each variant of the dynamic CARP solver is run on each instance 20 times, and its performance on that instance is assessed based on average results over 20 runs. The performance of the dynamic CARP solver without and with the waiting strategy is displayed in Figures 5.3 and 5.4, concerning percentage deviations from a posteriori lower bounds[1] and service completion time, respectively. The computational results show that the waiting strategy generally makes little difference to the solution quality regarding both total distance and service completion time.

We now investigate why the waiting strategy has only a small effect on the solution quality. First, we examine how the waiting strategy affects the time at which routes are constructed throughout the planning horizon. Figure 5.5 shows the number of runs (out of 800 runs for each degree of dynamism) in which the dynamic CARP solver returns solutions with a given number of routes. Here, only the results given by the solver with 20 updates are shown for conciseness; the results corresponding to 5 and 10 updates are similar and shown in Figure B.8 in Appendix B. It can be seen that the waiting strategy encourages the solver to give a solution with a smaller number of routes. This suggests that the waiting strategy helps reduce the need to construct additional routes.

Figure 5.6 shows the proportion of routes with a range of demands of serviced tasks (up to half the vehicle capacity) in final solutions, i.e. the solutions at the end of the planning horizon, from all runs given by the dynamic CARP solver without and with the waiting strategy; here we only show the results of the dynamic CARP solver with 20 updates. The results of the dynamic CARP with 5 and 10 updates are similar and shown respectively in Figures B.9 and B.10 in Appendix B. It can be seen that the proportion of routes that are less than or exactly half full (i.e. the total amount of demands of serviced tasks in a given route is no greater than half the

[1]Recall that a percentage deviation from a posteriori lower bound is computed as follows: percentage deviation = $\left(\frac{\text{solution cost}-\text{a posteriori lower bound}}{\text{a posteriori lower bound}}\right) \times 100$.

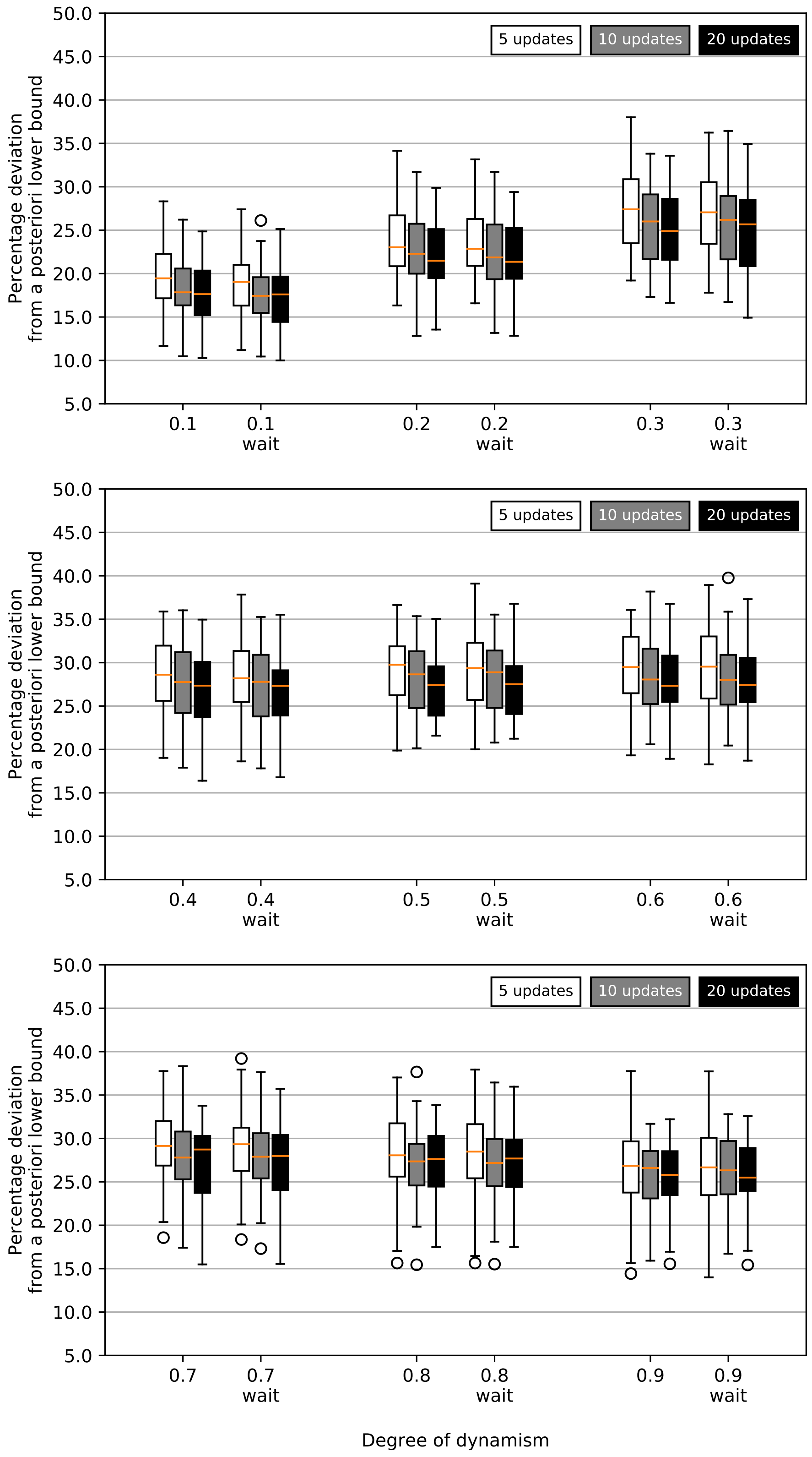


Figure 5.3: Distributions of percentage deviations from a posteriori lower bounds with respect to total distances on 40 dynamic CARP instances for each degree of dynamism (0.1, 0.2, . . . , 0.9) given by the dynamic CARP solver without and with the “waiting at the end of last task” strategy

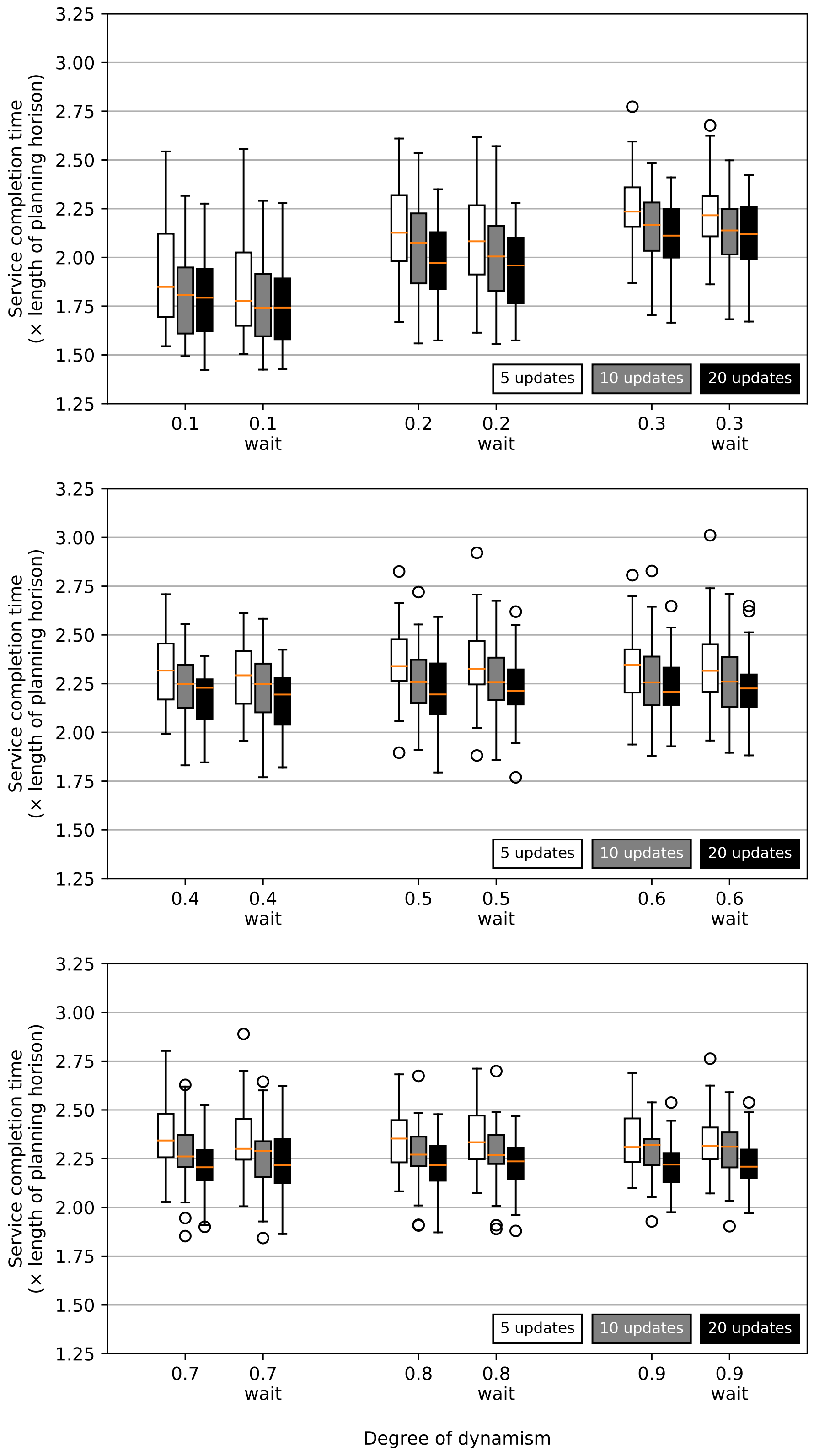
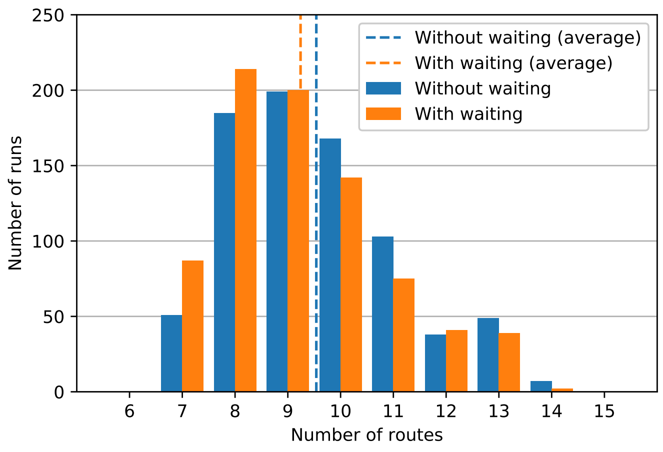
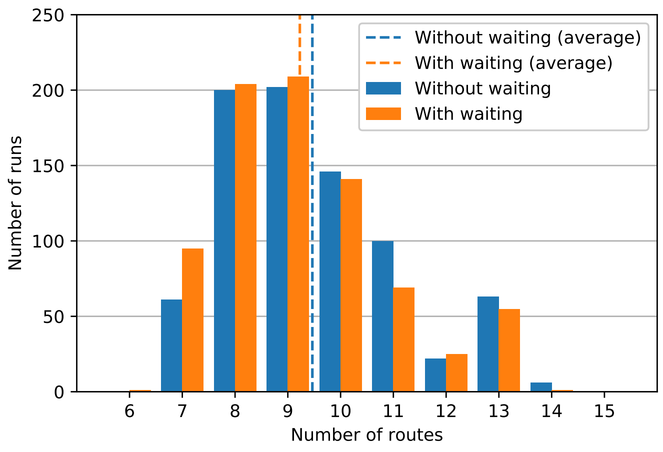


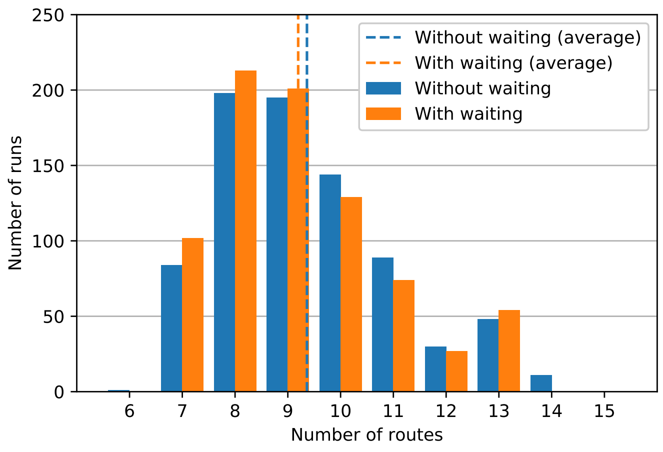
Figure 5.4: Distributions of service completion times on 40 dynamic CARP instances for each degree of dynamism (0.1, 0.2, . . . , 0.9) given by the dynamic CARP solver without and with the "waiting at the end of last task" strategy



(a) Degree of dynamism = 0.2



(b) Degree of dynamism = 0.5



(c) Degree of dynamism = 0.8

Figure 5.5: The number of runs in which the dynamic CARP solver (with 20 updates) returns solutions with a given number of routes from 800 runs (40 instances × 20 runs) for each degree of dynamism; a vertical dashed line shows an average number of routes of the solutions over 800 runs

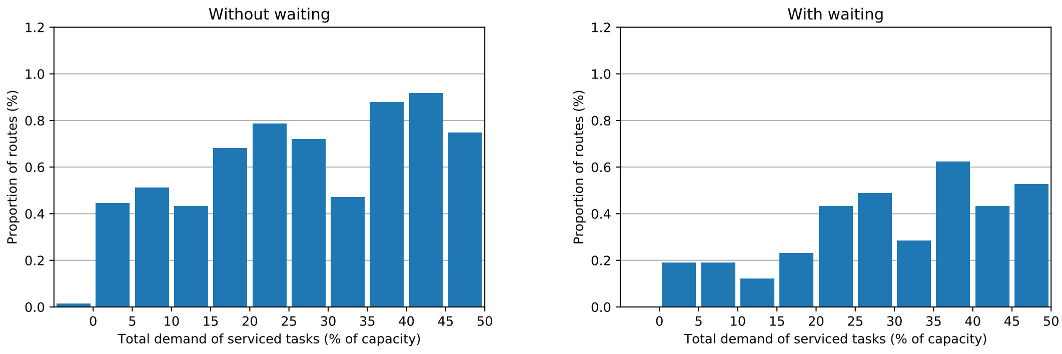
vehicle capacity) decreases when the waiting strategy is implemented. Also notice that among solutions given by the solver without waiting, there are some routes with zero demands[2] (illustrated by the leftmost bars, i.e. to the left of the '0' tick on the x-axis in the histograms), but no such routes exist when the waiting strategy is implemented. This illustrates that the waiting strategy encourages vehicles to service a greater amount of demands. In other words, it helps reduce a waste of vehicle capacity.

We have seen that the waiting strategy indeed has an effect on the construction of routes over the planning horizon: it reduces the need to construct additional routes and helps avoid a waste of remaining capacity. However, this does not seem sufficient to effectively decrease the total distance, as we saw in Figures 5.3 and 5.4. To better understand why the waiting strategy does not lead to noticeable improvement, we further investigate how the waiting strategy was utilised in the computational results. In particular, we examine which vehicles are more likely to be instructed to wait than the others. Histograms in Figure 5.7 display the total number of times vehicles are instructed to wait in all 2,400 runs for each degree of dynamism (40 dynamic CARP instances $\times$ 20 runs $\times$ 3 update schedules). The results are grouped according to "route demands", that is, the amount of demands in the vehicles' routes at the time they were instructed to wait. Figure 5.7 shows that for all degrees of dynamism, most waiting instructions are given to vehicles that are nearly full (in all instances in this experiment, the capacity is 300). Since nearly full vehicles are unlikely to receive additional tasks, instructing those vehicles to wait would generally make little difference to the solution. This could be a cause of the marginal difference made by the waiting strategy in the computational results seen in Figures 5.3 and 5.4.

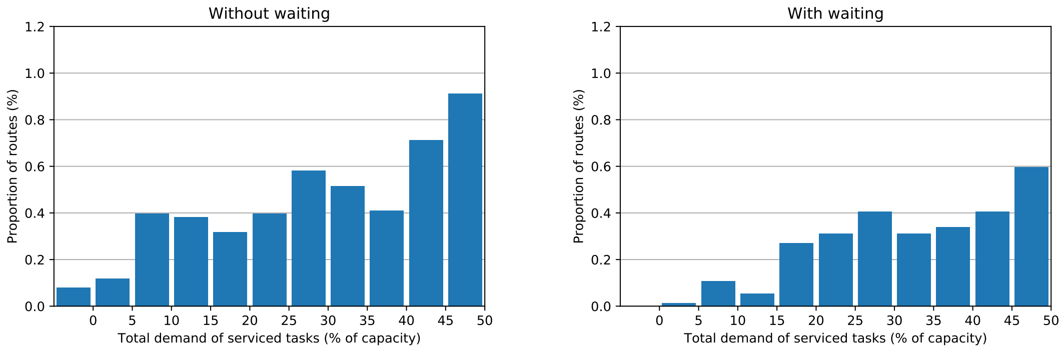
5.3 Waiting Thresholds

If a vehicle is nearly full, it may be unlikely to receive additional tasks, so instructing the vehicle to wait would cause it to return to the depot later than necessary. Omitting the waiting instruction would allow the vehicle to return to the depot sooner and help reduce the service completion time. In fact, it can be beneficial for a nearly full vehicle to go back to the depot and restore its capacity, thus getting ready to service new tasks. To explicitly determine which vehicle is "nearly full" and which should be instructed to wait, a *waiting threshold* $\omega \in [0, 1]$ is introduced:

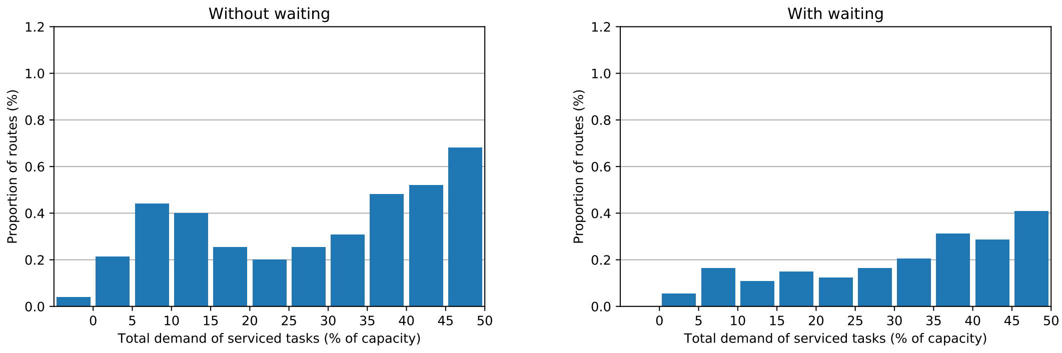
[2]This can happen when a solution update moves all tasks from a certain route, especially those with very few tasks, to the others before the corresponding vehicle reaches its first task, and so the vehicle instead heads towards the depot and ends its route there.



(a) Degree of dynamism = 0.2

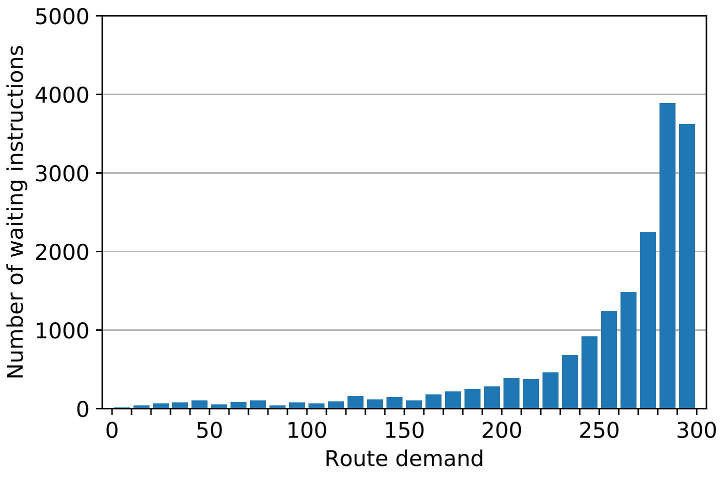


(b) Degree of dynamism = 0.5

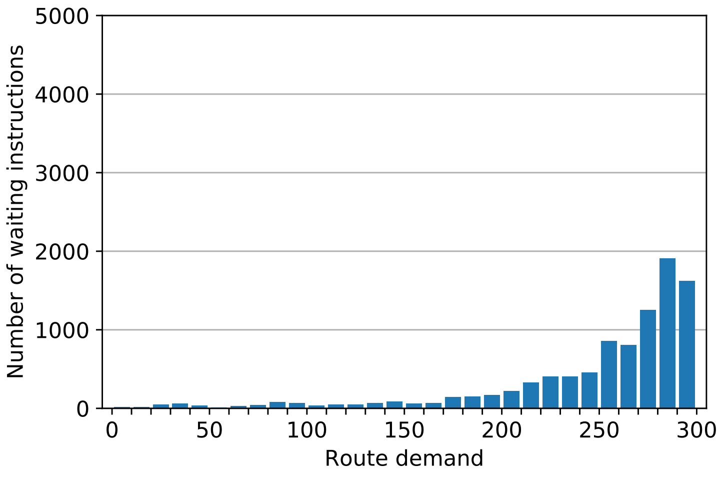


(c) Degree of dynamism = 0.8

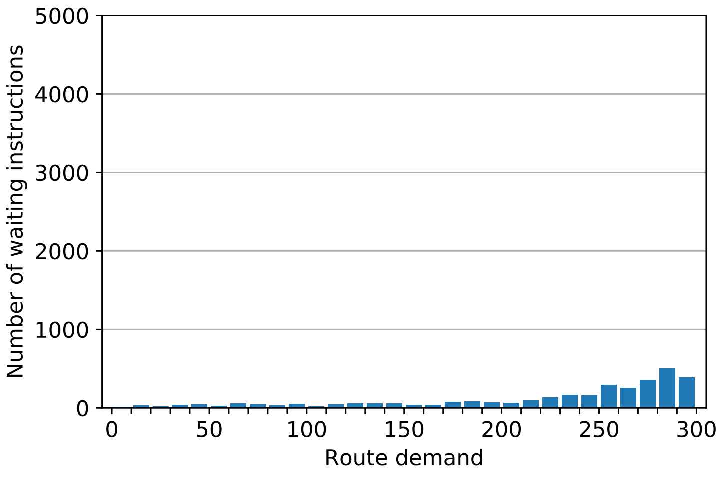
Figure 5.6: Histograms showing the proportion of routes with a range of demands of serviced tasks in solutions at the end of the planning horizon given by the dynamic CARP solver (with 20 updates) without and with the waiting strategy over 800 runs (40 instances × 20 runs) for each degree of dynamism



(a) Degree of dynamism = 0.2



(b) Degree of dynamism = 0.5



(c) Degree of dynamism = 0.8

Figure 5.7: The total number of waiting instructions over 2,400 runs (40 dynamic CARP instances × 20 runs × 3 update schedules) grouped by the amount of demand in vehicles' routes at the time of being instructed to wait for each degree of dynamism

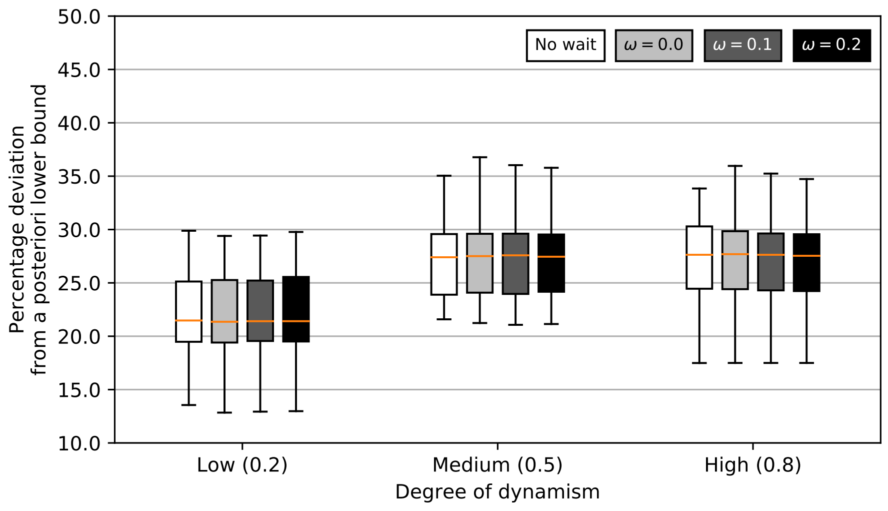
a vehicle is instructed to wait if its remaining capacity is at least $\omega \times Q$, where Q is the vehicle capacity. In particular, $\omega = 0$ means that all vehicles are instructed to wait regardless of their remaining capacities, and the waiting threshold $\omega = 1$ would correspond to no waiting and always letting vehicles head towards the depot after servicing their last tasks. A pseudocode of the waiting strategy with a waiting threshold is given in Algorithm 8.

Algorithm 8 Instructing vehicles to wait at the end of last tasks with a waiting threshold

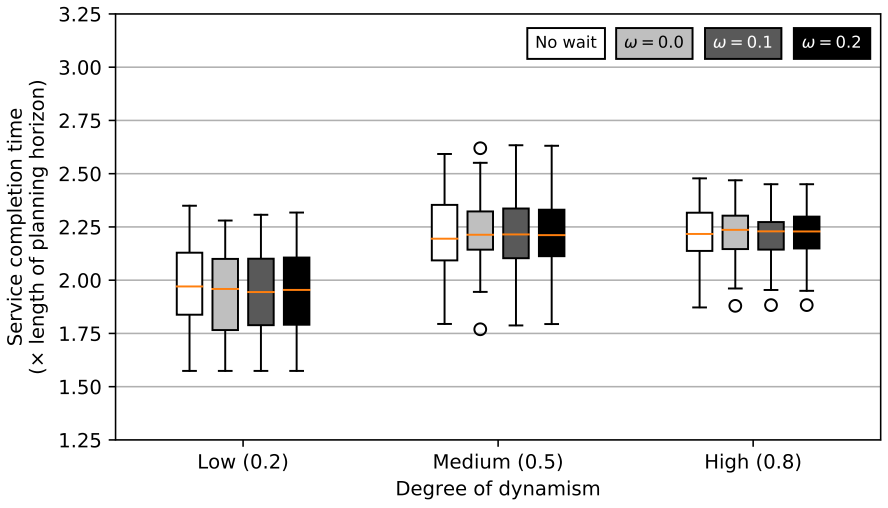
```
1: given a set of routes $S$, the time of the next update, waiting threshold $\omega \in [0, 1]$,
   vehicle capacity $Q$
2: for each route in $S$ do
3:     if the remaining capacity of the route is no less than $\omega \times Q$ then
4:         if the service on last task will finish before the next update then
5:             instruct the corresponding vehicle to wait at the end of its last task
               until the next update
```

A computational experiment is conducted to compare the effect of the waiting strategy different waiting thresholds: $\omega = 0.0, 0.1$ and 0.2. The dynamic CARP solver used in this experiment involves 20 updates and the same configuration as in the previous experiment (the method of integrating new tasks being the Random Insertion method, and the maximum iteration limit for tabu search in each update being $50n_t$, where n_t is the number of tasks in each update). Different waiting thresholds are tested on the instances with "low," "moderate," and "high" dynamism, represented by the degrees of dynamism 0.2, 0.5, and 0.8, respectively. Computational results in Figure 5.8 suggest that varying the waiting threshold generally makes little difference to the solution quality with respect to both total distance and service completion time. Furthermore, as shown in Table 5.1, no statistically significant difference is found between the results given by different waiting thresholds in most cases; the only exception is that using the waiting strategy with the waiting threshold $\omega = 0.0, 0.1$ or 0.2 leads to significant reduction in service completion time on the instances with a low degree of dynamism (0.2), although the reduction is marginal. One possible reason for a small effect of adjusting the waiting threshold is that vehicles are rarely given instructions to wait and increasing the waiting threshold further reduces the chance of doing so.

The likelihood of vehicles being instructed to wait is investigated and displayed in Table 5.2. The column n_{all} shows the total number of all "active" vehicles in each update except at the end of the planning horizon in all 800 runs (40 instances $\times$ 20 runs). Here, an "active" vehicle means it has not returned to the depot at a given update. Notice that n_{all} is also equal to the maximum possible number of



(a)



(b)

Figure 5.8: Distributions of percentage deviations from a posteriori lower bounds with respect to total distances (a) and distributions of service completion times (b) over 40 dynamic CARP instances given by the waiting strategy with waiting thresholds $\omega = 0.0$, 0.1, and 0.2

Table 5.1: Medians of percentage deviations from a posteriori lower bounds with respect to total distances and medians of service completion times (as multiples of the length of the planning horizon) over 40 dynamic CARP instances given by the waiting strategy with waiting thresholds $\omega = 0.0$, 0.1, and 0.2

Degree of dynamism	Percentage deviation from a posteriori lower bound (%)				Service completion time ($\times$ planning horizon length)			
	Without waiting	With waiting			Without waiting	With waiting		
		$\omega = 0.0$	$\omega = 0.1$	$\omega = 0.2$		$\omega = 0.0$	$\omega = 0.1$	$\omega = 0.2$
Low (0.2)	21.47	21.36	21.41	21.41	1.97	1.96*	1.94*	1.95*
Moderate (0.5)	27.40	27.51	27.58	27.45	2.19	2.21	2.21	2.21
High (0.8)	27.64	27.69	27.63	27.54	2.22	2.24	2.23	2.23

* significantly better than the results without waiting, based on a two-tailed Wilcoxon signed rank test with a Bonferroni correction (for 6 pairwise comparisons), resulting in a significance level of $0.05/6 \approx 0.0083$

Table 5.2: The total number of routes in all updates except at the end of the planning horizon (n_{all}) and the number of times vehicles are instructed to wait (n_{wait}) in all 800 runs (40 instances $\times$ 20 runs) for the waiting strategy with each waiting threshold $\omega = 0.0$, 0.1, 0.2 and each degree of dynamism; $n_{\text{wait}}(\%)$ is the ratio of n_{wait} to n_{all}

Degree of dynamism	Waiting threshold								
	$\omega = 0.0$			$\omega = 0.1$			$\omega = 0.2$		
	n_{all}	n_{wait}	$n_{\text{wait}}(\%)$	n_{all}	n_{wait}	$n_{\text{wait}}(\%)$	n_{all}	n_{wait}	$n_{\text{wait}}(\%)$
Low (0.2)	130169	8865	6.8%	127832	3976	3.1%	126899	2061	1.6%
Moderate (0.5)	109526	4723	4.3%	108256	2244	2.1%	107660	1143	1.1%
High (0.8)	89766	1651	1.8%	89547	1019	1.1%	89392	653	0.7%

waiting instructions. The column n_{wait} shows the total number of times vehicles are instructed to wait in all 800 runs. The column $n_{\text{wait}}(\%)$ shows the ratio of n_{wait} to n_{all}, which indicates the likelihood that a vehicle is instructed to wait for a given waiting threshold and a given degree of dynamism. It can be see that in all cases, a small minority of active vehicles are given instructions to wait. In particular, when the waiting threshold $\omega = 0.2$ is used on the instances with the degree of dynamism $= 0.8$, only about 0.7% of all active vehicles in all updates are instructed to wait. This confirms that the waiting strategy is quite rarely used. This is because whether or not the waiting strategy is utilised depends largely on the existence of vehicles that will finish the last task before an upcoming update. As the degree of dynamism increases, fewer routes are constructed at the beginning of the planning horizon and more routes constructed in the middle of the planning horizon. In other words, routes tend to start, and thus end, at a later time. This generally decreases the chance that vehicles will finish the last task before a given update, thereby reducing the probability of vehicles being instructed to wait.

5.4 Instructing Vehicles to Wait Away from Other Vehicles

This section proposes another waiting strategy: to instruct vehicles to wait “far away” from the locations at which the other vehicles will be at the time of the next update. The motivation for this is that having many vehicles close to each other runs the risk of a new task appearing far away from all vehicles. In contrast, sending different vehicles to different areas would generally reduce the distance between a new task and the nearest vehicle. Furthermore, notice that in our dynamic CARP solver, the tabu search algorithm in each update is implemented using only information that is known in the current update, without anticipating changes that could occur at a later time. This waiting strategy gives a route planner more control over the vehicles’ starting vertices at the subsequent update instead of letting their starting vertices depend solely on the solution returned by the tabu search algorithm.

A waiting location according to this waiting strategy is determined by the following heuristic method. First, according to the solution given by the tabu search algorithm, if there will be more than one idle vehicle at the upcoming update, randomly order the idle vehicles. A waiting location will then be determined in such an order. Among vertices that can be reached by the upcoming update, the waiting location of each idle vehicle is a vertex v_{wait} that maximises the sum

$$\sum_{k \in W} D(v_{\text{wait}}, v_k), \tag{5.1}$$

where $D(u, v)$ denotes the shortest distance between two vertices u, v, and W is the set of non-idle vehicles plus idle vehicles for which waiting locations have been determined. For a non-idle vehicle, v_k is the vertex which the vehicle k will be at or will be heading towards (if it is in the middle of an edge) at the time of the next update. For an idle vehicle, v_k is its waiting location. To put it another away, Expression (5.1) places each idle vehicle in such a way that all the non-idle vehicles and the idle vehicles whose waiting locations have been determined are collectively far away from it. If there is more than one vertex with equal sums according to Expression (5.1), then one of them is randomly chosen to be the waiting location v_{wait}. A pseudocode of this novel process of determining a waiting location is shown in Algorithm 9. This waiting strategy will be implemented in place of the previous waiting strategy (waiting at the end of last task) within the dynamic CARP solver, i.e. Lines 4 and 15 of Algorithm 6.

It is possible that the above calculation results in v_{wait} being the depot for some idle vehicle. If this is the case, no waiting instruction is given to that vehicle, i.e. it is allowed to head towards the depot, where its route will end. This vehicle is then omitted from Expression (5.1) when determining waiting locations for the remaining idle vehicles.

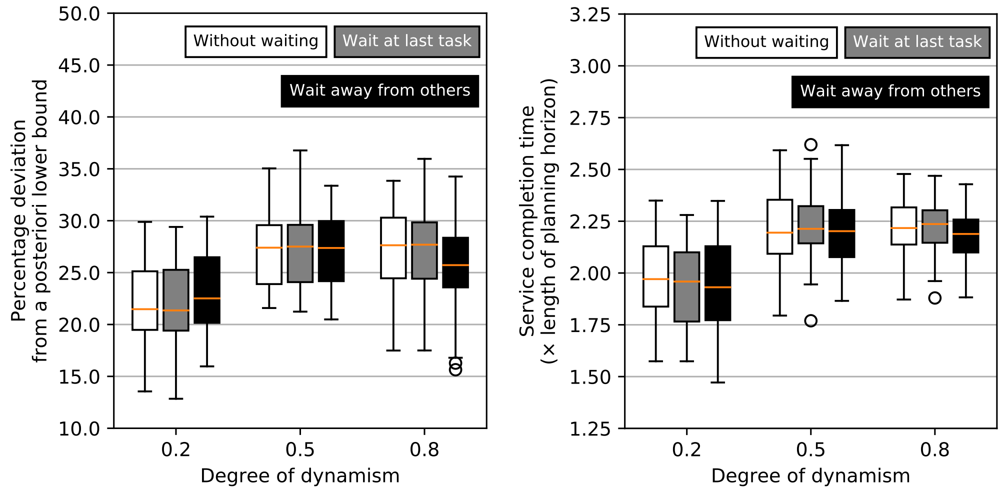
Algorithm 9 Instructing vehicles to wait away from other vehicles

```
1: given a set of routes, the time of the next update
2: check each route to see whether the corresponding vehicle will finish its last task
   before the upcoming update; if it does, call it "idle"; otherwise, call it "non-idle"
3: let $W = \{\text{non-idle vehicles}\}$   ▷ $W$ records vehicles whose locations at the next
   update have been determined
4: for each idle vehicle $\mathcal{V}$ (in a random order) do
5:     among vertices that the vehicle $\mathcal{V}$ can reach by the next update after servicing
       its last task, find a vertex $v_{\text{wait}}$ that maximises $f(v) = \sum_{k \in W} D(v, v_k)$ (if
       there is more than one such vertex, choose one of them randomly)
6:     if $v_{\text{wait}}$ is the depot then
7:         let the vehicle $\mathcal{V}$ head towards the depot (and end its route there)
8:     else
9:         instruct the vehicle $\mathcal{V}$ to head towards $v_{\text{wait}}$ after servicing its last task
           and wait there until the next update
10:        add the vehicle $\mathcal{V}$ to the set $W$
```

Provided that the shortest distances between each pair of vertices have been computed, Expression (5.1) can be computed in $O(K)$ time for each possible choice of v_{wait}, where K is the number of all vehicles. As the number of possible choices for v_{wait} is $O(|V|)$, the waiting location for each idle vehicle can be determined in $O(K|V|)$ time, hence $O(K^2|V|)$ time for finding the waiting locations for all idle vehicles in each update. This ensures that the computation time needed to find waiting locations according to this waiting strategy does not grow exponentially.

As was the case for the previous waiting strategy, this waiting strategy is tested with the dynamic CARP solver such that the number of updates is 20, the method of integrating new tasks is the Random Insertion method, and the maximum iteration limit for tabu search in each update is $50n_t$, where n_t is the number of tasks in each update. Figure 5.9 shows percentage deviations from a posteriori lower bounds and service completion times given by different waiting strategies, namely no waiting, waiting at the end of the last task (introduced in Section 5.2), and waiting away from other vehicles (introduced in this section). A two-tailed Wilcoxon signed rank test is also conducted to compare all 3 ways of waiting in pairs, and the significance results are shown in Table 5.3.

The computational results suggest that instructing vehicles to wait away from other



(a) (b)

Figure 5.9: Distributions of percentage deviations from a posteriori lower bounds with respect to total distances (a) and distributions of service completion times (b) over 40 dynamic CARP instances given by each waiting strategy

vehicles does not significantly improve the solution on the instances with low and moderate degrees of dynamism. In fact, regarding total distance, it is significantly worse than the other waiting strategies. However, with a high degree of dynamism, waiting away from other vehicles gives significantly better results than the other two waiting strategies with respect to both measures of solution quality. This illustrates the benefit of encouraging vehicles to be at different areas especially on the instances with a high degree of dynamism, i.e. in the case where a majority of tasks are known after vehicles leave the depot at the beginning of the planning horizon.

The likelihood of vehicles being instructed to wait according to each waiting strategy is shown in Table 5.4. The meanings of n_{all}, n_{wait}, and $n_{\text{wait}}(\%)$ are the same as those in Table 5.2. It can be seen that the "waiting away from other vehicles" strategy is also rarely used. The fact that instructing vehicles to wait away from other vehicles could give lower total distance and earlier service completion time than instructing vehicles to wait at the end of last task suggests that waiting locations could have an effect on the solution quality despite not being implemented very often.

5.5 Employing Extra Routes

One might expect that the greater number of tasks would require greater total distance. However, in computational results from Chapter 4, it was found that the

Table 5.3: Medians of percentage deviations from a posteriori lower bounds with respect to total distances and medians of service completions times (as multiples of the length of the planning horizon) over 40 dynamic CARP instances given by each waiting strategy

Degree of dynamism	Percentage deviation from a posteriori lower bound (%)			Service completion time ($\times$ planning horizon length)		
	Without waiting	Waiting at the end of last task	Waiting away from other vehicles	Without waiting	Waiting at the end of last task	Waiting away from other vehicles
Low (0.2)	21.47^{c}	21.36^{c}	22.52	1.97	1.96^{a}	1.93
Moderate (0.5)	27.40	27.51	27.37	2.19	2.21	2.20^{b}
High (0.8)	27.64	27.69	25.72^{ab}	2.22	2.24	2.19^{ab}

^{a} significantly better than without waiting

^{b} significantly better than waiting at the end of last task

^{c} significantly better than waiting away from other vehicles

based on a two-tailed Wilcoxon signed-rank test with Bonferroni correction (for 3 pairwise comparisons), resulting in a significance level of $0.05/3 \approx 0.017$

Table 5.4: The total number of routes in all updates except at the end of the planning horizon (n_{all}) and the number of times vehicles are instructed to wait (n_{wait}) in all 800 runs (40 instances $\times$ 20 runs) for each waiting strategy and each degree of dynamism; $n_{\text{wait}}(\%)$ is the ratio of n_{wait} to n_{all}

Degree of dynamism	Waiting at the end of last task			Waiting away from other vehicles		
	n_{all}	n_{wait}	$n_{\text{wait}}(\%)$	n_{all}	n_{wait}	$n_{\text{wait}}(\%)$
Low (0.2)	130169	8865	6.8%	128649	8171	6.4%
Moderate (0.5)	109526	4723	4.3%	108754	4142	3.8%
High (0.8)	89766	1651	1.8%	89589	1566	1.7%

solution cost (total distance of planned routes) decreased in some updates despite the appearance of new tasks. For example, Figure 5.10 shows the solution cost and the number of routes at each update on the C01-d80-1 instance[3] from an individual run of the dynamic CARP solver in Chapter 4 (with the 20-update schedule, the Random Insertion method, and the maximum iteration limit of $50n_t$, where n_t is the number of tasks at each update). In this sample run, the solution cost decreased when the solution was updated at time $0.7T$ and $0.85T$, where T is the length of the planning horizon. At these two updates, the number of routes increased by 1; in other words, a new route was created. This suggests that there might be a connection between the decrease in solution cost and the construction of a new route (at least for the dynamic CARP solver that was implemented here).

It is worth reminding ourselves at this stage that in this dynamic CARP solver, routes are constructed on an as-needed basis, that is, a new route is constructed only if the method of integrating new tasks could not assign all tasks in the current update to the existing routes. After the method of integrating new tasks, the number of routes remains unchanged throughout the course of the tabu search algorithm because the neighbourhood moves in the algorithm (see Section 3.2) operate in such a way that they do not create a new route. Consequently, if many routes are nearly full (i.e. the total demand almost reaches the capacity), there would be a relatively small number of feasible neighbourhood moves, especially Single Insertion and Double Insertion since nearly full routes would be unlikely to feasibly receive additional tasks. This restricts the way in which the tabu search algorithm could amend the solution. Introducing an extra (empty) route to the existing solution would allow more possible ways to move tasks between routes, which could increase the chance of tabu search improving a solution.

For a more comprehensive result, Figure 5.11 displays the number of updates in which the solution cost (i.e. total distance of planned routes) decreased and the proportion of those updates in which an extra route was created from the computational results of 800 runs (40 instances $\times$ 20 runs) of the dynamic CARP solver in Chapter 4 (with the Random Insertion method and the maximum iteration limit $50n_t$, where n_t is the number of tasks at each update). It can be seen that among the updates with a decrease in solution costs, a significant proportion of updates involved the construction of new routes, further emphasising the connection between the decrease in solution cost and the construction of a new route.

[3]The C01-d80-1 instance is generated based on a static CARP instance named C01 from the BMCV dataset with the degree of dynamism = 0.8 (i.e. 80%); see Section 4.3.

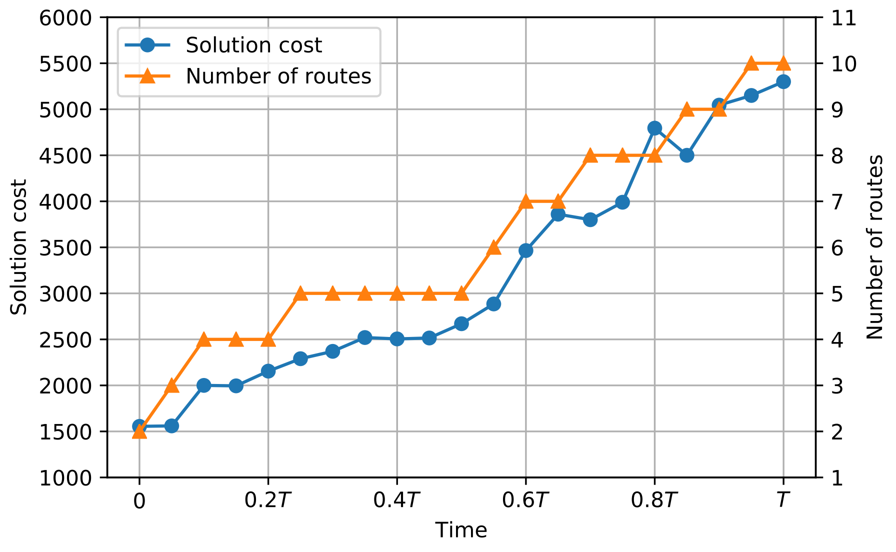


Figure 5.10: Solution cost at each update in an individual run on the C01-d80-1 instance given by the dynamic CARP solver from Chapter 4 with 20 updates and the Random Insertion method and the maximum iteration limit $50n_t$, where n_t is the number of tasks at each update; T denotes the length of the planning horizon

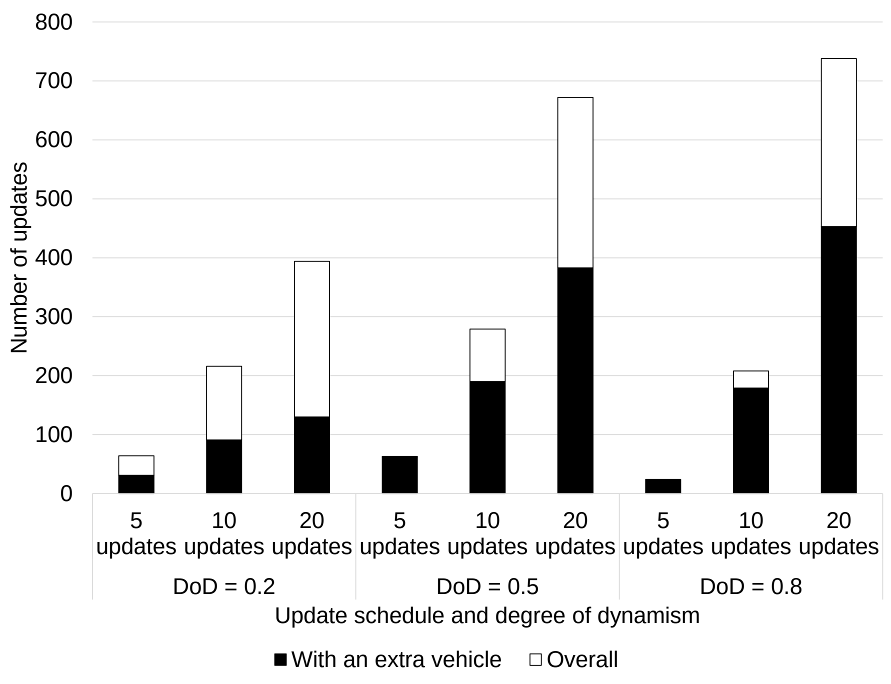


Figure 5.11: The number of updates in which the solution cost decreased given by the dynamic CARP solver from Chapter 4 with the Random Insertion method and the maximum iteration limit $50n_t$, where n_t is the number of tasks at each update; the black bars show the proportion of those updates in which an extra route was created.

This motivates the idea of introducing an extra route to the solution in each update. More precisely, in each update, several empty routes (the number of which is to be specified) are added to the solution before new tasks are added to it by the Random Insertion method. By an "empty" route, we mean a route that has only the starting and ending vertices (both of which are the depot) and has no tasks. Once the tabu search algorithm returns a solution, any empty routes that exist in the solution are removed to avoid accumulating empty routes throughout the planning horizon. The dynamic CARP solver with the use of extra routes (and a waiting strategy) is given in Algorithm 10.

Algorithm 10 Configuration of the dynamic CARP solver with waiting and the use of extra routes

```
1: given max_time (i.e. the length of the planning horizon), a rule for deciding
   when to perform solution updates (see Section 4.2.1), the number of extra routes,
   a waiting strategy (either no waiting, Algorithm 7, or Algorithm 9)
2: construct an initial solution $S$ by the Path Scanning algorithm
3: add a given number of empty route to $S$
4: apply the tabu search algorithm (from Chapter 3) to $S$
5: remove empty routes (if any) from $S$
6: implement the chosen waiting strategy
7: set time = 0
8: while time < max_time do
9:     increase time by 1 unit
10:    if time is an update time then
11:        let $\mathcal{T}$ be the set of new tasks that appear since last update
12:        if there exist new tasks then
13:            identify the fixed part and the starting vertex of each route for the
               current update (see Section 4.2.2)
14:            add a given number of empty route to $S$
15:            add the new tasks in $\mathcal{T}$ to $S$ by the Random Insertion method (see
               Algorithm 5)
16:            apply the tabu search algorithm (from Chapter 3) to $S$
17:            remove empty routes (if any) from $S$
18:        if this update is not the last one then
19:            implement the chosen waiting strategy
```

5.5.1 Computational Results

The idea of an extra route to a solution in each update is tested with 3 waiting strategies we saw previously in this chapter, namely no waiting, waiting at the end of last task, and waiting away from other vehicles, and the computational results are displayed in Tables 5.5 to 5.7, respectively. For ease of reference, the results without the use of extra routes (previously seen in Table 5.3) are also displayed in the same tables under the columns “0 extra routes”.

For the dynamic CARP solver with the “without waiting” and the “waiting at the end of last task” strategies (Tables 5.5 and 5.6), computational results show that the use of an extra route significantly reduces the total distance across different degrees of dynamism. The use of 2 extra routes also leads to significant improvement compared with no extra routes, although there is no significant difference between the results with 1 and 2 extra routes. This could be because the total demand of new tasks in each update is generally small in relation to the total capacity of two vehicles, and so the second extra route is unlikely to be used (i.e. it has no tasks) in the solution returned by the tabu search algorithm.

Regarding service completion time, Tables 5.5 and 5.6 also shows that the use of either 1 or 2 extra routes significantly reduces the service completion time in the case of moderate and high degrees of dynamism. This could be because there are more tasks appearing after the planning horizon begins on an instance with a higher degree of dynamism, and hence greater need to construct new routes. Adding an extra route to a solution in each update encourages new routes to be constructed at an earlier time, which in turn encourages routes to end at an earlier time.

For the dynamic CARP solver with the “waiting away from other vehicles” strategy (Table 5.7), the computational results show no significant improvement due to the use of an extra route in most cases. This suggests that the effect of an extra route on solution quality can vary with a waiting strategy, and in particular, a waiting location.

Overall, the use of an extra route can significantly improve the performance of the dynamic CARP solver when implemented without waiting or with the “waiting at the end of the last task” strategy. In fact, implementing the use of an extra route with the “waiting at the end of the last task” strategy appears to give the most promising results among all numbers of extra routes and all waiting strategies considered here. However, the second extra route does not appear to yield further significant improvement, although this is expected to depend on characteristics of problem instances, especially demands of new tasks in relation to the vehicle capacity; if their demands are sufficiently large, the benefit of the second extra route (or more extra routes in general) might be more apparent.

Table 5.5: Medians of percentage deviations from a posteriori lower bounds with respect to total distances and medians of service completion times over 40 dynamic CARP instances given by the dynamic CARP solver without waiting

Degree of dynamism	Percentage deviation from a posteriori lower bound (%)			Service completion time (× planning horizon length)		
	0 extra routes	1 extra route	2 extra routes	0 extra routes	1 extra route	2 extra routes
Low (0.2)	21.47	20.11*	20.14*	1.97	1.93	1.99
Moderate (0.5)	27.40	26.54*	27.02*	2.19	2.18*	2.19*
High (0.8)	27.64	25.29*	25.33*	2.22	2.17*	2.18*

* significantly better than not using extra routes, based on a two-tailed Wilcoxon signed rank test with a Bonferroni correction (for 3 pairwise comparisons), resulting in a significance level of $0.01/3 \approx 0.0033$

Table 5.6: Medians of percentage deviations from a posteriori lower bounds with respect to total distances and medians of service completion times over 40 dynamic CARP instances given by the dynamic CARP solver with waiting at the end of last task

Degree of dynamism	Percentage deviation from a posteriori lower bound (%)			Service completion time (× planning horizon length)		
	0 extra routes	1 extra route	2 extra routes	0 extra routes	1 extra route	2 extra routes
Low (0.2)	21.36	19.88*	19.71*	1.96	1.93	1.91
Moderate (0.5)	27.51	26.01*	26.39*	2.21	2.17*	2.19*
High (0.8)	27.69	25.42*	25.43*	2.24	2.18*	2.17*

* significantly better than not using extra routes, based on a two-tailed Wilcoxon signed rank test with a Bonferroni correction (for 3 pairwise comparisons), resulting in a significance level of $0.01/3 \approx 0.0033$

Table 5.7: Medians of percentage deviations from a posteriori lower bounds with respect to total distances and medians of service completion times over 40 dynamic CARP instances given by the dynamic CARP solver with waiting away from other vehicles

Degree of dynamism	Percentage deviation from a posteriori lower bound (%)		Service completion time (× planning horizon length)	
	0 extra routes	1 extra route	0 extra routes	1 extra route
Low (0.2)	22.52	22.29	1.93	1.91
Moderate (0.5)	27.37	27.61	2.20	2.15*
High (0.8)	25.72	25.93	2.19	2.17

* significantly better than not using extra routes, based on a two-tailed Wilcoxon signed rank test with a significance level of 0.01

5.6 Conclusions

In this chapter, we investigated ways of exploiting the notion of time in order to improve the effectiveness of route planning for the dynamic CARP. Prior to this chapter, it was implicitly assumed that vehicles travel without stopping until reaching the depot at the end of their routes, as they would in the static CARP. As a result, vehicles would return to the depot despite having some remaining capacity after the completion of services on their last tasks. Consequently, some remaining capacity would be wasted if vehicles return to the depot before the end of the planning horizon. This motivated the idea of instructing vehicles to wait at certain locations.

The first waiting strategy that we investigated was to let vehicles wait at the end of their last tasks until further instructions from an upcoming update. It was found that this waiting strategy indeed reduces the need to construct additional routes as new tasks appear. It can also prevent unnecessary routes that service zero demands, which can occur without waiting when a solution update moves all tasks from a certain route to the other routes before the vehicle reaches its first task. Despite its ability to prevent a waste of vehicle capacity, however, computational results suggested that this waiting strategy had only a marginal effect on the solution quality with respect to both the total distance and the service completion time.

The waiting strategy was then refined by the waiting threshold, which was a means to explicitly determine whether a vehicle has “too little” remaining capacity, which would be unlikely to receive additional tasks and thus should head towards the depot instead of waiting. Nevertheless, using the waiting threshold to restrict waiting instructions to vehicles with certain remaining capacity rarely made a noticeable difference to the solution quality. One possible reason for the waiting strategy and the waiting threshold having a small effect on the solution quality is that the waiting instructions are rarely given to vehicles, which in turn follows from the fact that there are not many vehicles whose last tasks are scheduled to be completed before an upcoming update, at least on the dynamic CARP instances considered here.

An alternative waiting strategy was also considered, namely instructing vehicles to wait away from other vehicles. This is to reduce the risk of a new task appearing far away from all vehicles when they are near each other. It was expected that placing vehicles far away from each other would allow vehicles to cover a wider area and thus could reach a new task faster, regardless of where the task appeared. Computational results suggested that this waiting strategy is particularly beneficial on instances with a relatively high degree of dynamism.

The idea of adding an extra route to the solution before the tabu search algorithm in each update was also investigated. This was expected to give the tabu search algorithm (or any local search algorithm in general) more flexibility in making changes to the solution, especially when the amounts of demands in many routes are close to the capacity, limiting the number of feasible neighbourhood moves. Empirical results showed that the use of one extra route significantly reduced both the total distance and the service completion time when implemented without waiting or with waiting at the end of the last task. However, the second extra route did not seem to give much further improvement, although the effect of the second extra route is expected to be more apparent on other instances in which the demands of new tasks are sufficiently large.

Furthermore, the dynamic CARP solver with the "waiting at the end of last task" and the use of 1 extra route gives the most promising results among all variants tested here, illustrating the benefit of combining an appropriate waiting location with an appropriate number of extra routes. It is also interesting to note that all waiting strategies introduced in this chapter do not significantly increase the service completion time.

Chapter 6

Conclusion

6.1 The Problem Investigated

This thesis concerns the capacitated arc routing problem (CARP), in which the goal is to find a minimum-cost set of routes such that (i) each route starts and ends at the depot, (ii) each task is serviced in one of the routes, and (iii) the total demand in each route does not exceed the capacity. The CARP can be used as a model of various real-life scenarios such as rubbish collection, snow ploughing, street sweeping, and other situations where an emphasis is placed on providing a certain service along streets (as opposed to individual locations).

Up until recently, the study of the CARP is concentrated on its "static" version, that is, all information of the problem is assumed to be available at the time of route planning, or equivalently, it is assumed that the problem does not change after vehicles start their journeys. Nevertheless, with the availability of global positioning systems and today's communication technology, a route planner has the capability to track vehicles and to be informed about any changes in the problem that may occur while the vehicles are travelling and performing their services. This opens up an opportunity for a route planner to amend vehicles' routes if deemed necessary or appropriate for the current state of the problem and, more importantly, to inform drivers of any changes in their routes so that the amended routes can indeed be followed. This motivates the study of a dynamic CARP.

A variety of possible changes means there are many types of dynamic CARPs. This thesis focusses on the dynamic CARP with the appearance of new tasks. In particular, all tasks that appear within a certain time interval, also referred to as the planning horizon, must be serviced. However, in the dynamic CARP that we consider here, it is not required that a new task is integrated into a solution as soon as it appears. In other words, a route planner is allowed to update the solution at any time and any number of times as long as all tasks are included in the solution at the end of the planning horizon.

Although the dynamic CARP can be viewed as a sequence of static CARPs, it has been shown in the literature that the static CARP is NP-hard. Literature review suggests that to find a good solution for the CARP in a reasonably short time, heuristic algorithms are a more promising choice than exact algorithms. For this reason, we opt to investigate ways of finding a solution for a dynamic CARP based on heuristic algorithms. In particular, a metaheuristic methodology called tabu search is considered as it usually requires a small number of parameters. This allows us to focus more on factors beyond tackling the static CARP in each update, including the frequency of solution updates, the method of integrating new tasks into a solution, and waiting strategies.

6.2 Summary of Findings

The findings and contributions made throughout this thesis are summarised below in connection with the research aims stated in Section 1.4.

1st Research Aim: To compare several variants of a tabu search algorithm for the static CARP and identify variants that can provide good solutions within limited time.

In existing literature, the performance of heuristic algorithms for the static CARP is commonly reported based on a given stopping criterion. In Chapter 3, we study the performance of variants of tabu search by analysing solution costs over a range of iteration numbers. This helps us better understand what variant of tabu search can improve solution quality at a relatively fast rate. This information will be useful for designing a heuristic algorithm for dynamic CARP, where it is preferable to find good solutions within limited time.

Variants of tabu search for the static CARP that we investigate in Chapter 3 differ in ways of defining tabu moves. In our tabu search algorithm, the process of determining the tabu status of neighbourhood move relies on recording and checking a specific type of solution attribute. Four types of solution attributes are compared: task-in-a-route, 2-task, 2-task-in-a-route, and 3-task attributes.

Computational results (shown in Table 3.2) suggest that the definition of tabu moves has a significant effect on the rate at which tabu seach improves the static CARP solution. The variants of tabu search that generally improves the solution at a faster rate than the others are those with the task-in-a-route and the 2-task attributes. Furthermore, the costs of solutions visited over a range of iterations by tabu search with the 3-task attribute tend to fluctuate noticeably less than the other types of attributes (examples are shown in Figure 3.4). This suggests that tabu search for the CARP can perform relatively well when the definition of tabu moves is relatively restrictive. In other words, there are generally a larger proportion of moves that are tabu, which consequently encourages the tabu search algorithm to explore a wider variety of solutions. This is because there are generally many CARP solutions that have the same total distance, so the ability to move away from a set of solutions with a certain total distance is important for achieving a high-quality solution over the course of execution of the tabu search algorithm.

In addition, a deadheading cycle remover is proposed with the aim of further improving a solution found by tabu search (or other heuristic algorithms for the CARP in general) without too much additional computational effort. Given a route, the deadheading cycle remover attempts to detect and remove deadheading cycles without disconnecting the route. In Section 3.5.1, the deadheading cycle remover is tested with the tabu search algorithm based on the task-in-a-route attribute. Empirical results (Figure 3.8) show that the tabu search algorithm is not always able to eliminate deadheading cycles on its own, and so the solution returned by the tabu search algorithm can be further improved by applying the deadheading cycle remover to the solution as a post-optimisation step. Moreover, the performance of the tabu search algorithm can be further improved by implementing the deadheading cycle remover in each iteration instead of only after the final iteration (as shown in Figure 3.9 and Table 3.4).

Recall that the tabu search algorithm with the task-in-a-route attribute involves two separate tabu lists, one list for 2-Opt moves and another list for the other neighbourhood moves. Although the computational results suggest that this variant of the tabu search algorithm is among those that improve the solution at the fastest rate, it is later discovered that using multiple tabu lists could fail to prevent the tabu search algorithm from returning to the solutions in previous iterations. For this reason, a dynamic CARP solver in Chapters 4 and 5 is based on the tabu search algorithm with another attribute that has similar performance, namely the 2-task attribute.

2nd Research Aim: To investigate the effect of different frequencies of updating the solution, and the way of integrating new tasks into an existing solution in each update on the quality of the final solution (i.e. the solution at the end of the planning horizon).

Existing works on dynamic CARP with new tasks or dynamic vehicle routing with new requests (i.e. demands on vertices) consider solving the problem with given update times (Chen and Xu, 2006; Liu et al., 2014a,b). There also exists other work on dynamic CARP that consider several choices of update frequencies (Montemanni et al., 2005), although it is not explicitly investigated how or whether a promising update frequency would vary with the degree of dynamism. This is studied in Chapter 4, where we test several frequencies of solution updates on dynamic CARP instances with a range of degrees of dynamism.

The first experiment in Chapter 4 reveals that running the tabu search algorithm for a greater number of iterations in each update does not always lead to significant improvement of dynamic CARP solutions (Section 4.4.1). This is possible because running the tabu search algorithm for different numbers of iterations generally gives different solutions, and different solutions in one update lead to different CARPs in a subsequent update. Therefore, in order to improve the solution at the end of the planning horizon, it does not suffice to rely solely on executing the tabu search algorithm for more iterations in each update. This highlights the need to improve the dynamic CARP solver by other means; here we consider adjusting the frequency of solution updates and the method of integrating new tasks.

Computational results in Chapter 4 show that changing the update frequency could significantly affect the solution quality with respect to both the final total distance, i.e. the total distance at which time all vehicles return to the depot after all tasks are serviced, and the service completion time, i.e. the time at which the last vehicle returns to the depot. However, its effect depends on the method of integrating new tasks. Recall that two such methods are considered: (i) “Reconstruction” – reconstructing the solution from scratch, and (ii) “Random Insertion” – retaining the solution from the previous update and inserting new tasks into the solution in a random order.

For the dynamic CARP solver that solves the problem in each update from scratch (i.e. using the Reconstruction method), the performance of different update schedule varies with the degree of dynamism (Section 4.4.2). When the degree of dynamism is relatively low, a more frequent update schedule tends to give solutions with lower total distance. In the case of relatively high degree of dynamism, the opposite is the case. However, regarding service completion time, a more frequent update schedule generally results in earlier service completion time, especially when the degree of dynamism is relatively low.

For the dynamic CARP solver with the Random Insertion method, a more frequent update schedule tends to give better solutions with respect to both total distance and service completion time, and this is the case across different degrees of dynamism (Section 4.5).

Overall, among all variants of the dynamic CARP solver considered here, empirical results show that the one that performs the best in general involves a 20-update schedule (the most frequent among those considered) and integrates new tasks into the solution by the Random Insertion method.

3rd Research Aim: To investigate the trade-off between the total distance travelled and resultant service completion time given by waiting strategies.

In existing work on dynamic CARP (Liu et al., 2014b), a solution at each update is found based solely on information that is available at the current update. However, when new tasks (or other types of changes in general) are expected to occur, it may be beneficial to anticipate those changes and amend the solution at each update accordingly. In Chapter 5, we investigate several ways of doing so, including instructing vehicles to wait and stand by at certain locations, and adding extra routes to the solution even if the current set of routes have enough capacity to service all tasks that have appeared so far.

Chapter 5 proposes and analyses the idea of instructing vehicles to wait at the end of their last tasks until further instructions from an upcoming update. Computational results (Section 5.2) show that this waiting strategy has a small effect on the quality of dynamic CARP solutions with respect to both total distance and service completion time. Nevertheless, it can reduce the chance of the vehicle capacity being underused and also encourages a dynamic CARP solver to return solutions with a smaller number of routes.

Apart from waiting at the end of the last task, another waiting strategy is also investigated, namely instructing vehicles to wait away from other vehicles. This waiting strategy attempts to prevent many vehicles from staying close to each other as this could result in a new task appearing far away from all vehicles. By letting vehicles to be far away from each other, they could cover a wider area and generally reach a new task sooner, regardless of where a new task appears. Empirical results show that this waiting strategy leads to significant improvement when the degree of dynamism is relatively high (Section 5.4).

The idea of adding extra (empty) routes to the solution before the tabu search algorithm in each update is also investigated. Adding an extra route to the solution would allow more ways of moving tasks between routes and potentially help tabu search explore a wider range of solutions, thereby increasing the chance of finding a better solution. Computational results (Section 5.5.1) show that the use of an extra route indeed improves the solution quality. Nevertheless, the second extra route does not seem to give much further improvement. This could be because the second extra route is not often used in the solution returned by the tabu search algorithm on the instances considered.

It is also interesting to note that implementing the “waiting at the last task” strategy together with the use of 1 extra route generally gives the best solutions among different waiting strategies and different numbers of extra routes. This illustrates the benefit of combining an appropriate waiting strategy with an appropriate number of extra routes. However, such appropriate choices are expected to depend on instance features such as the amount of demands in relation to the vehicle capacity.

6.3 Further Work

There are a variety of ways in which the study in this thesis could be extended. This section provides several suggestions.

Deadheading cycle remover. In Section 3.5, where we investigated a deadheading cycle remover, the focus was on deadheading cycles that arose from edges traversed more than twice in a given route. This type of deadheading cycles is guaranteed to be removable without disconnecting the route, and thus no computation is needed to check whether or not such cycles are removable. Nevertheless, there are also other types of deadheading cycles, including those on edges that are traversed twice or simple cycles (i.e. cycles that has no repeated edges) that are composed of traversals on different edges. However, when these types of deadheading cycles are encountered, it is necessary to check whether they are removable. One way of checking this is to view the route as a graph, say G_r, and remove a cycle from G_r. Then starting from any vertex, determine a set of vertices in G_r that can be reached from the starting vertex, which can be achieved by breadth-first search, for example. If all vertices in G_r can be reached, this means that G_r remains connected and therefore the cycle can be removed. This would allow us to find more removable deadheading cycles and potentially better solutions, although additional computation for checking the removability would be needed. Investigating a balance between considering more deadheading cycles and saving computational time would provide more insights about how to develop an algorithm that is both quick and effective.

Update schedules. In Chapter 4, we opted to investigate regular update schedules, i.e. the time between consecutive updates is the same throughout the planning horizon. Our experiments involved only 3 regular update schedules, in which there are either 5, 10, or 20 updates. To extend this research, a wider range of numbers of updates may be studied. Alternatively, "irregular" update schedules could also be considered. That is, instead of fixed time intervals, the solution updates may be scheduled according to how new tasks appear, for example, updating the solution as soon as each task appears or after a certain number of tasks appear. In practice, there may also be information about the appearance of tasks throughout the planning horizon, for example, new tasks may be more likely to appear within a certain time interval than others. It is interesting to find a way to exploit such information to adjust the solution update schedule accordingly.

Algorithms for finding solutions in each update. In addition to tabu search, which was investigated in this thesis, other heuristic algorithms could also be used to improve the solution in each update. In the literature, there are a wide variety of heuristic algorithms proposed for the static CARP, although their performance is usually reported based on solutions they return after a certain stopping criterion is satisfied (for example, when a given maximum number of iterations is reached). Investigating the rate at which they improve the solution would provide more insights about which algorithm tends to improve a solution at a faster rate than others and what makes such an algorithm able to do so. This would help us develop a better algorithm for planning routes in the dynamic CARP, where the time taken to find a solution is not necessarily less important than the solution quality.

Appearance times of new tasks. In the dynamic CARP instances that are considered in our experiments, it is assumed that the times at which new tasks appear are uniformly distributed over the planning horizon. In practice, however, those appearance times may have different distributions. A wider range of dynamic CARP instances are therefore needed to be considered in order to cover a greater variety of the dynamic CARP.

Waiting locations. In Chapter 5, we investigate two ways of determining waiting locations: at the end of the last task or at a location far away from other vehicles. It was found that the first waiting strategy has a very small effect on the quality of dynamic CARP solution (except when it is implemented with the use of an extra route), and the second waiting strategy only yields significant improvement on instances with relatively high degrees of dynamism. Consequently, further research is needed to identify a more robust waiting strategy that can enhance the performance of a dynamic CARP solver over a wide range of degrees of dynamism.

Moreover, it is assumed in Chapter 5 that vehicle drivers can wait at any vertex. In reality, however, it might be impractical to wait at a certain vertex, for example, if that vertex represents an intersection of small streets, in which case parking could block the traffic. Also, vehicle drivers may prefer to wait at a certain location such as a rest stop at the side of a highway rather than at a remote area. Taking into account the practicality of each possible waiting location would allow us to find a balance between improving the solution quality and ensuring drivers' healthy working conditions.

Postponing tasks until the next planning horizon. In this thesis, we opt to consider the dynamic CARP on a single planning horizon. Nevertheless, some services such as street sweeping are usually performed regularly, in which case a series of planning horizons exist. For a service that is not urgent, some tasks may be better serviced in the next planning horizon, for example, when it appears very close to the end of the current planning horizon and all vehicles are nearly full and/or almost reach the depot. Considering multiple planning horizons would allow a route planner more options to decide how tasks should be serviced and could reduce the overall total distance (or some other objectives in general) than focussing on one planning horizon at a time.

In general, considering route planning in a larger scale of time and/or space would provide a route planner with greater flexibility to plan and amend routes, thus potentially achieving better routes in the long run. Nevertheless, with greater flexibility comes a vast increase in the number of possible solutions, not to mention the fact that a larger scale of time would allow a larger number of changes to occur over time, further adding the intractability of the problem. The quest for long-term efficient routes necessitates the development of algorithms for route planning in the face of changes that are often unpredictable in the dynamic CARP, as well as other dynamic routing problems.

Appendix A

Performance of Existing Metaheuristic Algorithms for the Static CARP

This chapter presents the performance (solution costs and computation times) of existing metaheuristic algorithms for the static CARP as reported in their original papers. These include:

- a tabu search algorithm ('CARPET') by Hertz et al. (2000),
- a guided local search local search ('GLS') by Beullens et al. (2003),
- a memetic algorithm ('MA') by Lacomme et al. (2004),
- a tabu search algorithm ('TSA') by Brandão and Eglese (2008),
- a memetic algorithm ('MAENS') by Tang et al. (2009),
- an ant colony optimisation algorithm ('Ant-CARP') by Santos et al. (2010).

For some of the above studies, the performance of their algorithms is reported in multiple ways:

- For the guided local search algorithm by Beullens et al. (2003), the results with 10^5 iterations and 5×10^5 iterations are displayed in the columns 'GLS1' and 'GLS2', respectively.

- In the original paper of Brandão and Eglese (2008), their main algorithm is referred to as 'TSA Version 1', whereas 'TSA Version 2' results from their attempt to obtain better solutions by running their main algorithm several times successively; these variants are denoted here by 'TSA1', and 'TSA2', respectively.
- In the work of Santos et al. (2010), their ant colony optimisation algorithm involves local search. They consider two versions of their algorithm, one involving 6 types of neighbourhood moves in the local search, and the other involving 12 types (including the former 6 types). These variants are denoted here by 'Ant-CARP-6' and 'Ant-CARP-12', respectively.

It should also be noted that Tang et al. (2009) report the results of their algorithm in terms of averages over 30 independent runs, while Santos et al. (2010) report median costs and average computation times (to achieve the median costs) calculated from results of executing their algorithm 15 times on each instance.

For ease of reference, Tables A.1 to A.6 also include computation times of the respective algorithms as reported in their original papers; these are displayed under the columns 'Time'. It should be noted, however, that the computation times from different papers are obtains from computers with different specifications (including different speeds of CPUs). Therefore, care needs to be taken when comparing these computation times to ensure a fair comparison of the algorithm speeds.

Details of benchmark instance sets for the static CARP can be found in Table 2.1. These instances, as well as their optimal costs or best known lower bounds can be found at the website `http://logistik.bwl.uni-mainz.de/benchmarks.php` (last accessed 21 March 2018).

Table A.1: Solution costs given by existing metaheuristic algorithms for the CARP on the VAL instance set

Instance	Optimal*	CARPET		GLS1		GLS2		MA		TSA1		TSA2		MAENS	Ant-CARP-6		Ant-CARP-12	
		Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time		Cost	Time	Cost	Time
1A	173	173	0.1	173	0.1	173	0.1	173	0.0	173	0.0	173	0.0	173.0	173	0.0	173	0.1
1B	173	173	50.2	173	0.3	173	0.3	173	8.0	173	0.2	173	0.9	173.0	173	2.2	173	0.6
1C	245	245	506.1	245	45.3	245	198.2	245	28.7	245	0.8	245	12.1	245.0	245	0.2	245	0.2
2A	227	227	0.9	227	0.0	227	0.0	227	0.1	227	0.0	227	0.0	227.0	227	0.0	227	0.0
2B	259	260	70.7	259	0.3	259	0.3	259	0.2	259	0.1	259	0.3	259.0	259	0.1	259	0.1
2C	457	494	171.6	457	44.9	457	180.0	457	21.8	457	1.5	457	7.8	457.2	457	0.2	457	0.2
3A	81	81	4.2	81	0.1	81	0.1	81	0.1	81	0.0	81	0.0	81.0	81	0.0	81	0.0
3B	87	87	15.1	87	0.1	87	0.1	87	0.0	87	0.0	87	0.0	87.0	87	0.2	87	0.1
3C	138	138	225.8	138	0.6	138	0.6	138	28.2	138	0.6	138	1.3	138.0	138	0.2	138	0.2
4A	400	400	153.5	400	0.1	400	0.1	400	0.7	400	0.1	400	0.4	400.0	400	0.0	400	0.9
4B	412	416	410.1	412	0.1	412	0.1	412	1.2	414	1.7	412	5.5	412.0	412	1.5	412	1.5
4C	428	453	379.7	428	0.3	428	0.3	428	19.1	444	1.7	428	38.0	431.1	428	14.7	428	25.0
4D	528	556	1265.9	530	63.0	530	275.9	541	103.3	538	10.3	530	110.0	532.9	530	50.1	530	74.2
5A	423	423	20.6	423	0.3	423	0.3	423	1.9	423	0.3	423	0.3	423.0	423	1.1	423	0.9
5B	446	448	224.3	446	0.4	446	0.4	446	1.0	446	0.1	446	0.1	446.0	446	1.4	446	0.9
5C	474	476	288.7	474	59.0	474	266.7	474	101.0	474	1.2	474	10.6	474.0	474	56.6	474	82.8
5D	575	607	1214.7	583	56.7	579	241.2	581	90.7	583	6.5	583	73.3	582.9	582	41.4	583	61.4
6A	223	223	21.1	223	0.1	223	0.1	223	0.2	223	0.1	223	1.6	223.0	223	0.1	223	0.2
6B	233	241	146.0	233	31.6	233	148.9	233	67.3	233	2.5	233	12.7	233.0	233	30.8	233	44.6
6C	317	329	461.7	317	39.1	317	168.7	317	52.2	323	3.1	317	22.9	317.0	317	22.1	317	32.2
7A	279	279	35.7	279	0.1	279	0.1	279	2.0	283	0.9	279	1.0	279.0	279	0.2	279	0.1
7B	283	283	0.1	283	0.2	283	0.2	283	0.4	283	0.1	283	0.5	283.0	283	0.4	283	0.4
7C	334	343	658.2	334	48.0	334	225.8	334	101.2	335	4.0	334	37.0	334.0	334	5.0	334	5.0
8A	386	386	20.8	386	0.1	386	0.1	386	0.7	386	0.6	386	0.3	386.0	386	0.8	386	0.4
8B	395	401	441.5	395	0.2	395	0.2	395	10.0	407	1.0	395	1.8	395.0	395	2.0	395	1.9
8C	521	533	798.9	523	53.3	521	235.1	527	71.5	545	1.9	529	55.7	525.9	527	39.2	527	55.6
9A	323	323	154.5	323	2.3	323	2.3	323	18.3	323	0.7	323	0.0	323.0	323	9.5	323	8.9
9B	326	329	324.6	326	0.6	326	0.6	326	29.4	326	1.3	326	0.5	326.0	326	9.0	326	7.6
9C	332	332	305.9	332	1.2	332	1.2	332	71.2	332	0.7	332	0.4	332.0	332	10.2	332	11.0
9D	388	409	1914.8	391	81.1	391	372.0	391	211.1	404	7.3	391	60.4	391.0	391	104.6	391	155.2
10A	428	428	29.9	428	0.3	428	0.3	428	25.5	430	3.5	428	3.2	428.0	428	34.4	428	22.2
10B	436	436	99.9	436	1.5	436	1.5	436	4.7	438	3.6	436	1.8	436.0	436	47.4	436	49.1
10C	446	451	506.6	446	1.6	446	1.6	446	17.3	447	4.6	446	7.5	446.0	446	34.7	446	41.0
10D	525	544	847.2	529	96.1	526	441.4	530	215.0	534	10.9	530	218.1	533.6	530	126.1	528	175.6

* Originally, service costs on these instances are given in addition to edge costs. Here, the optimal costs are adjusted so that the cost of servicing each task is equal to the cost of traversing the corresponding edge.

Table A.2: Solution costs given by existing metaheuristic algorithms for the CARP on the BMCV instance set ('C' instances)

Instance	Best known lower bound	GLS1		GLS2		TSA1		TSA2		MAENS	Ant-CARP-6		Ant-CARP-12	
		Cost	Time	Cost	Time	Cost	Time	Cost	Time		Cost	Time	Cost	Time
C1	4150	4150	72.9	4150	325.4	4190	12.7	4150	127.1	4197.0	4195	108.6	4195	143.5
C2	3135	3135	7.9	3135	7.9	3205	1.6	3135	26.6	3135.7	3135	39.3	3135	53.8
C3	2575	2575	40.5	2575	172.6	2585	3.8	2575	19.4	2577.8	2585	45.3	2575	61.4
C4	3510	3510	64.3	3510	283.7	3685	11.0	3510	65.4	3512.7	3510	97.0	3510	121.5
C5	5365	5370	52.8	5370	234.3	5525	6.8	5365	47.8	5417.3	5370	60.5	5435	85.0
C6	2535	2535	37.4	2535	160.4	2550	2.7	2535	9.3	2547.5	2535	46.7	2535	58.3
C7	4075	4075	38.2	4075	166.6	4075	5.4	4075	29.9	4075.0	4075	42.5	4075	54.1
C8	4090	4090	52.1	4090	229.8	4100	7.2	4090	44.4	4092.3	4090	45.8	4090	62.8
C9	5245	5270	99.5	5265	445.1	5320	24.6	5270	245.9	5292.8	5300	171.2	5300	226.9
C10	4700	4720	41.8	4720	181.1	4800	5.5	4700	30.6	4747.8	4760	46.0	4735	60.7
C11	4615	4640	92.8	4640	423.6	4765	22.8	4640	209.4	4678.7	4645	164.9	4645	215.5
C12	4240	4240	64.5	4240	285.2	4390	10.8	4240	46.2	4240.0	4240	79.0	4240	101.6
C13	2955	2955	40.5	2955	173.6	2960	4.4	2955	23.8	2967.0	2955	49.1	2955	58.4
C14	4030	4030	44.4	4030	198.9	4060	6.4	4030	54.6	4037.3	4030	43.2	4030	56.9
C15	4920	4970	116.0	4940	552.3	4990	30.8	4945	335.3	4976.5	4950	208.5	4960	315.2
C16	1475	1475	29.1	1475	121.9	1475	0.8	1475	5.1	1475.2	1475	16.7	1475	25.8
C17	3555	3565	32.0	3555	137.5	3575	3.2	3555	14.0	3563.3	3555	29.3	3555	38.3
C18	5580	5660	120.7	5645	565.6	5695	38.8	5650	520.8	5646.7	5625	307.1	5625	397.0
C19	3115	3115	46.4	3115	210.2	3145	7.0	3120	76.2	3145.7	3120	61.7	3120	77.7
C20	2120	2120	1.2	2120	1.2	2250	1.0	2120	2.6	2123.5	2120	41.7	2120	51.2
C21	3970	3970	72.7	3970	326.9	3970	3.7	3970	32.0	3970.2	3970	114.1	3970	140.2
C22	2245	2245	2.8	2245	2.8	2245	0.1	2245	0.7	2245.0	2245	29.9	2245	36.3
C23	4075	4085	83.7	4085	381.6	4170	18.9	4095	99.5	4119.3	4095	138.1	4105	191.1
C24	3400	3405	68.6	3400	311.4	3445	6.0	3400	78.9	3408.5	3400	121.2	3400	165.7
C25	2310	2310	0.3	2310	0.3	2340	1.9	2310	0.7	2312.0	2310	22.6	2310	31.4

Table A.3: Solution costs given by existing metaheuristic algorithms for the CARP on the BMCV instance set ('D' instances)

Instance	Best known lower bound	GLS1		GLS2		TSA1		TSA2		MAENS	Ant-CARP-6		Ant-CARP-12	
		Cost	Time	Cost	Time	Cost	Time	Cost	Time		Cost	Time	Cost	Time
D1	3215	3215	11.8	3215	11.8	3355	2.8	3230	62.7	3235.0	3235	145.7	3235	199.7
D2	2520	2520	1.2	2520	1.2	2520	1.6	2520	0.8	2520.0	2520	51.4	2520	72.6
D3	2065	2065	0.4	2065	0.4	2065	0.2	2065	0.2	2065.2	2065	46.2	2065	67.1
D4	2785	2785	0.5	2785	0.5	2800	4.4	2785	1.7	2786.0	2785	125.3	2785	171.5
D5	3935	3935	1.4	3935	1.4	3975	2.4	3935	2.8	3935.0	3935	79.6	3935	117.5
D6	2125	2125	0.2	2125	0.2	2165	0.7	2125	7.9	2133.0	2125	45.5	2125	65.6
D7	3115	3115	30.7	3115	138.9	3195	3.4	3115	22.8	3127.3	3165	49.4	3135	58.9
D8	3045	3055	43.3	3045	195.1	3045	4.5	3045	49.1	3064.2	3055	58.0	3045	77.5
D9	4120	4120	1.1	4120	1.1	4250	6.1	4120	30.1	4120.0	4120	196.8	4120	262.7
D10	3340	3340	31.1	3340	149.1	3340	1.3	3340	16.5	3340.0	3340	56.4	3340	78.1
D11	3745	3755	75.7	3755	368.3	3815	8.0	3785	112.0	3760.2	3775	184.7	3760	273.9
D12	3310	3310	0.4	3310	0.4	3365	2.5	3310	26.0	3310.0	3310	92.4	3310	122.3
D13	2535	2535	0.8	2535	0.8	2540	1.1	2540	13.9	2536.0	2535	60.7	2535	75.5
D14	3280	3280	38.8	3280	180.2	3300	1.0	3290	21.1	3281.0	3280	54.1	3280	76.3
D15	3990	4000	92.1	3990	368.8	4180	23.6	4030	102.3	3999.0	4000	301.7	4000	408.6
D16	1060	1060	0.0	1060	0.0	1065	0.1	1060	0.4	1060.0	1060	19.2	1060	28.0
D17	2620	2620	0.2	2620	0.2	2620	0.0	2620	0.0	2620.0	2620	34.7	2620	46.1
D18	4165	4165	2.5	4165	2.5	4310	13.9	4165	190.5	4169.2	4165	380.1	4165	519.9
D19	2400	2400	32.8	2400	154.8	2410	2.7	2410	20.1	2400.0	2400	78.5	2400	101.6
D20	1870	1870	0.1	1870	0.1	1875	2.5	1870	1.1	1870.2	1870	58.0	1870	73.4
D21	3005	3055	53.6	3050	251.8	3110	2.3	3070	60.9	3079.2	3055	126.7	3055	164.3
D22	1865	1865	0.3	1865	0.3	1865	0.0	1865	0.0	1865.0	1865	34.3	1865	43.4
D23	3130	3130	71.6	3130	335.3	3175	5.3	3130	111.1	3143.2	3140	157.2	3130	215.1
D24	2710	2710	50.2	2710	248.0	2710	10.3	2710	105.2	2723.5	2710	176.8	2710	249.5
D25	1815	1815	0.0	1815	0.0	1915	0.2	1815	0.6	1815.0	1815	25.2	1815	34.0

Table A.4: Solution costs given by existing metaheuristic algorithms for the CARP on the BMCV instance set ('E' instances)

Instance	Best known lower bound	GLS1		GLS2		TSA1		TSA2		MAENS	Ant-CARP-6		Ant-CARP-12	
		Cost	Time	Cost	Time	Cost	Time	Cost	Time		Cost	Time	Cost	Time
E1	4900	4920	74.9	4915	342.0	5135	13.4	4910	203.1	4942.8	4925	106.3	4920	140.7
E2	3990	3990	40.8	3990	188.6	4080	5.5	3990	53.2	3995.5	3990	52.2	3990	67.0
E3	2015	2015	0.8	2015	0.8	2015	0.1	2015	7.7	2017.0	2015	38.6	2015	46.9
E4	4155	4155	69.3	4155	328.3	4305	11.1	4160	132.2	4229.3	4220	99.7	4220	123.1
E5	4585	4670	48.0	4595	214.1	4660	8.5	4585	81.9	4653.7	4645	59.6	4645	77.9
E6	2055	2055	0.1	2055	0.1	2055	0.0	2055	0.7	2055.0	2055	32.2	2055	38.6
E7	4155	4155	37.7	4155	161.7	4155	0.9	4155	39.1	4155.0	4155	41.6	4155	52.5
E8	4710	4710	49.8	4710	221.6	4740	6.6	4715	83.7	4710.5	4710	59.0	4710	74.2
E9	5805	5865	95.7	5835	440.2	6025	24.3	5885	257.3	5912.7	5900	210.6	5880	237.7
E10	3605	3605	0.2	3605	0.2	3605	1.3	3605	3.7	3606.5	3605	36.7	3605	45.8
E11	4650	4670	91.7	4670	420.4	4800	22.4	4675	174.3	4752.0	4680	169.6	4680	233.5
E12	4180	4200	61.8	4195	264.7	4220	7.1	4215	89.7	4249.3	4245	78.3	4245	93.1
E13	3345	3345	42.0	3345	178.5	3435	1.0	3345	39.0	3355.3	3345	46.0	3345	54.1
E14	4115	4115	43.0	4115	190.2	4145	3.8	4115	45.4	4122.0	4135	49.1	4115	62.8
E15	4205	4230	109.5	4225	503.4	4260	29.2	4225	303.3	4232.8	4225	245.2	4225	324.4
E16	3775	3775	50.7	3775	199.3	3820	5.5	3775	70.7	3775.0	3775	56.0	3775	72.5
E17	2740	2740	6.3	2740	6.3	2775	1.6	2740	0.0	2744.3	2740	28.3	2740	36.3
E18	3835	3835	78.8	3835	363.1	3885	5.0	3835	123.2	3837.3	3835	149.1	3835	212.5
E19	3235	3235	46.4	3235	211.5	3275	7.9	3235	101.6	3237.0	3235	81.5	3235	103.4
E20	2825	2825	51.2	2825	232.2	2855	7.3	2825	78.8	2825.0	2825	88.5	2825	104.7
E21	3730	3740	64.7	3730	293.4	3815	3.8	3730	78.4	3780.5	3785	111.6	3785	136.0
E22	2470	2470	29.8	2470	129.0	2500	2.8	2470	30.7	2472.5	2470	34.4	2470	45.5
E23	3710	3730	86.9	3710	394.8	3810	17.8	3725	189.6	3749.0	3755	172.8	3715	209.6
E24	4020	4025	79.7	4020	362.1	4085	13.4	4020	112.5	4057.2	4020	148.0	4020	195.0
E25	1615	1615	0.0	1615	0.0	1615	0.1	1615	0.1	1615.0	1615	14.9	1615	27.7

Table A.5: Solution costs given by existing metaheuristic algorithms for the CARP on the BMCV instance set ('F' instances)

Instance	Best known lower bound	GLS1		GLS2		TSA1		TSA2		MAENS	Ant-CARP-6		Ant-CARP-12	
		Cost	Time	Cost	Time	Cost	Time	Cost	Time		Cost	Time	Cost	Time
F1	4040	4040	3.4	4040	3.4	4065	11.5	4060	88.5	4046.0	4050	120.4	4040	157.4
F2	3300	3300	2.3	3300	2.3	3320	2.6	3300	0.3	3300.0	3300	53.1	3300	76.4
F3	1665	1665	0.0	1665	0.0	1745	1.8	1665	1.3	1665.0	1665	45.8	1665	65.6
F4	3485	3485	53.6	3485	247.8	3515	2.7	3505	63.7	3508.5	3500	130.8	3500	182.7
F5	3605	3605	0.2	3605	0.2	3610	1.8	3605	17.1	3605.3	3605	77.8	3605	108.2
F6	1875	1875	0.0	1875	0.0	1925	0.4	1875	0.0	1875.0	1875	42.7	1875	55.4
F7	3335	3335	0.1	3335	0.1	3365	3.6	3335	1.7	3345.7	3335	45.1	3335	52.9
F8	3705	3705	40.1	3705	185.5	3715	1.1	3705	35.7	3705.0	3705	78.9	3705	93.4
F9	4730	4730	3.7	4730	3.7	5105	7.4	4755	145.9	4782.8	4810	308.7	4810	424.6
F10	2925	2925	0.1	2925	0.1	2925	0.1	2925	0.1	2925.0	2925	50.8	2925	69.4
F11	3835	3835	4.5	3835	4.5	3920	8.5	3835	125.8	3857.5	3865	198.6	3865	265.7
F12	3395	3395	50.3	3395	228.7	3485	4.9	3395	47.9	3424.5	3405	76.6	3460	136.8
F13	2855	2855	0.2	2855	0.2	2875	0.8	2855	0.2	2855.0	2855	60.9	2855	73.7
F14	3330	3330	18.8	3330	18.8	3390	2.5	3340	29.7	3370.5	3330	53.8	3330	76.4
F15	3560	3560	1.3	3560	1.3	3930	22.1	3605	145.4	3566.7	3560	298.0	3560	409.6
F16	2725	2725	0.1	2725	0.1	2895	1.9	2725	3.4	2725.0	2725	67.6	2725	83.4
F17	2055	2055	0.1	2055	0.1	2110	0.3	2080	4.7	2055.0	2055	23.9	2055	31.9
F18	3065	3075	56.8	3075	274.5	3170	13.9	3075	92.9	3086.2	3075	163.8	3075	243.5
F19	2515	2525	33.1	2525	158.0	2540	1.5	2540	34.9	2525.0	2525	88.1	2525	125.5
F20	2445	2445	1.3	2445	1.3	2445	0.3	2445	9.8	2449.8	2450	102.2	2445	141.7
F21	2930	2930	4.0	2930	4.0	2965	2.1	2930	45.5	2930.0	2930	138.6	2930	183.3
F22	2075	2075	0.4	2075	0.4	2075	0.1	2075	0.4	2075.0	2075	50.6	2075	65.7
F23	3005	3005	68.4	3005	319.6	3175	4.6	3010	80.2	3016.3	3010	177.6	3010	234.4
F24	3210	3210	26.6	3210	26.6	3275	10.3	3245	79.2	3236.3	3210	168.8	3210	243.3
F25	1390	1390	0.1	1390	0.1	1390	0.0	1390	0.1	1390.0	1390	16.7	1390	22.3

Table A.6: Solution costs given by existing metaheuristic algorithms for the CARP on the EGL instance set

Instance	Best known lower bound	MA		TSA1		TSA2		MAENS	Ant-CARP-6		Ant-CARP-12	
		Cost	Time	Cost	Time	Cost	Time		Cost	Time	Cost	Time
E1-A	3548	3548	74.3	3548	2.1	3548	22.1	3548.0	3548	0.5	3548	0.5
E1-B	4498	4498	69.5	4533	4.8	4533	28.0	4516.5	4539	47.3	4539	63.5
E1-C	5595	5595	71.2	5659	5.1	5595	24.1	5601.6	5613	46.4	5595	72.6
E2-A	5018	5018	152.6	5018	7.7	5018	63.4	5018.0	5018	113.7	5018	152.5
E2-B	6317	6340	153.4	6385	11.5	6343	66.7	6341.4	6344	98.0	6344	130.3
E2-C	8335	8415	129.6	8400	12.0	8347	78.7	8355.7	8339	102.7	8335	140.1
E3-A	5898	5898	242.0	6044	17.9	5902	77.3	5898.8	5898	53.0	5898	32.4
E3-B	7744	7822	255.4	7916	17.8	7816	113.4	7802.9	7789	159.0	7787	211.8
E3-C	10 244	10 433	206.4	10 309	23.2	10 309	134.3	10 321.9	10 305	140.7	10 292	193.9
E4-A	6408	6461	291.9	6476	14.0	6473	135.5	6475.2	6471	224.0	6464	313.5
E4-B	8935	9021	312.9	9134	26.9	9063	167.6	9023.0	9065	193.9	9047	275.1
E4-C	11 512	11 779	252.4	11 627	31.8	11 627	188.6	11 645.8	11 658	182.5	11 645	286.8
S1-A	5018	5018	208.6	5171	10.3	5072	66.6	5039.8	5018	98.4	5018	157.3
S1-B	6388	6435	208.8	6388	13.1	6388	80.8	6433.4	6435	106.8	6388	155.1
S1-C	8518	8518	165.6	8739	6.9	8535	79.2	8518.3	8518	97.7	8518	151.1
S2-A	9825	9995	874.4	10 190	70.2	10 038	395.1	9959.2	9985	484.9	9974	709.0
S2-B	13 017	13 174	760.5	13 284	78.2	13 178	448.3	13 231.6	13 266	578.5	13 283	772.6
S2-C	16 425	16 795	746.9	16 709	53.6	16 505	515.8	16 509.8	16 636	596.2	16 558	684.6
S3-A	10 165	10 296	1070.5	10 508	79.3	10 451	554.2	10 312.7	10 306	675.3	10 306	843.0
S3-B	13 648	14 053	1064.0	13 981	84.2	13 981	570.6	13 876.6	13 899	800.6	13 890	889.5
S3-C	17 188	17 297	874.3	17 346	99.1	17 346	596.4	17 305.8	17 341	597.7	17 304	768.6
S4-A	12 153	12 442	1537.6	12 546	129.8	12 462	696.8	12 419.2	12 457	1186.4	12 439	1548.2
S4-B	16 113	16 531	1430.3	16 695	141.4	16 490	954.6	16 441.2	16 502	1077.2	16 502	2067.2
S4-C	20 430	20 832	1495.0	20 981	144.9	20 733	934.5	20 767.2	20 796	1069.7	20 731	1453.4

Appendix B

Additional Computational Results

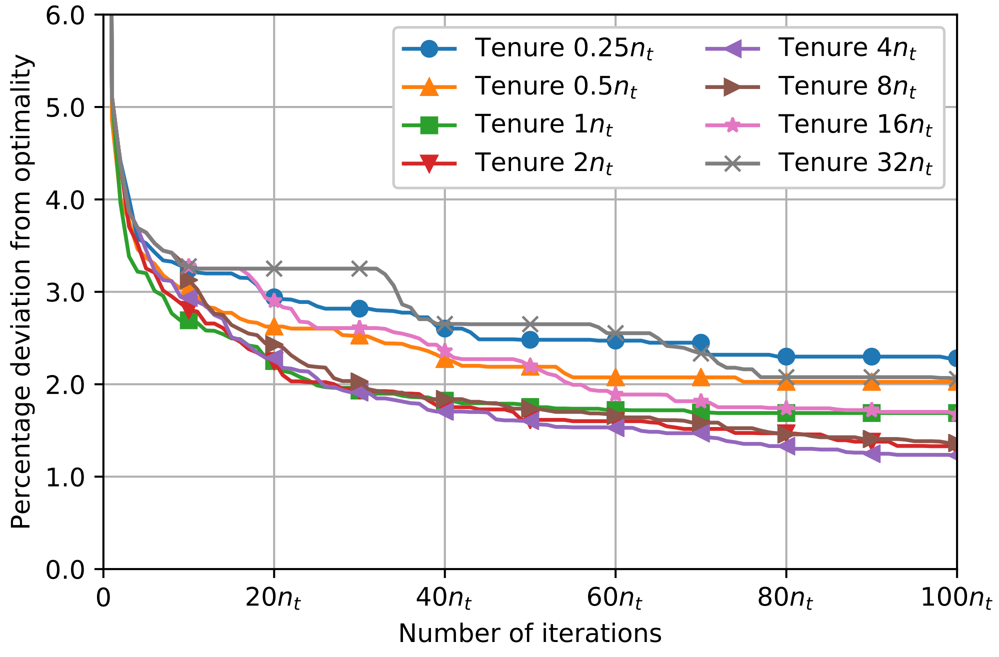


Figure B.1: Percentage deviations for the task-in-a-route attribute with different tabu tenures

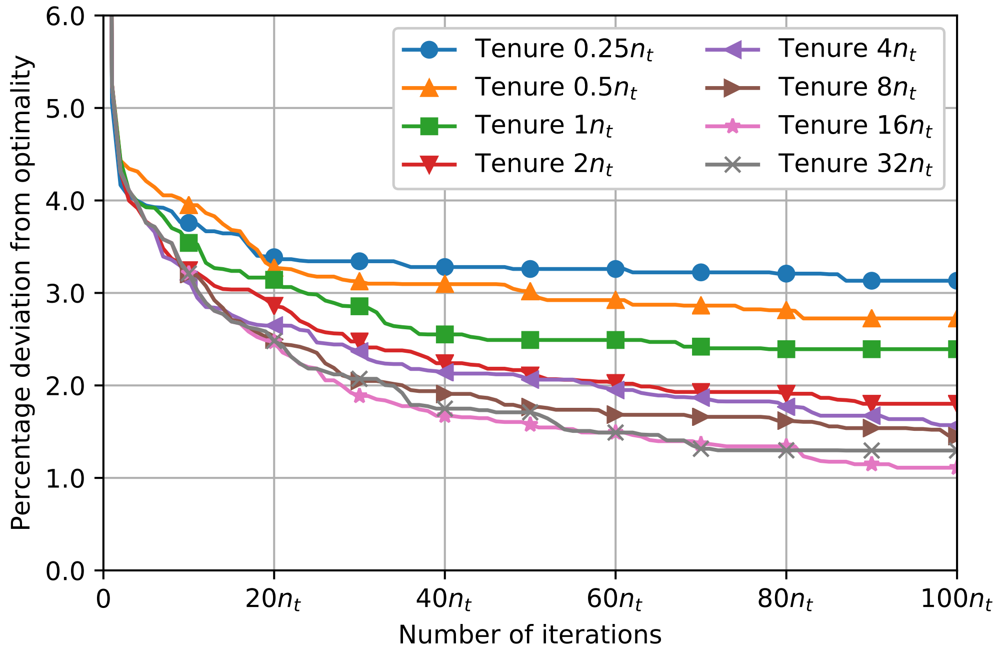


Figure B.2: Percentage deviations for the 2-task attribute with different tabu tenures

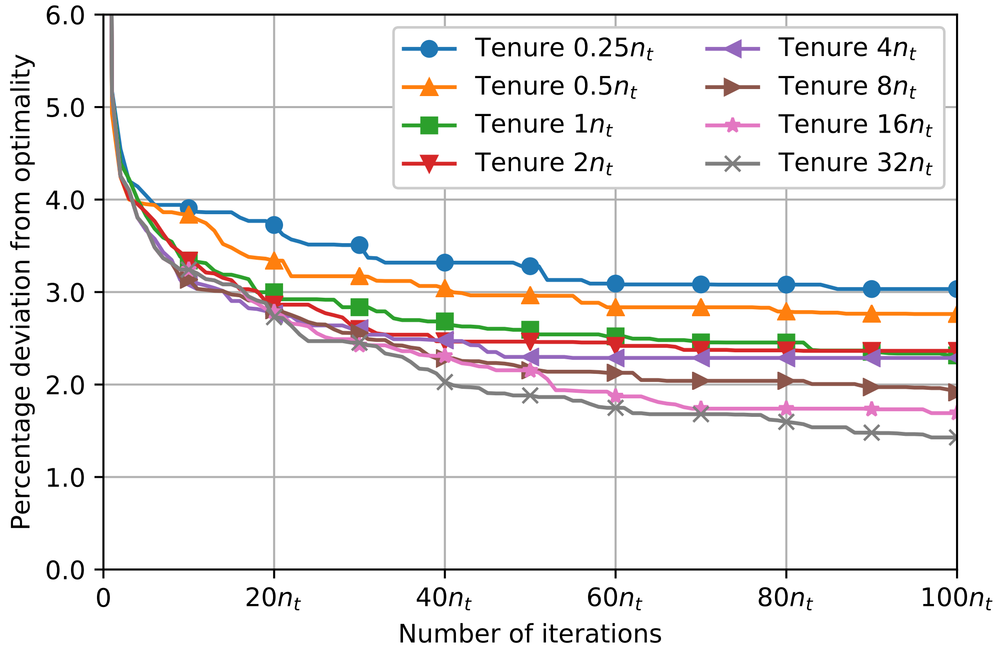


Figure B.3: Percentage deviations for the 2-task-in-a-route attribute with different tabu tenures

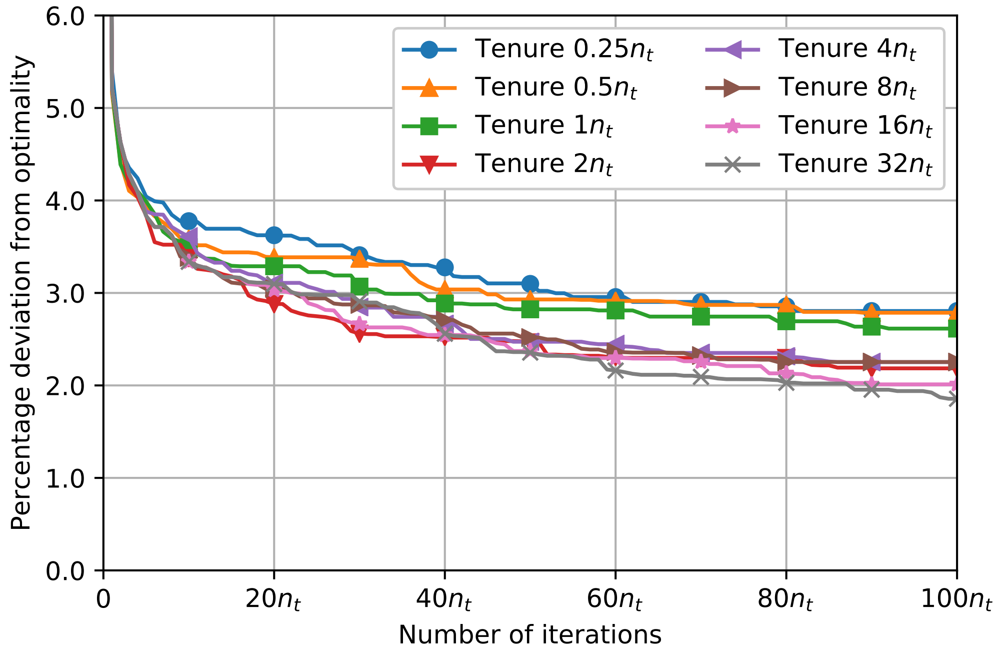
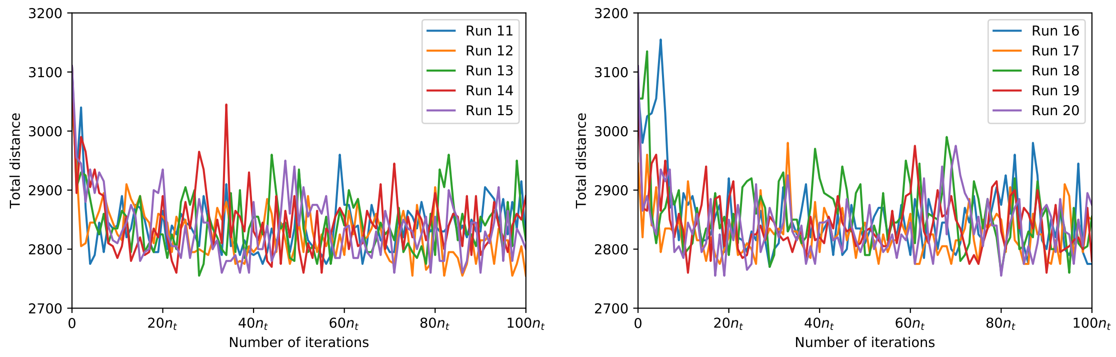
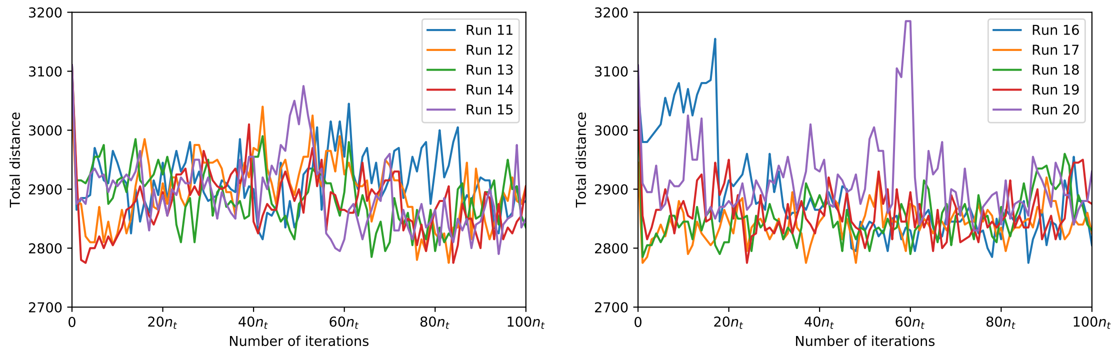


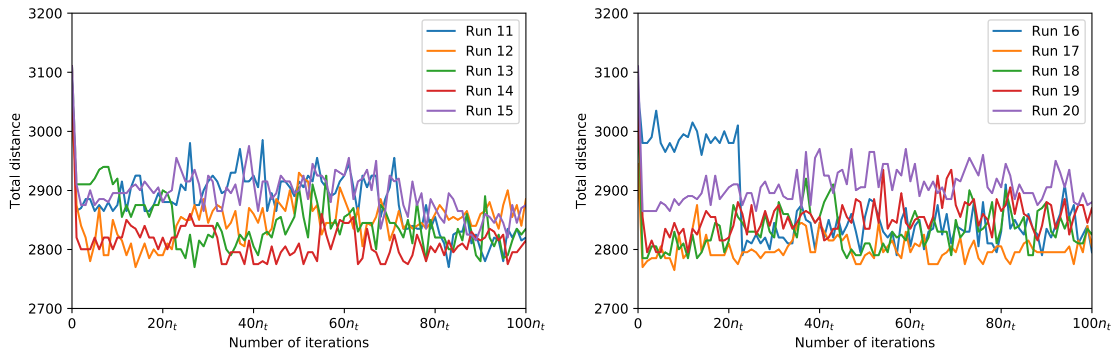
Figure B.4: Percentage deviations for the 3-task attribute with different tabu tenures



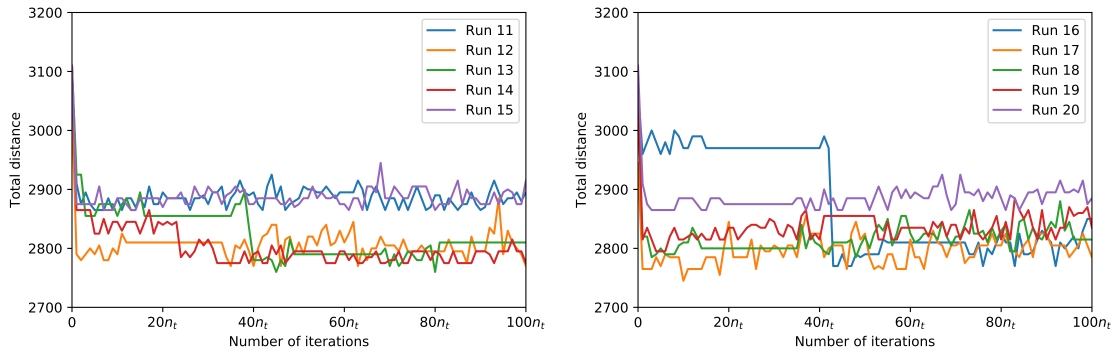
(a) Task-in-a-route



(b) 2-task



(c) 2-task-in-a-route



(d) 3-task

Figure B.5: Total distances of current solutions over the course of the tabu search algorithm from 10 sample runs ("Runs 11-20") on the E17 instance for each type of attribute; n_t denotes the number of tasks

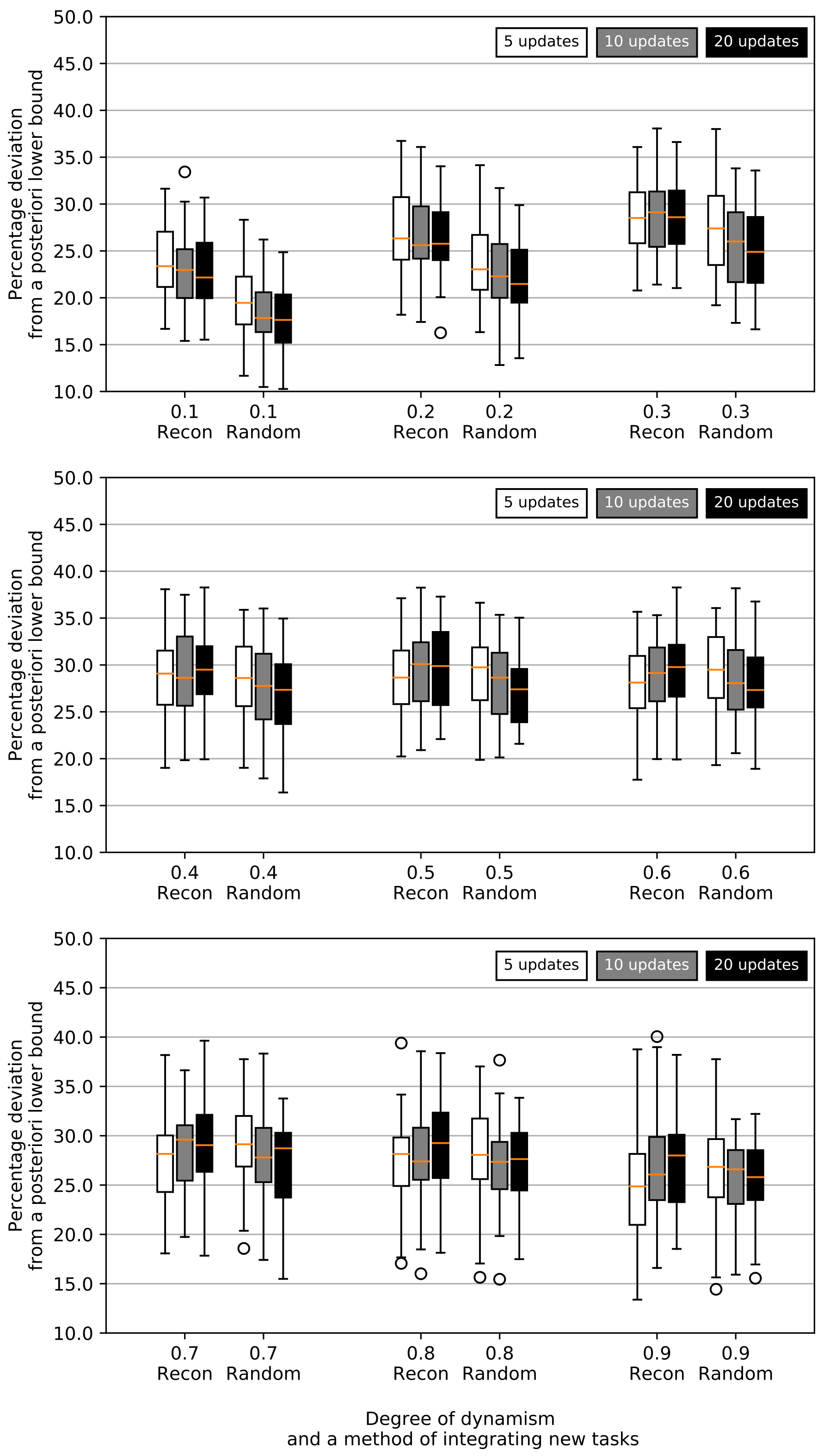


Figure B.6: Distributions of percentage deviations from a posteriori lower bounds with respect to total distances on 40 dynamic CARP instances for each degree of dynamism (0.1, 0.2, . . . , 0.9) given by the dynamic CARP solver with different update schedules and different methods of integrating tasks ("Recon" means the Reconstruction method, and "Random" means the Random Insertion method)

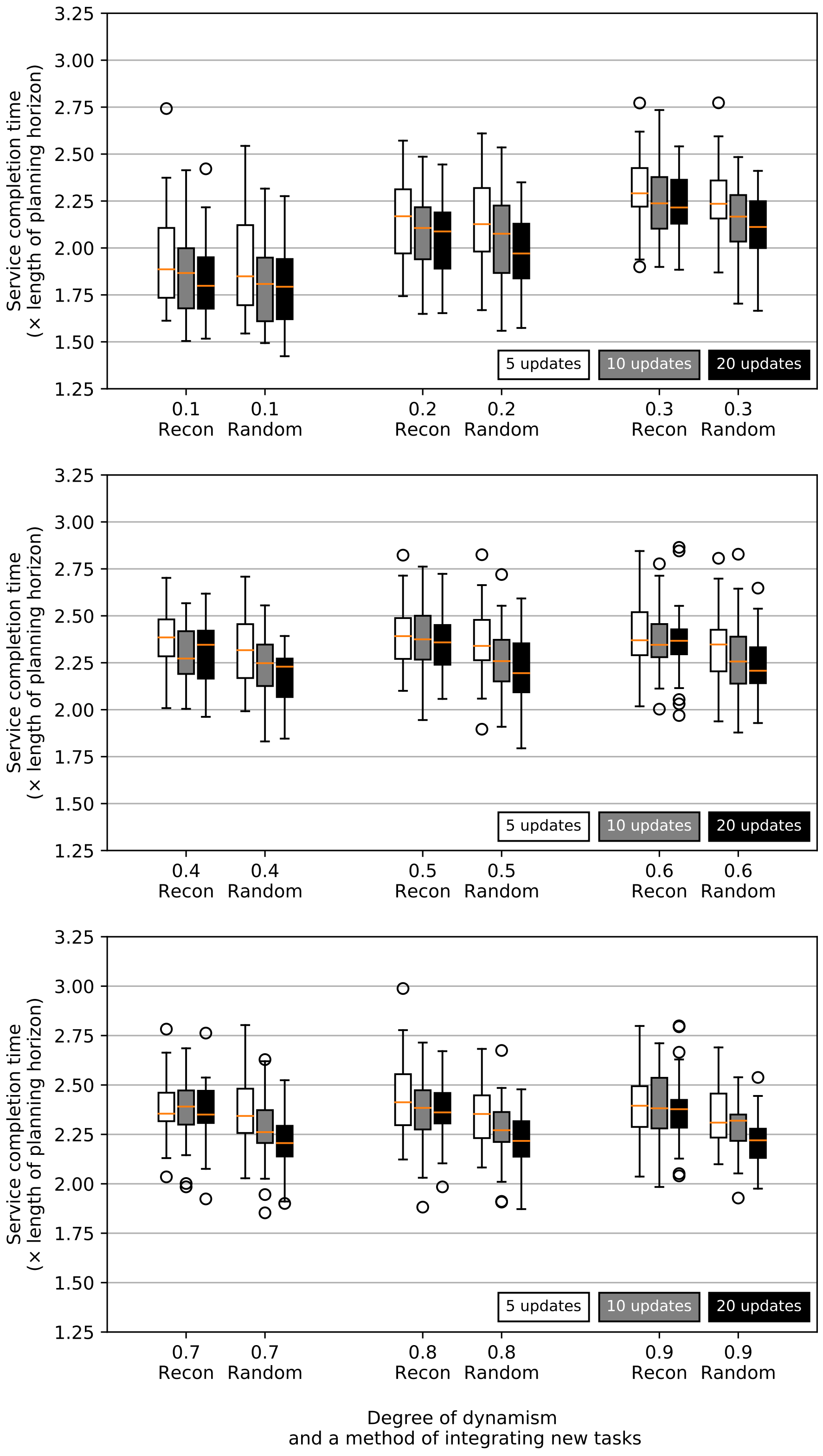
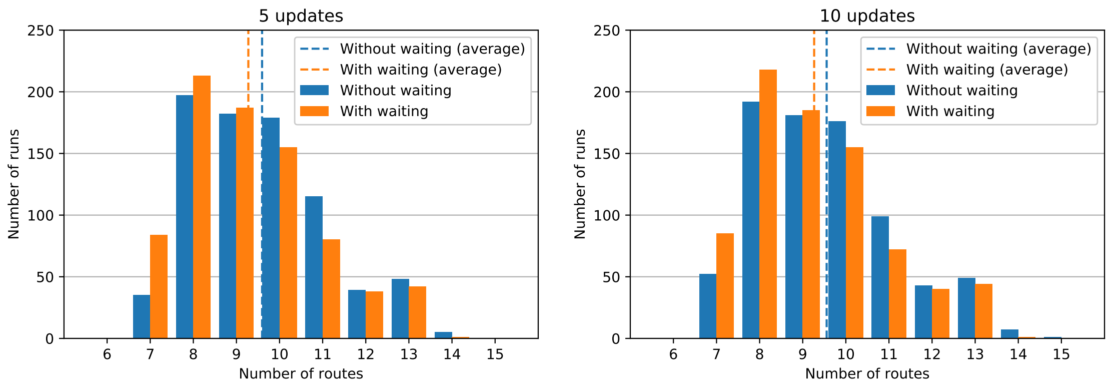
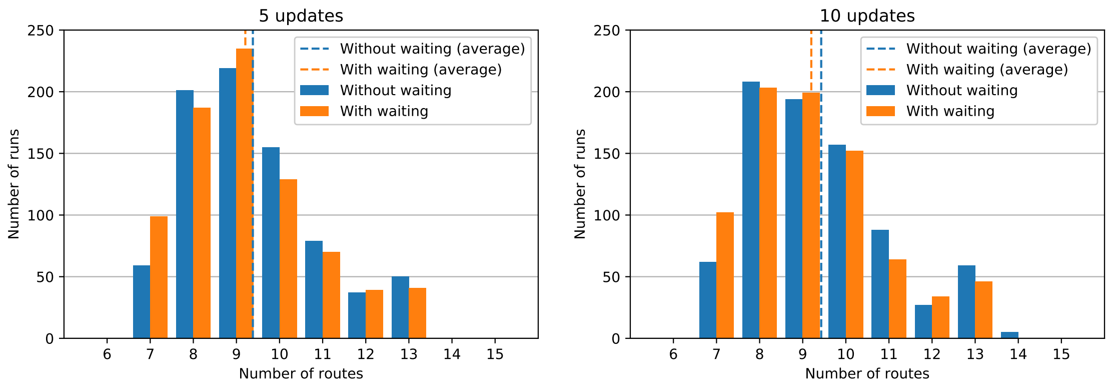


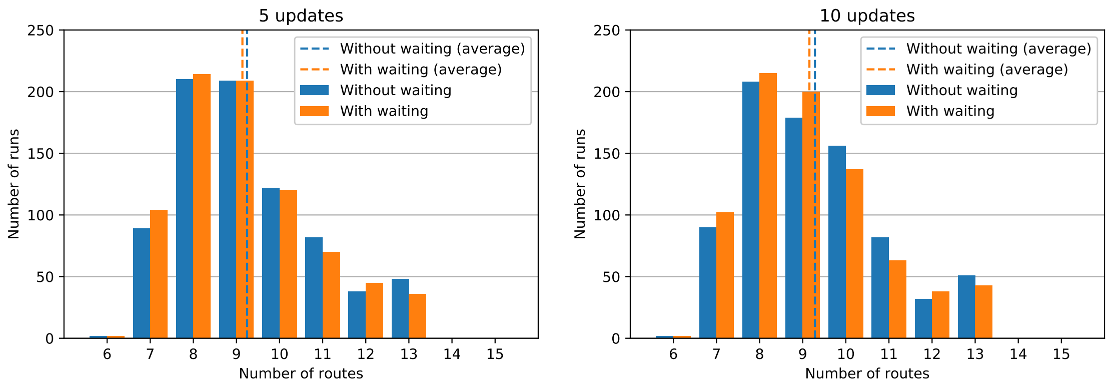
Figure B.7: Distributions of service completion times on 40 dynamic CARP instances for each degree of dynamism (0.1, 0.2, . . . , 0.9) given by the dynamic CARP solver with different update schedules and different methods of integrating tasks ("Recon" means the Reconstruction method, and "Random" means the Random Insertion method)



(a) Degree of dynamism = 0.2

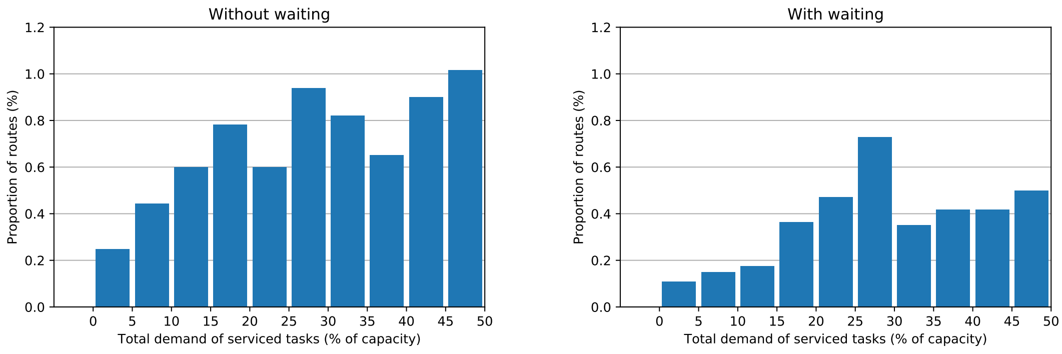


(b) Degree of dynamism = 0.5

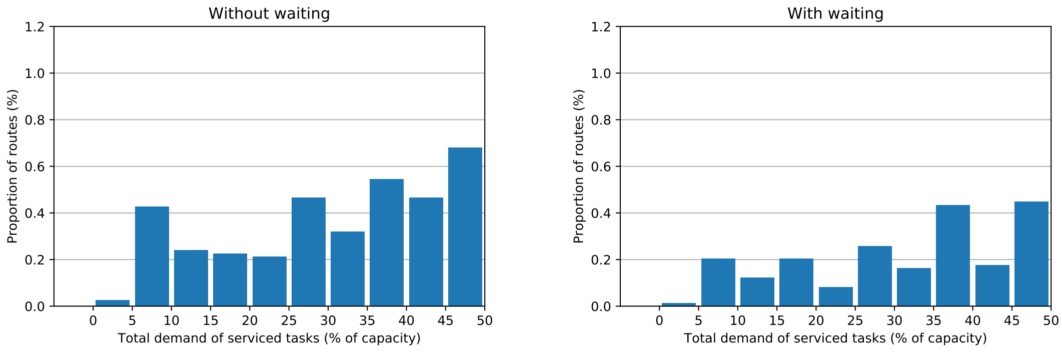


(c) Degree of dynamism = 0.8

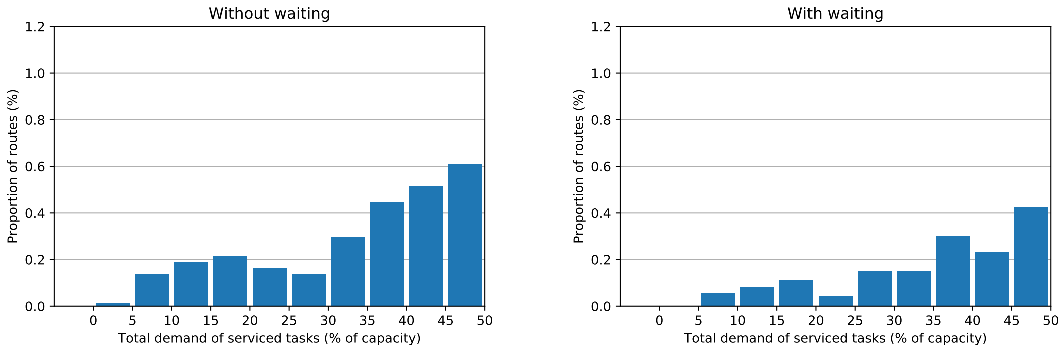
Figure B.8: The number of runs in which the dynamic CARP solver (with 5 and 10 updates) returns solutions with a given number of routes from 800 runs (40 instances × 20 runs) for each degree of dynamism; a vertical dashed line shows an average number of routes of the solutions over 800 runs



(a) Degree of dynamism = 0.2

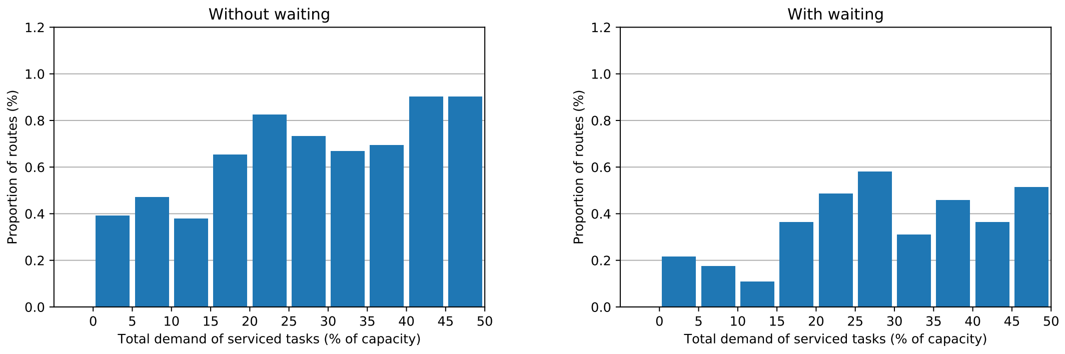


(b) Degree of dynamism = 0.5

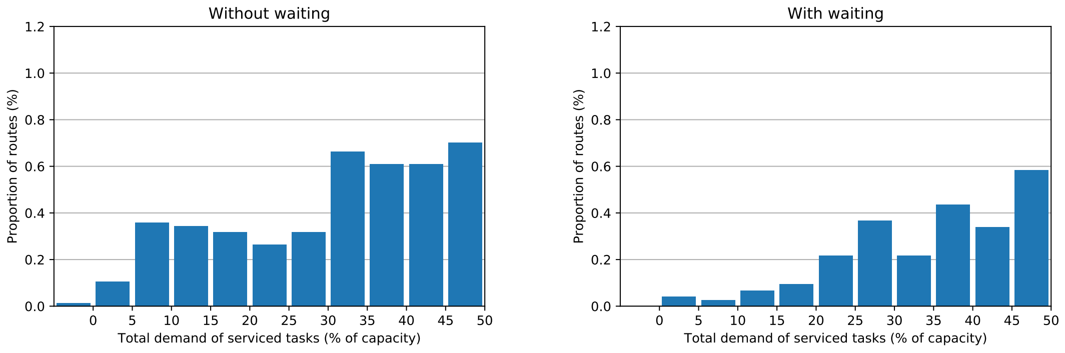


(c) Degree of dynamism = 0.8

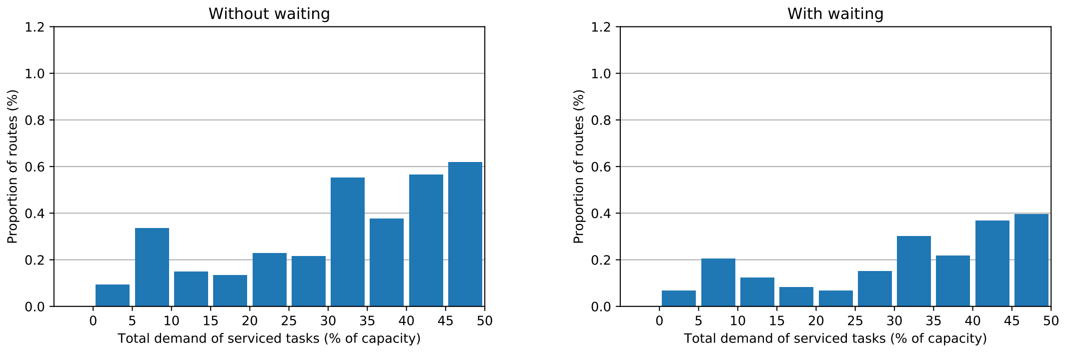
Figure B.9: Histograms showing the proportion of routes with a range of demands of serviced tasks in solutions at the end of the planning horizon given by the dynamic CARP solver (with 5 updates) without and with the waiting strategy over 800 runs (40 instances $\times$ 20 runs) for each degree of dynamism



(a) Degree of dynamism = 0.2



(b) Degree of dynamism = 0.5



(c) Degree of dynamism = 0.8

Figure B.10: Histograms showing the proportion of routes with a range of demands of serviced tasks in solutions at the end of the planning horizon given by the dynamic CARP solver (with 10 updates) without and with the waiting strategy over 800 runs (40 instances $\times$ 20 runs) for each degree of dynamism

Table B.1: Medians of percentage deviations from a posteriori lower bounds given by the dynamic CARP solver with different update schedules and different methods of integrating new tasks

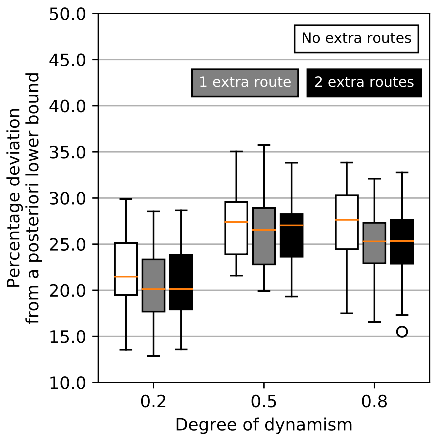
Degree of dynamism	Reconstruction			Random Insertion		
	5 updates	10 updates	20 updates	5 updates	10 updates	20 updates
0.1	23.4	23.0[a]	22.2[a]	19.5[abc]	17.8[abcd]	17.6[abcde]
0.2	26.3	25.6	25.8	23.0[abc]	22.3[abcd]	21.5[abcde]
0.3	28.5	29.1	28.6	27.4[ab]	26.0[abcd]	24.9[abcde]
0.4	29.1	28.6	29.5	28.6	27.8[bc]	27.3[abcde]
0.5	28.7	30.1	29.9	29.7	28.7[bc]	27.4[abcd]
0.6	28.1[c]	29.1	29.8	29.5	28.1	27.3[c]
0.7	28.2[bcd]	29.6	29.0	29.1	27.8[d]	28.7[bcd]
0.8	28.1	27.4	29.3	28.1	27.4[cd]	27.6[cd]
0.9	24.9[c]	26.1	28.0	26.8	26.6	25.8[c]

[a] significantly better than the Reconstruction method with 5 updates
[b] significantly better than the Reconstruction method with 10 updates
[c] significantly better than the Reconstruction method with 20 updates
[d] significantly better than the Random Insertion method with 5 updates
[e] significantly better than the Random Insertion method with 10 updates
based on a two-tailed Wilcoxon signed-rank test with a Bonferroni correction (for 15 pairwise comparisons), resulting in a significance level of $0.05/15 \approx 0.0033$

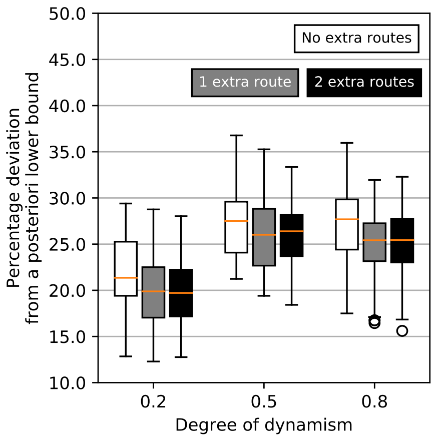
Table B.2: Medians of service completion times (as multiples of the planning horizon length) the dynamic CARP solver with different update schedules and different methods of integrating new tasks

Degree of dynamism	Reconstruction			Random Insertion		
	5 updates	10 updates	20 updates	5 updates	10 updates	20 updates
0.1	1.89	1.87[a]	1.80[ad]	1.85	1.81[ad]	1.79[abd]
0.2	2.17	2.11[ad]	2.09[ad]	2.13	2.08[ad]	1.97[abcde]
0.3	2.29	2.24[a]	2.22[a]	2.24[a]	2.17[abcd]	2.11[abcde]
0.4	2.38	2.27[a]	2.35[a]	2.32[a]	2.25[abcd]	2.23[abcd]
0.5	2.39	2.37	2.36	2.34	2.26[abcd]	2.19[abcd]
0.6	2.37	2.35	2.37	2.35	2.26[abcd]	2.21[abcd]
0.7	2.35	2.39	2.35	2.34	2.26[abcd]	2.21[abcde]
0.8	2.41	2.38	2.36[a]	2.35[a]	2.27[abcd]	2.22[abcde]
0.9	2.40	2.38	2.38	2.31	2.32[abcd]	2.22[abcde]

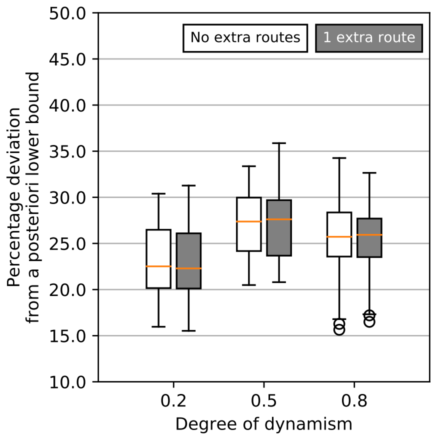
[a] significantly better than the Reconstruction method with 5 updates
[b] significantly better than the Reconstruction method with 10 updates
[c] significantly better than the Reconstruction method with 20 updates
[d] significantly better than the Random Insertion method with 5 updates
[e] significantly better than the Random Insertion method with 10 updates
based on a two-tailed Wilcoxon signed-rank test with a Bonferroni correction (for 15 pairwise comparisons), resulting in a significance level of $0.05/15 \approx 0.0033$



(a) Without waiting

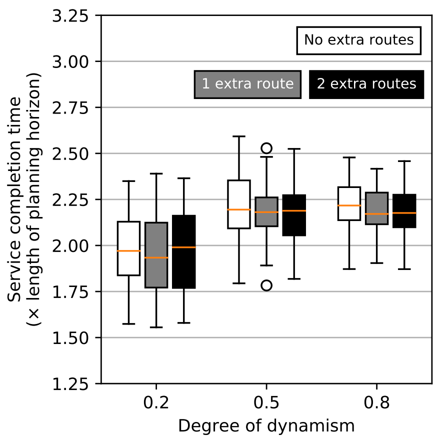


(b) Waiting at the end of last task

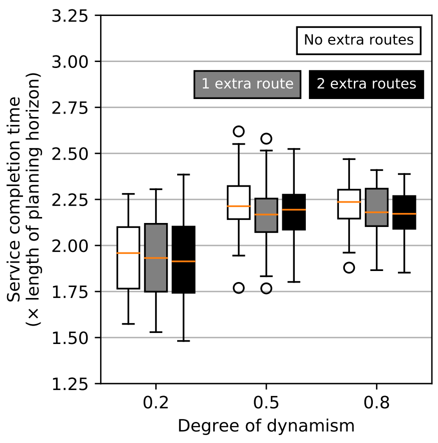


(c) Waiting away from other vehicles

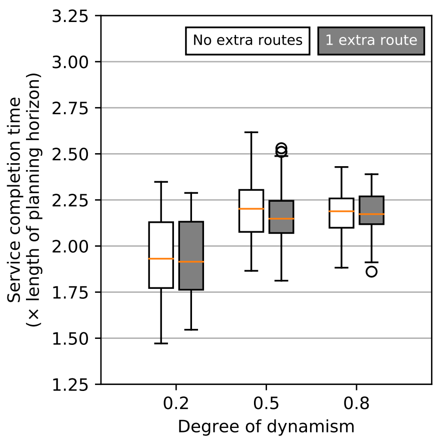
Figure B.11: Distributions of percentage deviations from a posteriori lower bounds with respect to total distances over 40 dynamic CARP instances given by the use of extra routes and different waiting strategies



(a) Without waiting



(b) Waiting at the end of last task



(c) Waiting away from other vehicles

Figure B.12: Distributions of service completion times over 40 dynamic CARP instances given by the use of extra routes and different waiting strategies

Bibliography

Amberg, A., Domschke, W., and Voß, S. (2000). Multiple center capacitated arc routing problems: A tabu search algorithm using capacitated trees. *European Journal of Operational Research*, 124(2):360–376.

Assad, A. A., Pearn, W.-L., and Golden, B. L. (1987). The capacitated chinese postman problem: Lower bounds and solvable cases. *American Journal of Mathematical and Management Sciences*, 7(1-2):63–88.

Baldacci, R. and Maniezzo, V. (2006). Exact methods based on node-routing formulations for undirected arc-routing problems. *Networks*, 47(1):52–60.

Bartolini, E., Cordeau, J.-F., and Laporte, G. (2013). Improved lower bounds and exact algorithm for the capacitated arc routing problem. *Mathematical Programming*, 137(1-2):409–452.

Belenguer, J. M. and Benavent, E. (2003). A cutting plane algorithm for the capacitated arc routing problem. *Computers & Operations Research*, 30(5):705–728.

Belenguer, J.-M., Benavent, E., Lacomme, P., and Prins, C. (2006). Lower and upper bounds for the mixed capacitated arc routing problem. *Computers & Operations Research*, 33(12):3363–3383.

Benavent, E., Campos, V., Corberán, Á., and Mota, E. (1990). The capacitated arc routing problem. a heuristic algorithm. *Qüestiió: quaderns d'estadística i investigació operativa*, 14(1).

Benavent, E., Campos, V., Corberán, A., and Mota, E. (1992). The capacitated arc routing problem: lower bounds. *Networks*, 22(7):669–690.

Beullens, P., Muyldermans, L., Cattrysse, D., and Van Oudheusden, D. (2003). A guided local search heuristic for the capacitated arc routing problem. *European Journal of Operational Research*, 147(3):629–643.

Biggs, N., Lloyd, E. K., and Wilson, R. J. (1976). *Graph Theory, 1736-1936*. Oxford University Press.

Bode, C. and Irnich, S. (2012). Cut-first branch-and-price-second for the capacitated arc-routing problem. *Operations research*, 60(5):1167–1182.

Brandão, J. (2009). A deterministic tabu search algorithm for the fleet size and mix vehicle routing problem. *European journal of operational research*, 195(3):716–728.

Brandão, J. and Eglese, R. (2008). A deterministic tabu search algorithm for the capacitated arc routing problem. *Computers & Operations Research*, 35(4):1112–1126.

Bullnheimer, B., Hartl, R. F., and Strauss, C. (1997). A new rank based version of the ant system. a computational study.

Chen, Z.-L. and Xu, H. (2006). Dynamic column generation for dynamic vehicle routing with time windows. *Transportation Science*, 40(1):74–88.

Chiang, W.-C. and Russell, R. A. (1997). A reactive tabu search metaheuristic for the vehicle routing problem with time windows. *INFORMS Journal on computing*, 9(4):417–430.

Christofides, N., Campos, V., Corberán, A., and Mota, E. (1981). An algorithm for the rural postman problem. *Report IC. OR*, 81.

Cook, W. and Rohe, A. (1999). Computing minimum-weight perfect matchings. *INFORMS Journal on Computing*, 11(2):138–148.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271.

Edmonds, J. and Johnson, E. L. (1973). Matching, euler tours and the chinese postman. *Mathematical programming*, 5(1):88–124.

Eiselt, H. A., Gendreau, M., and Laporte, G. (1995). Arc routing problems, part i: The chinese postman problem. *Operations Research*, 43(2):231–242.

Euler, L. (1741). Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, 8:128–140.

Eydi, A. and Javazi, L. (2012). A novel heuristic method to solve the capacitated arc routing problem. *International Journal of Industrial Engineering Computations*, 3(5):767–776.

Ford, L. and Fulkerson, D. R. (1962). *Flows in networks*, volume 3. Princeton Princeton University Press.

Foulds, L., Longo, H., and Martins, J. (2015). A compact transformation of arc routing problems into node routing problems. *Annals of Operations Research*, 226(1):177–200.

Fu, H., Mei, Y., Tang, K., and Zhu, Y. (2010). Memetic algorithm with heuristic candidate list strategy for capacitated arc routing problem. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE.

Fung, R. Y., Liu, R., and Jiang, Z. (2013). A memetic algorithm for the open capacitated arc routing problem. *Transportation Research Part E: Logistics and Transportation Review*, 50:53–67.

Glover, F. (1989). Tabu search—part i. *ORSA Journal on Computing*, 1(3):190–206.

Golden, B. L., DeArmon, J. S., and Baker, E. K. (1983). Computational experiments with algorithms for a class of routing problems. *Computers & Operations Research*, 10(1):47–59.

Golden, B. L. and Wong, R. T. (1981). Capacitated arc routing problems. *Networks*, 11(3):305–315.

Greistorfer, P. (2003). A tabu scatter search metaheuristic for the arc routing problem. *Computers & Industrial Engineering*, 44(2):249–266.

Hertz, A., Laporte, G., and Mittaz, M. (2000). A tabu search heuristic for the capacitated arc routing problem. *Operations research*, 48(1):129–135.

Ho, S. C. and Haugland, D. (2004). A tabu search heuristic for the vehicle routing problem with time windows and split deliveries. *Computers & Operations Research*, 31(12):1947–1964.

Holborn, P., Thompson, J., and Lewis, R. (2012). Combining heuristic and exact methods to solve the vehicle routing problem with pickups, deliveries and time windows. *Evolutionary Computation in Combinatorial Optimization*, pages 63–74.

Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer.

Labadi, N., Prins, C., and Reghioui, M. (2008). GRASP with path relinking for the capacitated arc routing problem with time windows. In *Advances in computational intelligence in transport, logistics, and supply chain management*, pages 111–135. Springer.

Lacomme, P., Prins, C., and Ramdane-Chérif, W. (2001). A genetic algorithm for the capacitated arc routing problem and its extensions. In *Workshops on Applications of Evolutionary Computation*, pages 473–483. Springer.

Lacomme, P., Prins, C., and Ramdane-Cherif, W. (2004). Competitive memetic algorithms for arc routing problems. *Annals of Operations Research*, 131(1-4):159–185.

Larsen, A. and Madsen, O. B. (2000). *The dynamic vehicle routing problem.* PhD thesis, Technical University of Denmark (DTU).

Lenstra, J. K. and Kan, A. (1976). On general routing problems. *Networks*, 6(3):273–280.

Liu, M., Singh, H. K., and Ray, T. (2014a). A benchmark generator for dynamic capacitated arc routing problems. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*, pages 579–586. IEEE.

Liu, M., Singh, H. K., and Ray, T. (2014b). A memetic algorithm with a new split scheme for solving dynamic capacitated arc routing problems. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*, pages 595–602. IEEE.

Longo, H., De Aragao, M. P., and Uchoa, E. (2006). Solving capacitated arc routing problems using a transformation to the CVRP. *Computers & Operations Research*, 33(6):1823–1837.

Martinelli, R., Pecin, D., Poggi, M., and Longo, H. (2011). A branch-cut-and-price algorithm for the capacitated arc routing problem. In *International Symposium on Experimental Algorithms*, pages 315–326. Springer.

Mei, Y., Tang, K., and Yao, X. (2009). A global repair operator for capacitated arc routing problem. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 39(3):723–734.

Monroy-Licht, M., Amaya, C. A., Langevin, A., and Rousseau, L.-M. (2016). The rescheduling arc routing problem. *International Transactions in Operational Research.*

Montané, F. A. T. and Galvao, R. D. (2006). A tabu search algorithm for the vehicle routing problem with simultaneous pick-up and delivery service. *Computers & Operations Research*, 33(3):595–619.

Montemanni, R., Gambardella, L. M., Rizzoli, A. E., and Donati, A. V. (2005). Ant colony system for a dynamic vehicle routing problem. *Journal of Combinatorial Optimization*, 10(4):327–343.

Moreira, L. M., Oliveira, J. F., Gomes, A. M., and Ferreira, J. S. (2007). Heuristics for a dynamic rural postman problem. *Computers & operations research*, 34(11):3281–3294.

Pearn, W. L. (1988). New lower bounds for the capacitated arc routing problem. *Networks*, 18(3):181–191.

Pearn, W. L. (1991). Augment-insert algorithms for the capacitated arc routing problem. *Computers & Operations Research*, 18(2):189–198.

Pearn, W. L., Assad, A., and Golden, B. L. (1987). Transforming arc routing into node routing problems. *Computers & Operations Research*, 14(4):285–288.

Pearn, W. L. and Wu, T. (1995). Algorithms for the rural postman problem. *Computers & Operations Research*, 22(8):819–828.

Prins, C. (2004). A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers & Operations Research*, 31(12):1985–2002.

Psaraftis, H. N. (1980). A dynamic programming solution to the single vehicle many-to-many immediate request dial-a-ride problem. *Transportation Science*, 14(2):130–154.

Santos, L., Coutinho-Rodrigues, J., and Current, J. R. (2009). An improved heuristic for the capacitated arc routing problem. *Computers & Operations Research*, 36(9):2632–2637.

Santos, L., Coutinho-Rodrigues, J., and Current, J. R. (2010). An improved ant colony optimization based algorithm for the capacitated arc routing problem. *Transportation Research Part B: Methodological*, 44(2):246–266.

Saruwatari, Y., Hirabayashi, R., and Nishida, N. (1992). Node duplication lower bounds for the capacitated arc routing problem. *Journal of the Operations Research Society of Japan*, 35(2):119–133.

Tagmouti, M., Gendreau, M., and Potvin, J.-Y. (2011). A dynamic capacitated arc routing problem with time-dependent service costs. *Transportation Research Part C: Emerging Technologies*, 19(1):20–28.

Talbi, E.-G. (2009). *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons.

Tang, K., Mei, Y., and Yao, X. (2009). Memetic algorithm with extended neighborhood search for capacitated arc routing problems. *IEEE Transactions on Evolutionary Computation*, 13(5):1151–1166.

Ulusoy, G. (1985). The fleet size and mix problem for capacitated arc routing. *European Journal of Operational Research*, 22(3):329–337.

Usberti, F. L., França, P. M., and França, A. L. M. (2011). The open capacitated arc routing problem. *Computers & Operations Research*, 38(11):1543–1555.

Vansteenwegen, P., Souffriau, W., and Sörensen, K. (2010). Solving the mobile mapping van problem: A hybrid metaheuristic for capacitated arc routing with soft time windows. *Computers & operations research*, 37(11):1870–1876.

Voudouris, C. and Tsang, E. (1996). Partial constraint satisfaction problems and guided local search. *Proc., Practical Application of Constraint Technology (PACT'96), London*, pages 337–356.

Yazici, A., Kirlik, G., Parlaktuna, O., and Sipahioglu, A. (2014). A dynamic path planning approach for multirobot sensor-based coverage considering energy constraints. *IEEE transactions on cybernetics*, 44(3):305–314.