Parallelization of the Vehicle Routing Problem with Time Windows

Jesper Larsen

LYNGBY 1999
Ph.D. THESIS
NO. 62

IMM

ISSN 0909-3192

Printed by IMM - DTU Technical University of Denmark

Preface

This Ph.D. thesis entitled "Parallelization of the Vehicle Routing Problem with Time Windows" has been prepared by Jesper Larsen during the period June 1995 to May 1999, partly at the Department of Computer Science (DIKU) at the University of Copenhagen, partly at the Department of Mathematical Modelling (IMM) at the Technical University of Denmark (DTU).

The Ph.D. project has been supervised by my main advisor professor Jens Clausen and associate professor Oli B. G. Madsen. The subjects of the thesis are improving the performance of the sequential algorithm for the VRPTW and parallelization of the vehicle routing problem with time windows. The thesis is submitted as a partial fulfillment of the requirement for obtaining the Ph.D. degree at the Technical University of Denmark.

The project was supported by the EPOS (**E**fficient **P**arallel algorithms for **O**ptimization and **S**imulation) project of the Danish NSF (Statens Naturvidenskabelige Forskningsråd).

Acknowledgements

A Ph.D. project is not a one-man project, therefore I would first and foremost like to thank my supervisors Jens Clausen and Oli B. G. Madsen for their support and encouragements throughout the project. Without them this project would definitely not have been possible. I would also like to thank my former colleagues from the Algorithmics and Optimization group at the Department of Computer Science, University of Copenhagen, and my present colleagues from the section of Operations Research at the Department of Mathematical Modelling, Technical University of Denmark. Especially I would like to thank Stefan Karisch for a positive office-partnership during the 6 months we shared the same office. I would also like to thank Niklas Kohl, Ph.D. from the Department of Mathematical Modelling, for quick responses to my numerous e-mails on his code and the VRPTW in general.

A special thanks goes "down under" to New Zealand. My $5\frac{1}{2}$ month stay at the Department of Engineering Science at the University of Auckland, New Zealand, was in every respect wonderful. Professor David Ryan delivered invaluable support and supervising. Additionally I would like to thank the entire Ryan family for letting me become a natural part of 10 Ngapuhi Road for the $5\frac{1}{2}$ months.

I would like to thank Stefan Tschöke, Georg Kliewer and Torsten Fahle from the University of Paderborn, Germany, for their support. Although all production runs of my parallel code was made on the IBM SP2 at UNI•C, it was a significant help for me to be able to build and test the code on the parallel computers at the University of Paderborn, at a central moment where CPLEX was not available at the IBM SP2, and I always got an answer on my numerous emails, when I was stuck with some inane error message. Thanks to them the parallel code was ready for productions runs as soon as the necessary software was available on the IBM SP2.

Finally I would like to thank friends and family for their support during the almost 4 years it took to complete this thesis. During a project that lasts almost 4 years there are bound to be periods where things aren't working as they are supposed to do. In these periods encouragement and comfort from friends and family was never missing.

Lyngby, May 1999.

Jesper Larsen

Summary

Routing with time windows (VRPTW) has been an area of research that have attracted many researchers within the last 10 – 15 years. In this period a number of papers and technical reports have been published on the exact solution of the VRPTW.

The VRPTW is a generalization of the well-known capacitated routing problem (VRP or CVRP). In the VRP a fleet of vehicles must visit (service) a number of customers. All vehicles start and end at the depot. For each pair of customers or customer and depot there is a cost. The cost denotes how much is costs a vehicle to drive from one customer to another. Every customer must be visited exactly ones. Additionally each customer demands a certain quantity of goods delivered (know as the customer demand). For the vehicles we have an upper limit on the amount of goods that can be carried (known as the capacity). In the most basic case all vehicles are of the same type and hence have the same capacity. The problem is now for a given scenario to plan routes for the vehicles in accordance with the mentioned constraints such that the cost accumulated on the routes, the fixed costs (how much does it cost to maintain a vehicle) or a combination hereof is minimized.

In the more general VRPTW each customer has a time window, and between all pairs of customers or a customer and the depot we have a travel time. The vehicles now have to comply with the additional constraint that servicing of the customers can only be started within the time windows of the customers. It is legal to arrive before a time window "opens" but the vehicle must wait and service will not start until the time window of the

customer actually opens.

For solving the problem exactly 4 general types of solution methods have evolved in the literature: dynamic programming, Dantzig-Wolfe (column generation), Lagrange decomposition and solving the classical model formulation directly.

Presently the algorithms that uses Dantzig-Wolfe given the best results (Desrochers, Desrosiers and Solomon, and Kohl), but the Ph.D. thesis of Kontoravdis shows promising results for using the classical model formulation directly.

In this Ph.D. project we have used the Dantzig-Wolfe method. In the Dantzig-Wolfe method the problem is split into two problems: a "master problem" and a "subproblem". The master problem is a relaxed set partitioning problem that guarantees that each customer is visited exactly ones, while the subproblem is a shortest path problem with additional constraints (capacity and time window). Using the master problem the reduced costs are computed for each arc, and these costs are then used in the subproblem in order to generate routes from the depot and back to the depot again. The best (improving) routes are then returned to the master problem and entered into the relaxed set partitioning problem. As the set partitioning problem is relaxed by removing the integer constraints the solution is seldomly integral therefore the Dantzig-Wolfe method is embedded in a separation-based solution-technique.

In this Ph.D. project we have been trying to exploit structural properties in order to speed up execution times, and we have been using parallel computers to be able to solve problems faster or solve larger problems.

The thesis starts with a review of previous work within the field of VRPTW both with respect to heuristic solution methods and exact (optimal) methods. Through a series of experimental tests we seek to define and examine a number of structural characteristics.

The first series of tests examine the use of dividing time windows as the branching principle in the separation-based solution-technique. Instead of using the methods previously described in the literature for dividing a problem into smaller problems we use a methods developed for a variant of the VRPTW. The results are unfortunately not positive.

Instead of dividing a problem into two smaller problems and try to solve these we can try to get an integer solution without having to branch. A cut is an inequality that separates the (non-integral) optimal solution from all the integer solutions. By finding and inserting cuts we can try to avoid branching. For the VRPTW Kohl has developed the 2-path cuts. In the separationalgorithm for detecting 2-path cuts a number of test are made. By structuring the order in which we try to generate cuts we achieved very positive results.

In the Dantzig-Wolfe process a large number of columns may be generated, but a significant fraction of the columns introduced will not be interesting with respect to the master problem. It is a priori not possible to determine which columns are attractive and which are not, but if a column does not become part of the basis of the relaxed set partitioning problem we consider it to be of no benefit for the solution process. These columns are subsequently removed from the master problem. Experiments demonstrate a significant cut of the running time.

Positive results were also achieved by stopping the route-generation process prematurely in the case of time-consuming shortest path computations. Often this leads to stopping the shortest path subroutine in cases where the information (from the dual variables) leads to "bad" routes. The premature exit from the shortest path subroutine restricts the generation of "bad" routes significantly. This produces very good results and has made it possible to solve problem instances not solved to optimality before.

The parallel algorithm is based upon the sequential Dantzig-Wolfe based algorithm developed earlier in the project. In an initial (sequential) phase unsolved problems are generated and when there are unsolved problems enough to start work on every processor the parallel solution phase is initiated. In the parallel phase each processor runs the sequential algorithm. To get a good workload a strategy based on balancing the load between neighbouring processors is implemented. The resulting algorithm is efficient and capable of attaining good speedup values. The loadbalancing strategy shows an even distribution of work among the processors. Due to the large demand for using the IBM SP2 parallel computer at UNI•C it has unfortunately not be possible to run as many tests as we would have liked. We have although managed to solve one problem not solved before using our parallel algorithm.

Resumé (in Danish)

Ruteplanlægning med tidsvinduer (VRPTW) har inden for de sidste 10 – 15 år optaget mange forskere. Der er i denne periode publiceret mange artikler og rapporter inden for emnet eksakt løsning af VRPTW.

VRPTW er en generalisering af det velkendte ruteplanlægningsproblem med kapacitetsbegrænsninger (VRP eller CVRP). I VRP skal en flåde af biler besøge en række kunder. Bilerne starter og slutter deres ruter i et depot, og for hver direkte forbindelse mellem to kunder eller en kunde og depotet er der fastlagt en omkostning. Hver kunde skal besøges af præcis en bil. Desuden ønsker hver kunde en bestemt mængde varer leveret. Den samlede mængde varer, der kan transporteres af en bil er begrænset af bilens kapacitet. For et givet scenario ønskes at minimere den omkostning i form af kørt afstand som bilerne akkumulere under deres kørsel, de faste omkostninger (hvor meget koster det at holde en bil kørende) eller en kombination heraf.

I VRPTW tildeles hver kunde et tidsvindue samt rejsetider mellem hver af kunderne og mellem kunderne og depotet. En bil skal nu betjene kunden indenfor det givne tidsvindue. Kommer bilen før tidsvinduets start må den vente indtil kundens tidsvindue "åbner".

Indenfor eksakte metoder til løsning af VRPTW er der i litteraturen blevet beskrevet 4 mulige metoder: dynamisk programmering, Dantzig-Wolfe (søjle-generering), Lagrange dekomposition og direkte løsning af den klassiske modelformulering.

Til dato har algoritmer, der bygger på Dantzig-Wolfe givet de bedste re-

sultater (Desrochers, Desrosiers og Solomon, og Kohl), men Kontoravdis' ph.d.-afhandling, hvori der arbejdes direkte med den klassiske modelformulering ser lovende ud.

I Dantzig-Wolfe metoden, som benyttes i dette ph.d.-projekt, opdeles problemet i 2 delproblemer: et "master problem" og et "subproblem". Master problemet er et relaxeret klassedelingsproblem som sikrer, at hver kunde besøges præcis en gang, mens subproblemet et er korteste-vej-problem, som tager hensyn til kapacitetsbegrænsninger og overholdelse af tidsvinduer. Vha. master problemet beregnes de reducerede omkostninger for de enkelte direkte forbindelser mellem kunderne (og kunder og depotet). Disse bruges så i subproblemet til at beregne den/de korteste veje fra depot og tilbage til depotet. De bedste ruter returneres til master problemet, der tilføjer ruterne som søjler i det relaxerede klassedelingsproblem. Eftersom man ikke er garanteret en heltallig løsning indsættes Dantzig-Wolfe metoden i en separations-baseret løsningsmetode.

I denne afhandling er der dels arbejdet med udnyttelse af problemstrukturer til at give hurtigere løsningstider, dels brug af parallelcomputere for at kunne løse større problemer eller løse problemer hurtigere.

Afhandlingen indledes med en gennemgang af eksakte og heuristiske metoder for VRPTW, samt en teoretisk gennemgang af problemet. Igennem en lang række eksperimentelle afprøvninger søges en række strukturelle egenskaber defineret og undersøgt.

Undersøgelserne starter med en række tests af brug af tidsvinduerne som forgreningskriterie i en separations-baseret løsningsmetode. I stedet for at bruge de klassiske metoder til deling af et ikke løst delproblem i to mindre delproblemer udføres en metode beskrevet for en variant af VRPTW. Resultaterne er desværre ikke positive.

I stedet for at opdele et delproblem i to mindre delproblemer kan man gennem tilføjelse af gyldige uligheder, dvs. uligheder der afskærer den bedst fundne ikke-heltallige løsning fra alle heltalsløsningerne, søge at opnå en heltalsløsning uden brug af forgreningsteknikken. Til VRPTW er de såkaldte "2-path uligheder" udviklet af Kohl. I forbindelse med brug af 2-path uligheder udføres en række tests med henblik på effektivisering af separationsalgoritmen. Her opnås meget positive resultater i forbindelse med en ordnet gennemgang af mulige gyldige uligheder.

I Dantzig-Wolfe processen dannes en mængde ruter, og en stor del af disse vil ikke være interessante i forhold til master problemet. Det er a priori svært at se hvilke søjler der ikke er interessante, men hvis en søjle ikke opnår at indgå i en basis til løsningen af et LP-relaxeret klassedelingsproblem, må den skønnes at være uden nytte. Disse kan fjernes fra master problemet. Eksperimenter viser, at dette resulterer i en væsentlig nedgang i algoritmens køretid.

Positive resultater er også opnået ved at stoppe rutegenererings processen tidligere i tilfælde af lange køretider for korteste-vej-algoritmen. Ofte bliver processen stoppet i tilfælde, hvor manglende informationer giver "dårlige" ruter. Dermed undgås dannelsen af mange dårlige ruter. Denne idé giver meget gode resultater og har muliggjort løsningen af problemer, der ikke tidligere har været løst til optimalitet.

Den parallelle algoritme tager sit udgangspunkt i den sekventielle Dantzig-Wolfe baserede algoritme udviklede tidligere i projektet. Efter en initiel fase, hvor der dannes et antal endnu uløste delproblemer fordeles uløste delproblemer på alle processorer. Herefter sker problemløsning parallelt. For at sikre en ligelig lastfordeling, er der implementeret en strategi til lastfordeling. Den resulterende parallelle algoritme er effektiv og i stand til at opnå gode speedups. Lastfordelings-strategien fremviser en meget jævn fordeling af delproblemer imellem processorene. Grundet det store pres på UNI•C's SP2 parallelcomputer har det ikke været muligt at udføre ret mange eksperimenter på endnu uløste problemer. Det er dog lykkedes vha. den parallelle algoritme at løse et probleminstans som aldrig før har været løst til optimalitet.

List of Abbreviations

Abbreviations	Meaning	First occurrence
CVRP	Capacitated Vehicle Routing Problem	3
ESPP	Elementary Shortest Path Problem	16
ESPPCC	Elementary Shortest Path Problem with Capacity Constraints	16
ESPPTW	Elementary Shortest Path Problem with Time Windows	16
ESPPTWCC	Elementary Shortest Path Problem with Time Windows and Capacity Constraints	16
GAP	Generalized Assignment Problem	26
GLS	Guided Local Search	44
GRASP	Greedy Randomized Adaptive Search Procedure	43
MCVRPTW	Multi Compartment Vehicle Routing Problem with Time Windows	96
MDVRPTW	Multi Depot Vehicle Routing Problem with Time Windows	94
MIMD	Multiple Instruction-stream Multiple Data-stream	103
MISD	Multiple Instruction-stream Single Data-stream	105

Continued on the next page

Continued from the previous page

Abbreviations	Meaning	First occurrence
MPI	Message Passing Interface	7
MPP	Massively Parallel Processing	107
PRAM	Parallel Random Access Machine	102
PVM	Parallel Virtual Machine	7
RAM	Random Access Machine	102
SAP	Semi Assignment Problem	26
SIMD	Single Instruction-stream Multiple Data-stream	103
SISD	Single Instruction-stream Single Data-stream	103
SPP	Shortest Path Problem	80
SPPTW	Shortest Path Problem with Time Windows	85
SPPTWMCC	Shortest Path Problem with Time Windows and Multiple Capacity Constraints	96
TSP	Travelling Salesmans Problem	3
TSPTW	Travelling Salesmans Problem with Time Windows	14
VRP	Vehicle Routing Problem	3
VRPBTW	Vehicle Routing Problem with Back-hauling and Time Windows	97
VRPLC	Vehicle Routing Problem with Length Constraint	5
VRPTW	Vehicle Routing Problem with Time Windows	4

Contents

Chapter 1

Introduction

All obstacles in life are mere opportunities.
- (written on a bench at Mt. Hobson park, Auckland)

In modelling of routing problems terminology is to a great extent derived from graph theory. Notions like vertex, node, arc, path etc. will not be explained. In general it is assumed that the reader is familiar with the concepts of graph theory and linear programming.

The notation presented and used throughout this thesis is identical to the notation used in [Koh95].

1.1 Motivation

In the real world many companies are faced with problems regarding the transportation of people, goods or information – commonly denoted routing problems. This is not restricted to the transport sector itself but also other companies e.g. factories may have transport of parts to and from different sites of the factory, and big companies may have internal mail deliveries. These companies have to optimize transportation. As the world economy turns more and more global, transportation will become even more important in the future.

Back in 1983 Bodin et al. in [BGAB83] reported that in 1980 approximately \$400 billion were used in distribution cost in the United States and in the United Kingdom the corresponding figure was £15 billion. Halse reports in [Hal92] from an article from the Danish newspaper Berlingske Tidende that in 1989 76.5% of all the transportation of goods was done by vehicles, which underlines the importance of routing and scheduling problems. Fisher writes in [Fis97] that a study from the National Council of Physical Distribution estimates that transportation accounts for 15% of the U.S. gross national product (1978). In Denmark the figures are 13% for 1981 and 15% for 1994 according to [The98].

Therefore solving different kinds of routing and scheduling problems is an important area of Operations Research (OR). Cutting even a small fraction of the costs may result in large savings and reduce the strain on the environment caused by pollution and noise. In [Bod90] a number of successful applications made over the past 20 years are mentioned.

In a *pure* routing problem there is only a *geographic* component, while more realistic routing problems also include a scheduling part, that is, a time component.

The problems in research are often more simple than real-life problems. But even though a number of real-life constraints are left out (e.g. constraints forced by legislation, trade unions or nature) the research models typically model the basic properties and thereby provide the core results used in the analysis and implementation of systems in real-life problems. Therefore a number of basic models exist that researchers agree are important investigating. These models will briefly be introduced here.

One of the best known routing problem is at the same time the simplest one namely the Traveling Salesman Problem (or TSP). A number of cities have to be visited by a salesman who must return to the same city where he started. The route has to be constructed in order to minimize the distance to be traveled. A typical solution to a TSP problem is shown in figure 1.1. This problem often acts as a test bed for new ideas or paradigms before moving on to the more advanced models.

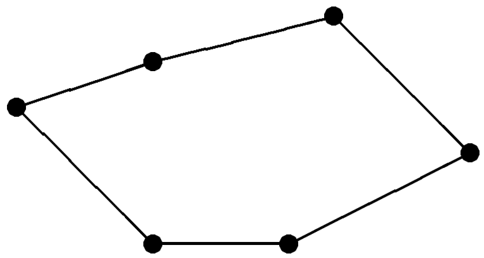


Figure 1.1: A typical solution to a TSP instance.

In the m-TSP problem, m salesmen have to cover the cities given. Each city must be visited by exactly one salesman. Every salesman starts off from the same city (called the depot) and must at the end of his journey return to this city again. We now want to minimize the sum of the distances of the routes. Both the TSP and m-TSP problems are pure routing problems in the sense defined above.

The Vehicle Routing Problem (or VRP) is the m-TSP where a demand is associated with each city, and each vehicle have a certain capacity (not necessarily identical). For a survey of the VRP refer to [Lap97, Gol84]. Be aware that during the later years a number of authors have "renamed" this problem the Capacitated Vehicle Routing Problem (or CVRP). The sum of demands on a route can not exceed the capacity of the vehicle assigned to this route. As in the m-TSP we want to minimize the sum of distances of the routes. Note that the VRP is not purely geographic since the demand may be constraining. The VRP is the basic model for a large number of vehicle routing problems. We mention the most important here. In figure 1.2 a typical solution to the VRP is shown. Note that the direction in which the route(s) are driven is unimportant both for the TSP and the VRP .

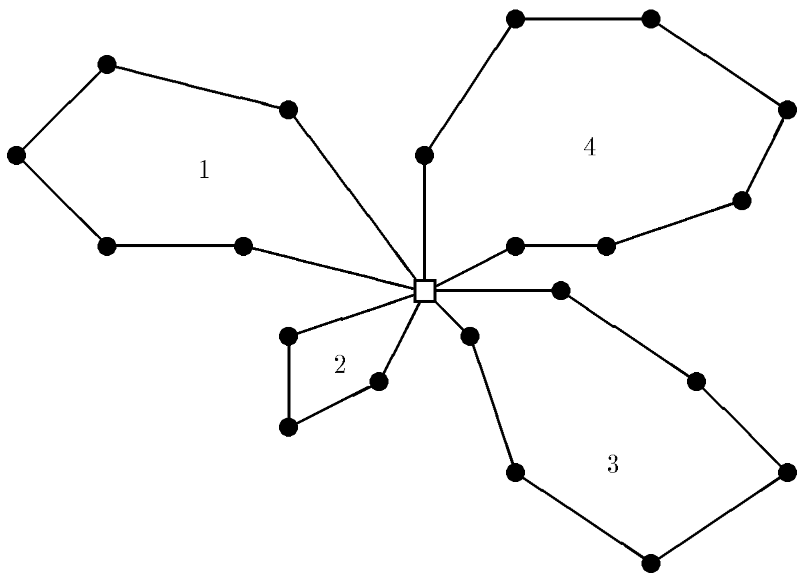


Figure 1.2: A typical solution to a VRP instance (4 routes). The square denotes the depot.

If we add a *time window* to each customer we get the Vehicle Routing Problem with Time Windows (VRPTW). In addition to the capacity constraint, a vehicle now has to visit a customer within a certain time frame. The vehicle may arrive before the time window "opens" but the customer can not be serviced until the time windows "opens". It is not allowed to arrive after the time window has "closed". Some models allow for early or late servicing but with some form of additional cost or penalty. These models are denoted "soft" time window models (see e.g. [Bal93]). By far the most research has been made on "hard" time window models and this is the problem dealt with in this thesis.

Another variant of the VRP is the Vehicle Routing Problem with Length Constraint (VRPLC). Here each route is not allowed to exceed a given distance. Other variants could be to include more than one depot, or more than one type of items to be delivered. In the *Split Delivery* model the demand of a customer is not necessarily covered by just one vehicle but may be split between two or more. The solutions obtained in a split delivery model will always be at least as good as for the "normal" VRP and often we may be able to utilize the vehicles better and thereby save vehicles. Finally we mention the *Pickup and Delivery* variant where the vehicles not only deliver items but also pick up items during the routes. This problem can be varied even more according to whether the deliveries must be completed before starting to pick up items or the two phases can be interleaved.

These problems are all "hard" to solve (a more formal complexity analysis is given in section 2.2). For the VRPTW exact solutions can be found within reasonable time for some instances up to about 100 customers. A review of exact methods for the VRPTW is given in section 2.3.

As indicated above, often the number of customers combined with the complexity of real-life data does not permit solving the problem exactly. In these situations one can apply approximation algorithms or heuristics. Both approximation algorithms and heuristics produce a feasible but not necessarily optimal solution. Whereas a worst-case deviation is known for approximation algorithms nothing a priori is known for heuristics, but typically they can be tuned to perform very well. These non-exact methods for the VRPTW will be reviewed in section 2.4.

If the term "vehicle" is considered more loosely, numerous scheduling problems can also be regarded as VRPTW. An example is that for a single machine, we want to schedule a number of jobs where we know the flow time and the time to go from running one job to the next one. This scheduling problem can be regarded as a VRPTW with a single depot, single vehicle and the customers represents the jobs. The cost of changing from one job to another is equal to the distance between the two customers. The time is takes to perform the action is the service time of the job.

For a general and in-depth description of the field of routing and scheduling see [DDSS93, Bre95, CL98].

1.2 Combinatorial Optimization

The problems that will be investigated in this thesis are all *optimization problems*. An optimization problem can generally be formulated as minimizing or maximizing (from now on we will only consider minimization) the value of a function f called the *objective function*. The variables x_i (for $i = 1, 2, \ldots, x_k$) are called *decision variables* (alternatively we write $x = (x_1, x_2, x_3, \ldots, x_k)$). A number of constraints determines the set of *feasible solutions* $\mathcal{S}$. $\mathcal{S}$ is called the *solutions space*. An optimization problem can then be stated as:

$$\begin{array}{l} \min f(x) \\ \textit{subject to } x \in \mathcal{S} \end{array}$$

The solution set $\mathcal{S}$ is typically described implicitly by a set of equations and inequalities to be fulfilled by feasible solutions. It is important to note that a given optimization problem can be formulated in a number of different ways with respect to objective function, decision variables and description of the solution space. Different formulations of the *same* problem can in practice yield very different algorithms for solving the problem and consequently different computational effects.

Combinatorial optimization problems are optimization problems for which the set of feasible solutions are finite, but usually exponentially large or countably infinite, as a function of the problem data. Efficient algorithms exist for some of these, while others seem solvable only by methods requiring exponential time. Problems of the latter type are called $\mathcal{NP}$-hard problems (for a further discussion on complexity issues see section 2.2 or refer to [GJ79]).

1.3 Why go parallel?

For the VRPTW optimal algorithms of today can solve problems up to about 100 customers. In order to push the limit further we have a number of methods in our tool-box. One of the methods is to use parallel computers, thereby increasing the computational power beyond what is available with sequential computers. A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem, a definition that includes both "real" parallel computers but also sequential computers in a network.

Besides solving previously unsolved problems another attractive aspect is the ability to solve problems faster. If it takes weeks or months before a solution to a problem is found it becomes hard to use the results in research that demands a lot of experiments for example because of tuning of the algorithm. But if it successfully can be parallelized the running time may be cut to days or even hours which makes it possible to run several tests.

But parallel computing is also interesting from the perspective of the industry. In highly time-critical environments programs on even the fastest sequential computer may not be fast enough. Here the parallel computer may be an alternative to cutting down on accuracy, problem size or flexibility.

One of the main problems of parallel computing is that a uniform model of computing does not exist, as a number of new parameters are introduced.

Another problem used to be that each parallel computer had its own set of commands for controlling the parallelism making it very difficult and expensive to port to another platform.

From the beginning of the 90's a number of message passing interfaces for parallel computers started to emerge as for example P4, PARMACS and PICL. These were macros and library routines so portability was often achieved by compromising performance, but they made migration from one platform to another, if not simple, then at least a lot easier than before.

With the development of *de facto* standards as Parallel Virtual Machines (PVM, see [GBD+94]) and Message Passing Interface (MPI, see [GLS94, Pac95]) this process has been made considerably more easy. As these became de facto standards the vendors have constructed portable and efficient implementations of the message passing paradigm. With MPI one can start off using a network of workstations as a parallel computer (e.g. at night when nobody else are using them). If the project then turns out to be a success or the financial support becomes available a "real" parallel computer can take over the job with a minimal effort in moving the code.

The target machine for the parallel programs developed in this thesis will be MIMD parallel computers with distributed memory (the terminology will be explained in chapter 6).

1.4 Outline of the thesis

In chapter 2 a mathematical model for the VRPTW is presented. Furthermore complexity results and a review of relevant literature is presented.

The sequential algorithm is presented in the chapters 3 and 4, and in chapter 6 the parallel algorithm is described.

Chapter 7 contains a description of the experimental setup and the results obtained for the experiments made with the sequential algorithm, while chapter 8 contains the tests made using the parallel algorithm. Finally the thesis ends with a conclusion in chapter 9.

1.5 Overview of contribution of this thesis

The two main contributions of this thesis are the development and implementation of a parallel algorithm for the VRPTW and a thorough investigation in the characteristics of the execution of a column-generation-based VRPTW algorithm. The analysis resulted in techniques for significant reduction of the running time. Among the developed techniques are new methods for constructing sets for 2-path cuts, removal of unused columns and premature stop of the route generating subroutine. These contributions have made it possible to solve instances from the Solomon test-set that have not previously been solved to optimality.

Other contributions are made on the investigation of resource constrained branching and use of a heuristic for solving the "feasibility TSPTW" as a subroutine in generating 2-path cuts.

Chapter 2

The Vehicle Routing Problem with Time Windows

In this chapter we formalize the loose description of the VRPTW as it was presented in the introduction. We state the problem mathematically and discuss complexity issues. Finally we look briefly at related research both for optimal algorithms and for heuristics and approximation algorithms. Huge amounts of results exist for the VRP. The amount of research done on the VRPTW is growing as an acknowledgement to the importance of the problem.

2.1 A mathematical model of the VRPTW

The VRPTW is given by a fleet of homogeneous vehicles (denoted $\mathcal{V}$), a set of customers $\mathcal{C}$ and a directed graph $\mathcal{G}$. The graph consists of $|\mathcal{C}|+2$ vertices, where the customers are denoted $1, 2, \dots, n$ and the depot is represented by the vertex 0 ("the driving-out depot") and $n+1$ ("the returning depot").

The set of vertices, that is, $0, 1, \ldots, n+1$ is denoted $\mathcal{N}$. The set of arcs (denoted $\mathcal{A}$) represents connections between the depot and the customers and among the customers. No arc terminates in vertex 0, and no arc originates from vertex $n+1$. With each arc (i, j), where $i \neq j$, we associate a *cost* c_{ij} and a *time* t_{ij}, which may include service time at customer i.

Each vehicle has a capacity q and each customer i a demand d_i. Each customer i has a *time window* $[a_i, b_i]$. A vehicle must arrive at the customer before b_i. It can arrive before a_i but the customer will not be serviced before. The depot also has a time window $[a_0, b_0]$ (the time windows for both depots are assumed to be identical). $[a_0, b_0]$ is called the *scheduling horizon*. Vehicles may not leave the depot before a_0 and must be back before or at time b_{n+1}.

It is assumed that q, a_i, b_i, d_i, c_{ij} are non-negative integers, while the t_{ij}'s are assumed to be positive integers (reasons for this assumption are a bit technical and will be discussed in chapter 4). It is also assumed that the triangular inequality is satisfied for both the c_{ij}'s and the t_{ij}'s.

The model contains two sets of decision variables x and s. For each arc (i, j), where $i \neq j, i \neq n+1, j \neq 0$, and each vehicle k we define x_{ijk} as

$$x_{ijk} = \begin{cases} 0, & \text{if vehicle } k \text{ does not drive from vertex } i \text{ to vertex } j \\ 1, & \text{if vehicle } k \text{ drives from vertex } i \text{ to vertex } j \end{cases}$$

The decision variable s_{ik} is defined for each vertex i and each vehicle k and denotes the time vehicle k starts to service customer i. In case the given vehicle k does not service customer i s_{ik} does not mean anything. We assume $a_0 = 0$ and therefore $s_{0k} = 0$, for all k.

We want to design a set of minimal cost routes, one for each vehicle, such that

- each customer is serviced exactly once,
- every route originates at vertex 0 and ends at vertex $n+1$, and
- the time windows and capacity constraints are observed.

We can state the VRPTW mathematically as:

$$\min \sum_{k\in\mathcal{V}} \sum_{i\in\mathcal{N}} \sum_{j\in\mathcal{N}} c_{ij}x_{ijk} \qquad s.t. \tag{2.1}$$

$$\sum_{k\in\mathcal{V}} \sum_{j\in\mathcal{N}} x_{ijk} = 1 \qquad \forall i \in \mathcal{C} \tag{2.2}$$

$$\sum_{i\in\mathcal{C}} d_i \sum_{j\in\mathcal{N}} x_{ijk} \leq q \qquad \forall k \in \mathcal{V} \tag{2.3}$$

$$\sum_{j\in\mathcal{N}} x_{0jk} = 1 \qquad \forall k \in \mathcal{V} \tag{2.4}$$

$$\sum_{i\in\mathcal{N}} x_{ihk} - \sum_{j\in\mathcal{N}} x_{hjk} = 0 \qquad \forall h \in \mathcal{C}, \forall k \in \mathcal{V} \tag{2.5}$$

$$\sum_{i\in\mathcal{N}} x_{i,n+1,k} = 1 \qquad \forall k \in \mathcal{V} \tag{2.6}$$

$$s_{ik} + t_{ij} - K(1 - x_{ijk}) \leq s_{jk} \qquad \forall i,j \in \mathcal{N}, \forall k \in \mathcal{V} \tag{2.7}$$

$$a_i \leq s_{ik} \leq b_i \qquad \forall i \in \mathcal{N}, \forall k \in \mathcal{V} \tag{2.8}$$

$$x_{ijk} \in \{0,1\} \qquad \forall i,j \in \mathcal{N}, \forall k \in \mathcal{V} \tag{2.9}$$

The constraints (2.2) states that each customer is visited exactly once, and (2.3) means that no vehicle is loaded with more than it's capacity allows it to. The next three equations (2.4), (2.5) and (2.6) ensures that each vehicle leaves the depot 0, after arriving at a customer the vehicle leaves again, and finally arrives at the depot $n+1$. The inequalities (2.7) states that a vehicle k can not arrive at j before $s_{ik} + t_{ij}$ if it is traveling from i to j. Here K is a large scalar. Finally constraints (2.8) ensures that time windows are observed, and (2.9) are the integrality constraints. Note that an unused vehicle is modelled by driving the "empty" route $(0, n+1)$.

As mentioned earlier VRPTW is a generalization if both TSP and VRP. In case the time constraints ((2.7) and (2.8)) are not binding the problem becomes a VRP. This can be achieved by setting $a_i = 0$ and $b_i = M$ (where M is a large scalar) for all customers i. It should be noted that the time variables enable us to formulate the VRP without subtour elimination constraints (described later). If only one vehicle is available the problem becomes a TSP. If more vehicles are available and additionally $c_{0j} = 1, j \in \mathcal{C}$ and $c_{ij} = 0$ otherwise we get the *Bin-packing problem*. As the order in which we visit the customers become unimportant (due to the "free" trips) the objective becomes to "squeeze" as much goods into as few vehicles (bins) as possible.

In case the capacity constraints (2.3) are not binding the problem becomes a m-TSPTW (again if only one vehicle is available we get a TSPTW).

Finally when we remove the assignment constraints (2.2) the problem becomes a Elementary Shortest Path Problem with Time Windows and Capacity Constraints (ESPPTWCC) for every vehicle, that is, find the shortest path from the depot and back to the depot that does not violate the time and capacity constraints and visits the customers on the route at most one time. As all vehicles are identical all ESPPTWCC's also become identical.

2.2 Complexity issues

In this section we discuss the computational complexity of the VRPTW and related problems. First though we give a small review of complexity theory.

An algorithm is a general step-by-step procedure for solving problems. For our purposes we can think of it as being a computer program. Generally we are interested in finding the most "efficient" algorithm for a given problem. The problem is how to measure the efficiency. Here, as in almost everywhere in the literature, we focus on efficiency with respect to the running time of the algorithm. We measure the time requirement in terms of the "size" of the problem instance. So in order to do this we need to specify an *encoding scheme* for the problem. The *input length* of an instance is then defined to be the number of symbols in the description of the instance. The *time complexity function* for an algorithm expresses the largest time requirement for each possible input length.

An algorithm is said to have *polynomial time complexity* if the running time is of order $O(n^k)$, where n denotes the *input length* and k is a constant independent of n. If the time complexity function can not be bounded by a polynomial the algorithm is said to have *exponential time complexity*. If the expression g^l, where l is a constant and g is the largest input value, is part of the bounding function then the algorithm is said to have *Pseudo-polynomial running time*. Hence, if an upper bound is imposed on g the algorithm becomes polynomial. In many algorithms used in practical situations such bounds may arise naturally, and in these cases the pseudo-polynomial algorithm may be just as good as a polynomial algorithm.

A *decision problem* is a problem that can be answered with "yes" or "no". $\mathcal{P}$ denotes the class of decision problems which can be solved in polynomial time, and $\mathcal{NP}$ is the class of decision problems solvable in *nondeterministic polynomial time*, that is, the problem can be solved in polynomial time by a nondeterministic Turing machine (computer). The nondeterministic computer can be viewed as a computer capable of executing an unbounded (but finite) number of computations in parallel.

A problem $\mathcal{X}$ is said to be $\mathcal{NP}$*-complete*, if any problem in $\mathcal{NP}$ can be transformed to $\mathcal{X}$ in polynomial time ($\mathcal{X}$ is said to belong to the class $\mathcal{NPC}$). So in a sense the $\mathcal{NP}$*-complete* problems constitutes a class of the "hardest" problems in $\mathcal{NP}$. If just one problem in $\mathcal{NPC}$ is solvable in polynomial time then by transitivity every problem can be solved in polynomial time. Obviously $\mathcal{P} \subseteq \mathcal{NP}$, but whether $\mathcal{P} = \mathcal{NP}$ holds or not is unknown. There are though many reasons to believe that $\mathcal{P} \neq \mathcal{NP}$. Finally a problem is $\mathcal{NP} - complete$ in the *strong sense* if no pseudo-polynomial algorithm exists unless $\mathcal{P} = \mathcal{NP}$.

Given a decision problem $\mathcal{Y}$, whether a member of $\mathcal{NP}$ or not. If a $\mathcal{NP}$*-complete* problem can be transformed to $\mathcal{Y}$, $\mathcal{Y}$ can not be solved in polynomial time (unless of course $\mathcal{P} = \mathcal{NP}$). The problem $\mathcal{Y}$ is at least as hard as the $\mathcal{NP}$*-complete* problems and therefore $\mathcal{Y}$ is called $\mathcal{NP}$*-hard*.

The VRPTW contains several $\mathcal{NP}$*-hard* optimization problems implying that VRPTW is also $\mathcal{NP}$*-hard*. Among the $\mathcal{NP}$*-hard* problems contained as special cases are TSP ([GJ79, problem ND22] and [LK81]), Bin Packing ([GJ79, problem SR1]) and VRP ([LK81]).

Even finding a feasible solution to VRPTW with a fixed number of vehicles is $\mathcal{NP}$*-hard* in the strong sense (see [Koh95]). If the number of vehicles available is unlimited feasibility amounts to determine whether a solution consisting of the routes *depot* - i - *depot*, $\forall i \in \mathcal{C}$, is feasible. This is an easy task and can be done in $O(n)$ time.

For the shortest path problems we know that the "normal" Shortest Path Problem (SPP) is polynomial. It can be solved in $O(nm)$ by the Bellmann-Ford-Moore algorithm (here n denotes the number of vertices and m the number of edges) (see e.g. [CGR93]).

Adding constraints that imposes each vertex must to be visited at most one time results in the Elementary Shortest Path Problem (ESPP). Adding capacity constraints defines the problem denoted ESPPCC, time windows gives us ESPPTW and finally adding both capacity constraints and time windows results in the ESPPTWCC. As the following proposition shows these problems are all $\mathcal{NP}$-hard in the strong sense.

Proposition 2.1 ([Koh95]) *The ESPPTWCC, ESPPCC, ESPPTW and ESPP are $\mathcal{NP}$-hard in the strong sense.*

Proof: *Let us consider the ESPP. As this problem is a special case of the remaining problems it suffices to prove the proposition.*

The proposition will be proven by proving that if a pseudo-polynomial algorithm exists for the ESPP then an pseudo-polynomial algorithm would also exist for the Directed Hamiltonian Circuit Problem*. This problem is stated as $\mathcal{NP}$-complete in the strong sense in [GJ79, problem GT38]. Thereby we would get $\mathcal{P} = \mathcal{NP}$.*

The Directed Hamiltonian Circuit Problem is given by a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{A})$. The question is whether $\mathcal{G}$ contains a Hamiltonian circuit.

Consider the instance of ESPP given by the fully connected directed graph $\mathcal{G}' = (\mathcal{N}, \mathcal{A}')$. Let us denote one vertex s as the origin. Now we associate a cost *c_e with each edge $e \in \mathcal{E}'$ as*

$$c_e = \begin{cases} -1, & \textit{if } e \in \mathcal{E} \\ 0, & \textit{otherwise} \end{cases}$$

Now if and only if the objective function value of the ESPP from s and back to s again is $-|\mathcal{N}|$, then $\mathcal{G}$ contains a Hamiltonian circuit. $\square$

We can make the problems somewhat easier by relaxing the "elementary" constraint as the following proposition states.

Proposition 2.2 *SPPTWCC, SPPTW and SPPCC are NP-hard, but solvable by a pseudo-polynomial algorithm if*

1. $t_{ij} > 0, \ \forall i, j \in \mathcal{N}$, *and*
2. $d_i > 0, \ \forall i \in \mathcal{C}$

Proof: *In [DS88a] an algorithm for the SPPTW problem is developed. The algorithm has a time complexity of* $O(\min\{md, nD\})$, *where* n *is the number of vertices and* m *the number of arcs. The widest time windows, that is,* $\max_{i\in\mathcal{N}}\{b_i - a_i + 1\}$ *is denoted by* d, *and* D *is the number of possible labels. So this algorithm is pseudo-polynomial. Along the same lines one can add another constraint and still get a pseudo-polynomial algorithm.*

Note that for SPPTWCC only one of the conditions has to be satisfied, for the two other problems one of the conditions is irrelevant. $\square$

2.3 Review of optimal algorithms

The first paper proposing an exact algorithm for solving the VRPTW was published back in 1987 in [KRT87]. Since then a number of papers have been published and almost all the algorithms use one of three principles:

1. Dynamic Programming.
2. Lagrange Relaxation-based methods.
3. Column Generation.

The method based on *column generation* is implemented as part of this Ph.D. project, as at the moment it looks to be the best of the available methods. Therefore a thorough discussion is presented in chapter 3. It has been implemented and described previously in [DDS92, Koh95].

Most of the approaches rely on the solution of a shortest paths problem with additional constraints.

A different approach is described in the Ph.D. thesis [Kon97] by Kontoravdis. This will be described later in this section.

The research on the VRPTW has been surveyed in the papers [BGAB83, SD88, DLSS88, DDSS93].

Presently no parallel implementation of the algorithms for the VRPTW is known to the author.

2.3.1 Dynamic Programming

The dynamic programming approach for VRPTW is presented for the first (and only) time in [KRT87]. The paper is inspired by a earlier paper [CMT81] where Christofides et al. use the dynamic programming paradigm to solve the VRP.

The algorithm of Kolen et al. use Branch-and-Bound to achieve optimality. Each node α in the Branch-and-Bound tree corresponds to three sets: $F(\alpha)$ which is the set of fixed feasible routes starting and finishing at the depot, $P(\alpha)$ which is a partially build route starting at the depot, and $C(\alpha)$ denotes the set of customers forbidden to be next on $P(\alpha)$.

Branching is done by selecting a customer i that is not forbidden, that is $i \notin C(\alpha)$, and that does not appear on any route, that is $i \notin F(\alpha) \cup P(\alpha)$. Branching decisions are taken on route-customer allocations. Then two branches are generated: one in which the partially build route $P(\alpha)$ is extended by i and one where i is forbidden as the next customer on the route, that is, i is added to $C(\alpha)$. Customer i is chosen as the customer the partial route $P(\alpha)$ was extended with in the calculation that lead to the lower bound of node α.

At each Branch-and-Bound node dynamic programming is used to calculate a lower bound on all feasible solutions defined by $F(\alpha), P(\alpha)$ and $C(\alpha)$.

First we discuss the case of the root node ($F(\alpha) = \emptyset, C(\alpha) = \emptyset$ and $P(\alpha) =$ *depot*). Here we construct a directed graph with vertices $v(i, q, k)$ for $i = 0, 1, \ldots, n$, $q = 0, 1, \ldots, Q$ and $k = 0, 1, \ldots, m$, where n is the number of customers, m the number of vehicles and Q is the sum of all customer demands q_i. Hence, associated with each Branch-and-Bound node is a set of routes.

A directed path from $v(0, 0, 0)$ to $v(i, q, k)$ in the graph corresponds to a set of k routes with a total load of q and with different last visited customers (each one in $\{1, 2, \ldots, i\}$). The arc lengths in the directed graph will be defined as the total length of the corresponding routes. The lower bound is then given by the minimum over $k = 1, 2, \ldots, m$ of the shortest paths lengths from $v(0, 0, 0)$ to $v(n, Q, k)$. Note that there are no constraints enforcing customers (not belonging to either $F(\alpha)$ or $P(\alpha)$) to be visited by any of the routes generated. Therefore the resulting minimum is a lower bound.

Dynamically we try to extend a set of k routes with load q and last customers $\{1, 2, \ldots, i\}$ to last customers $\{1, 2, \ldots, i + 1\}$. Here there are two possibilities:

- Customer $i + 1$ is not included as endpoint of any of the routes. This results in an arc from $v(i, q, k)$ to $v(i + 1, q, k)$ with length 0.

  Note that a customer $i + 1$ that is not an endpoint might still be member of one of the other routes generated by the function $F(i, q)$ described below.

- Insert customer $i + 1$ as the last customer on a route of load q'. This generates an arc from $v(i, q, k)$ to $v(i + 1, q + q', k + 1)$ of length $F(i + 1, q')$ for each possible value of q' (see figure 2.1). Generally, $F(i, q)$ is defined as the minimum length of a feasible route with total load q and last customer i. This problem is a shortest path problem with side constraints. It is relaxed to allow a customer to be serviced more than once and is solved by an "extended" version of the Dijkstra algorithm (as explained in chapter 4).

  The routes associated with a given vertex $v(i, q, k)$ are the routes given from the computation of $F(i, q)$ and the extension made using the arcs in the graph.

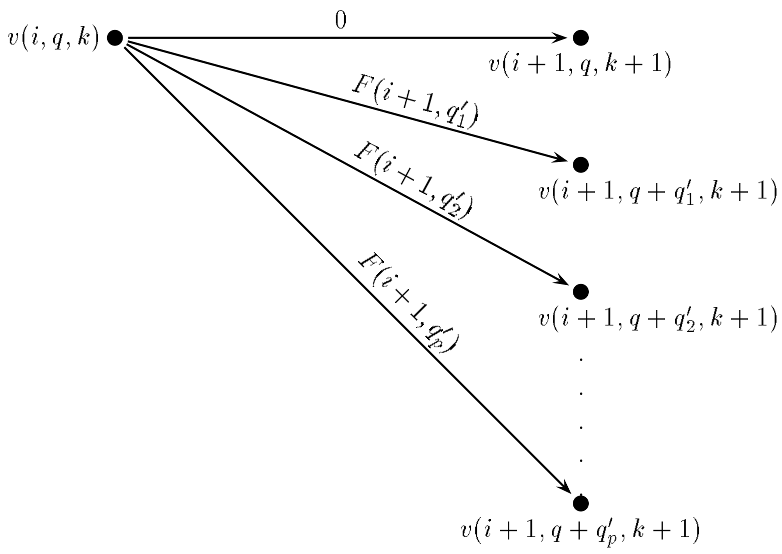


Figure 2.1: Possible outgoing arcs from a vertex $v(i,q,k)$ in the graph underlying the dynamic programming scheme. Note $q'_1 < q'_2 < \ldots < q'_p$.

If we are at an arbitrary node in the Branch-and-Bound tree we distinguish between two cases:

1. $P(\alpha) = \emptyset$ and
2. $P(\alpha) \neq \emptyset$.

If $P(\alpha) = \emptyset$ we just adjust the problem for the number of already generated routes $\hat{k}$, their load $\hat{q}$ and the set of customers already used in these routes

$\hat{I}$. The above described dynamic programming algorithm can then be used on this reduced problem.

In the case of $P(\alpha) \neq \emptyset$ exactly one of the routes in the lower bound is an extension of $P(\alpha)$. Now, let $\bar{F}(i, q)$ be the minimum length of such an extension ($\bar{F}(i, q)$ is calculated in the same way as $F(i, q)$). As before, the problem can be reduced according to $\hat{k}, \hat{q}$ and $\hat{I}$. The directed graph is now extended to contain vertices $v(i, q, k)$ and $\bar{v}(i, q, k)$ and the following arcs:

- Arcs of length 0 from $v(i, q, k)$ to $v(i + 1, q, k)$ and from $\bar{v}(i, q, k)$ to $\bar{v}(i + 1, q, k)$. These corresponds to not using the customer $i + 1$ in the routes.
- Arcs of length $F(i+1, q')$ from $v(i, q, k)$ to $v(i+1, q+q', k+1)$ and from $\bar{v}(i, q, k)$ to $\bar{v}(i + 1, q + q', k + 1)$ $\bar{v}(i + 1, q + q', k + 1)$ for each possible value of q'.

In [KRT87] problems up to 15 customers are solved by this method.

2.3.2 Lagrange Relaxation-based methods

The second method mentioned contains a number of papers using slightly different approaches. There is *variable splitting* followed by Lagrange relaxation [JMS86, FJM97, Mad88, Hal92], K-*tree* approach followed by Lagrange relaxation [FJM97] and finally Kohl et al. in [KM97] presented shortest path with side constraints approach followed by Lagrange relaxation.

In [KM97] Kohl et al. relaxes the constraints ensuring that every customer is served exactly once, that is

$$\sum_{k \in \mathcal{V}} \sum_{j \in \mathcal{N}} x_{ijk} = 1 \quad \forall i \in \mathcal{C}$$

is relaxed and the objective function with the added penalty term then becomes

$$\min \sum_{k \in \mathcal{V}} \sum_{i \in \mathcal{N}} \sum_{j \in \mathcal{N}} (c_{ij} - \lambda_j) x_{ijk} + \sum_{j \in \mathcal{C}} \lambda_j.$$

Here λ_j is the Lagrange multiplier associated with the constraint that ensures that customer j is serviced. The model now decomposes into one subproblem for each vehicle, but as the vehicle are assumed to be identical all the $|\mathcal{V}|$ subproblems are identical. The resulting subproblem is a shortest path problem with time window and capacity constraints and as the arc costs are modified by subtracting the relevant Lagrange multiplier the graph may even contain negative cycles. This shortest path problem is very difficult to solve, and a solution to the problem is discussed in chapter 4.

The master problem which consists of finding the optimal Lagrange multipliers, i.e. Lagrange multipliers that yields the best lower bound, is solved by a method using both sub-gradient optimization and a bundle method. Kohl et al. managed to solve problems of 100 customers from the Solomon test cases (see reference [Roc]), among them some previously unsolved problems.

In [FJM97] Fisher et al. presents an algorithm for solving the VRPTW optimally where the problem is formulated as a K-tree problem with degree $2K$ on the depot. A K-tree for a graph containing $n + 1$ vertices is a set of $n + K$ edges spanning the graph. Informally, the VRPTW could be described as finding a K-tree with degree $2K$ on the depot, degree 2 on the customers and subject to time and capacity constraints. A K-tree with degree $2K$ on the depot therefore becomes equal to K routes.

In [FJM97] the problem is defined as follows:

$$\min_{x\in X, y\in Y} \sum_{i,j\in\mathcal{N}, i\neq j} c_{ij}x_{ij} \quad \text{s.t.} \tag{2.10}$$

$$\sum_{i\in\mathcal{N}, i\neq j} x_{ij} = \begin{cases} 1 & \text{if } j \in \mathcal{C} \\ k & \text{else} \end{cases} \tag{2.11}$$

$$\sum_{j\in\mathcal{N}, j\neq i} x_{ij} = \begin{cases} 1 & \text{if } i \in \mathcal{C} \\ k & \text{else} \end{cases} \tag{2.12}$$

$$\sum_{i\in S}\sum_{j\in \hat{S}} x_{ij} \geq k(S) \qquad \forall S \subseteq \mathcal{C}, |S| \geq 2 \tag{2.13}$$

$$\sum_{i\in \hat{S}}\sum_{j\in S} x_{ij} \geq k(S) \qquad \forall S \subseteq \mathcal{C}, |S| \geq 2 \tag{2.14}$$

$$\sum_{h=i}^{m_p-1} x_{i_h,i_{h+1}} \leq m_p - 2 \qquad \forall p \in P \tag{2.15}$$

$$y_{ij} = x_{ij} + x_{ji} \qquad \forall i,j \in \mathcal{N} \tag{2.16}$$

$$x_{ij} \in \{0,1\} \tag{2.17}$$

$$y_{ij} \in \{0,1\} \tag{2.18}$$

where x_{ij} is set to 1 if a vehicle travels directly from customer i to customer j and 0 otherwise. y_{ij} is equal to the number of arcs joining customers i and j, that is $x_{ij} + x_{ji}$. $k(S)$ is a lower bound on the number of vehicles required to service the customers in S. Finally, X is all the x_{ij} variables and Y the set of y variables that defines a K-tree with degree $2K$ on the depot, that is

$$Y = \{y : y_{ij} \in \{0,1\} \text{ and } y \text{ defines a } K\text{-tree with } \sum_{i+1}^{n} y_{0i} = 2K\}.$$

$P = \langle i_1, i_2, \ldots, i_{m_p} \rangle$ is any time violating path, that is, there is no way to deliver the customers in the path given while satisfying the time windows.

All constraints except the constraints ensuring that at most one arc is joining customers i and j are then Lagrangian relaxed. The problem is then solved with a minimum degree-constrained K-tree problem as subproblem and the Lagrange multipliers are set using the sub-gradient approach (these ideas were already sketched in [Fis94a] as an extension of a optimal algorithm for the VRP but not implemented). Since there are exponentially many constraints in (2.13), (2.14) and (2.15) only a subset of these are generated and dualized. The constraints are generated as they are violated.

As mentioned, for given Lagrange multipliers the master subproblem is a minimum degree constraint K-tree problem. A polynomial algorithm for solving this problem is given in [Fis94b]. Here first a minimum spanning tree is constructed and then the K least cost unused arcs are added to the tree. If the depot is not incident to $2K$ arcs, the K-tree is modified by a series of arc exchanges until the depot has reached degree $2K$.

This algorithm (depicted in figure 2.2) was able to solve several of the clustered Solomon test case problems to optimality, but it was not able to solve any of the random Solomon test case problems exactly.

```
⟨ Initialize lagrangean multipliers ⟩
⟨ find a minimum spanning tree ⟩
⟨ add the k least cost unused arcs ⟩
⟨ exchange arcs until the depot has degree 2k ⟩
⟨ remove arcs incident to the depot ⟩
if ⟨ relaxed constraints are violated ⟩ then
    ⟨ update lagrangean multipliers ⟩
    if ⟨ more iterations ⟩
        goto 1
    else
        ⟨ Branch-and-Bound ⟩
        goto 1
else
⟨ optimal solution ⟩
```

Figure 2.2: The minimum degree constrained K-tree procedure.

Variable splitting (sometimes also referred to as *Lagrange decomposition*, or *Cost splitting* [NW88]) for the VRPTW was first presented in a technical report in [JMS86] by Jörnsten et al. in 1986 but no computational results were given here. In [Mad88] four different variable splitting approaches are mentioned but again none are implemented or tested. In the Ph.D. thesis [Hal92] of Halse three approaches are analysed and one of these is implemented (the fourth variable splitting approach outlined in [Mad88] seemed not be be competitive with the other three approaches at all).

In order to decompose the problem we introduce a decision variable y_{ik} defined as:

$$y_{ik} = \begin{cases} 1, & \text{if vehicle } k \text{ visits customer } i \\ 0, & \text{otherwise} \end{cases}$$

Now the constraints

$$y_{ik} = \sum_{j \in \mathcal{N}} x_{ijk} \qquad \forall i \in \mathcal{N}, k \in \mathcal{V} \tag{2.19}$$

are introduced to the problem. By expressing some of the constraints using x-variables in y-variables, different variable splitting approaches can be obtained.

First we rewrite (2.2) and (2.3) by using y_{ik} instead of x_{ijk} and thereby get:

$$\sum_{k \in \mathcal{V}} y_{ik} = 1 \qquad \forall i \in \mathcal{C} \tag{2.20}$$

$$\sum_{i \in \mathcal{C}} d_i y_{ik} \leq q \qquad \forall k \in \mathcal{V}. \tag{2.21}$$

Additionally we introduce

$$y_{ik} \in \{0, 1\}.$$

Note now that (2.19) is the only constraints coupling (2.20) and (2.21), and (2.4) to (2.9). If only the coupling constraints (2.19) are relaxed we get the objective function

$$\sum\sum(c_{ij} + \lambda_{ik})x_{ijk} - \sum\sum \lambda_{ik} y_{ik}. \tag{2.22}$$

The problem can now be split into two subproblems: One is based on (2.4) to (2.9):

$$\begin{array}{ll} \min & \sum\sum(c_{ij}+\lambda_{ik})x_{ijk} \\ \text{subject to} & \text{network constraints} \\ & \text{time constraints.} \end{array}$$

This problem is an Elementary Shortest Path Problem with Time Windows (ESPPTW), a problem that with respect to difficulty is closely related to the ESPPTWCC problem which is treated in chapter 4, and the other problem is based on (2.20) and (2.21):

$$\begin{array}{ll} \min & -\sum\sum\lambda_{ik}y_{ik} \\ \text{subject to} & \text{capacity constraints} \\ & \text{visiting customer constraints} \end{array}$$

which is a General Assignment Problem (GAP). The GAP is in itself a rather difficult combinatorial optimization problem to solve. Several methods exist and in [Mad90] a hybrid between two methods is used. This approach was implemented by Olsen [Ols88]. Instances containing up to 16 customers are solved.

Another way of decomposing is moving the capacity constraints from the second subproblem to the first. We then get:

$$\begin{array}{ll} \min & \sum\sum(c_{ij}+\lambda_{ik})x_{ijk} \\ \text{subject to} & \text{network constraints} \\ & \text{time constraints} \\ & \text{capacity constraints} \end{array}$$

which is an Elementary Shortest Path Problem with Time Windows *and* Capacity Constraints (ESPPTWCC), and the second subproblem becomes

$$\begin{array}{ll} \min & -\sum\sum\lambda_{ik}y_{ik} \\ \text{subject to} & \text{visiting customer constraints.} \end{array}$$

This problem is called a Semi Assignment Problem (SAP) and is basically a GAP without capacity constraints. The SAP can easily be solved by inspection. This approach was implemented by Halse in [Hal92] and some problems from the Solomon test cases (see reference [Roc]) with 100 customers were solved.

Finally one may duplicate the capacity constraints and place them in each of the two subproblems. This results in an ESPPTWCC and GAP. Even though this results in two time-consuming problems, one could hope for sharper bounds and thereby less work. In [Mad90], Madsen reports on not so promising results, although no figures are presented. This approach has not yet been implemented.

As all methods built on Lagrange relaxation use relaxation (eg. we solve a SPPTW or SPPTWCC instead of an ESPPTW or ESPPTWCC) a branch-and-bound framework has to be implemented as well. Kohl [Koh95] shows that if there exists feasible solutions to a VRPTW problem, the lower bound achieved by GAP and ESPPTWCC is not better than the one obtained by SAP and ESPPTWCC. Hence, solving the more difficult GAP instead of the SAP does not produce better bounds.

2.3.3 Other methods

The 3-index formulation of the VRPTW as presented in section 2.1 can be transformed into a 2-index formulation, if we do not care about which vehicle visits which the customers. This 2-index formulation is the basis of the algorithm presented in the just recently published Ph.D. Thesis by Kontoravdis [Kon97]. In contrast to the other approaches (dynamic programming, Lagrange relaxation and column generation), Kontoravdis works with the formulation as it is. The model is relaxed by removing the integrality constraints. Then a series of bounds are calculated in an effort to reduce the gap. The main part of the algorithm is a branch-and-cut-algorithm using well-known inequalities, but also a set of new inequalities – incompatible path inequalities and incompatible pair inequalities – are suggested. These new classes of inequalities can, however, only be used if one works directly with the formulation using x_{ij} variables. As we are going to use a different formulation, these inequalities can not be utilized in our algorithm and therefore further discussion is omitted.

Additionally, Kontoravdis works with an objective function not previously used when the problems are solved to optimality. His objective function is to minimize the number of vehicles used. The distance traveled is secondary and is only estimated by running a heuristic on the optimal solution found by the branch-and-bound-algorithm. Kontoravdis achieves some very good results on the standard problems, but this may have something to do with the objective function used.

2.4 Review of approximation algorithms and heuristics

The field of non-exact algorithms for the VRPTW problem has been very active – far more active than that of exact algorithms. A long series of papers has been published over the recent years.

In the field of approximation algorithms and heuristics one sometimes classifies an algorithm as sequential or parallel. In a sequential algorithm one route at a time is constructed, while a parallel algorithm may build more routes at the same time. This conflicts with the notion of a sequential algorithm (running on one processor) and a parallel algorithm (running on several processors) used later in the thesis. We will therefore avoid the use of this classification of heuristics.

Heuristic algorithms that build a set of routes from scratch are typically called *route-building* heuristics, while an algorithm that tries to produce an improved solution on the basis of an already available solution is denoted *route-improving*.

2.4.1 Route-building heuristics

The first paper on route-building heuristics for the VRPTW is [BS86]. Their algorithm is an extension of the legendary *Savings heuristic* of Clark and Wright for the VRP problem ([CW64]). The algorithm begins with all possible single-customer routes (*depot* - i - *depot*). In every iteration we calculate which two routes can be combined with the maximum saving, where the saving between customers i and j are calculated as:

$$sav_{ij} = d_{i0} + d_{0j} - Gd_{ij}. \tag{2.23}$$

Here G is sometimes referred to as the *route form factor*. In [BS86], a time-oriented nearest neighbour algorithm is developed by defining the savings as a combination of distance, time and "time until feasibility". A similar heuristic based on the savings algorithm is developed in [Sol87], but here the time aspect is not part of the savings function. Instead the arcs that can be used are limited by how large the waiting times get if they are used (if the waiting time would get larger than a threshold value W it is not valid). Due to the existence of time windows we have to take account of route orientation. Additionally we have to check for violation of the time windows when two routes are combined. These heuristics have time complexity of $O(n^2 \log n^2)$. Solomon reported reasonable results for this heuristic.

Also van Landeghem has presented a heuristic based on the Savings heuristic. His *bi-criteria heuristic* presented in [vL88] uses the time windows in (2.23) in order to get a measurement of how good a link between customers is in terms of timing.

Another heuristic described in the paper by Solomon is a *Time-Oriented, Nearest-Neighbour heuristic*. Every route in this heuristic is started by finding the unrouted customer that is closest to the depot. The "closeness relation" tries to take both geographical and temporal closeness of the customers into account. At every subsequent iteration the customer closest (again using the same metric as before) to the last customer added to the route is considered for insertion to the end of the route presently generated. When the search fails a new route is started.

The Solomon paper also describes three *Insertion heuristics*. Here the best feasible position for each unserviced customer i is computed using a function f_1. The best customer for insertion is then selected by another function f_2. If no insertion is possible a new route is started. In the Solomon paper three different sets of f_1 and f_2 functions are presented, each weighting different aspects of the routes (one of these function sets describes the I1 heuristic often used to generate the initial solution in improvement heuristics). The best of the three heuristics (I1) minimizes a weighted sum of detour (in time units) and delay to identify the best insertion place for each customer. The selection of the customer to be inserted is then based on a generalization of the Savings heuristic.

Finally a *Time-Oriented Sweep Heuristic* is presented. Here a popular decomposition into first a clustering phase, assigning customers to different clusters, and then a scheduling phase, building a route for each cluster is used. Building each route the becomes a TSPTW problem which can be solved using the TSPTW heuristics developed by Savelsbergh in [Sav56, Sav90].

Assigning customers to clusters is done by using a technique introduced in a paper by *Gillet and Miller* for the VRP. Here a "center of gravity" is computed and the clusters are partitioned according to their polar angle. Scheduling the customers, one of the previously developed tour-building heuristics are used to build a 1-route solution. Due to the time windows and/or capacity constraints some customers may now be unscheduled. In order to schedule these the scheduled customers are removed and the process is repeated.

The sweep heuristic typically performs better than the other heuristics in cases where many customers can be assigned to each route.

In general the heuristics of Solomon and van Landeghem return a solution fast. Their solution does however generally lack in quality. Most of the time the solutions are more than 10 percent from optimum.

A problem of building one route at a time is usually that the routes generated in the latter part of the process are of poor quality as the last unrouted customers tends to be scattered over the geographic area. In [PR93] Potvin and Rousseau tries to overcome this problem of the Insertion heuristics by building several routes simultaneously. The initialization of the routes is done by using the insertion heuristic of Solomon. On each route the customer farthest away from the depot is selected as a "seed customer". Then we proceed by computing the best feasible insertion place for each unserviced customer and insert the one with the largest difference between the best and the second best insertion place. This method is better than the Solomon heuristics but still the solutions are quite far away from optimum. Russell elaborates further on the insertion approach in [Rus95].

Antes and Derigs describes in [AD95] another approach build upon the classical insertion idea. Here every unrouted customer requests and receives from every route in the schedule a prize for insertion (set to infinity if insertion is not possible), defined in a similar way as in the Solomon heuristics. Then the unrouted customers send a proposal to the route with the best offer, and now each route accepts the best proposal *among those customers with the fewest number of alternatives*. Note that more customers can be inserted in each iteration. If a certain threshold of routes is violated a certain number of customers are removed and the process is initiated again. The results of Antes and Derigs are comparable to those presented in [PR93]. Generally building several routes in parallel results in better solutions than building the routes one by one.

Like the route-first scheduling-second principle mentioned above, Solomon in [Sol86] suggests doing it the other way around in the *Giant-Tour Heuristic*. First the customers are scheduled into one giant route and then this route is divided into a number of routes (the initial giant tour could for example be generated as a traveling salesman tour not considering capacity and time windows). No computational results are given in the paper for the heuristic.

The only implementation of a route-building heuristic on parallel hardware is reported in [FP93] where an insertion heuristic that simultaneously builds the routes is described using an (unnamed) Solomon heuristic to generate the initial seed customers.

2.4.2 Route-improving heuristics

The basis of almost every route-improving heuristic is the notion of a neighbourhood. The neighbourhood of a solution S is a set $N(S)$ of solutions that can be generated with a single "modification" of S.

Checking some or all of the solutions in a neighbourhood might reveal solutions that are better with respect to objective function. This idea can be repeated from the better solution. At some point no better solution can be found and an optimum has been reached. It is definitely a local optimum but it might even be global. This algorithm is called *local search*. Metaheuristics are typically based on local search but with methods added for escaping an optimum in order to check other parts of the search space for even better solutions.

First, neighbourhood structures used in various VRPTW heuristics will be introduced, and then the algorithms in which they are used are described.

Neighbourhoods for the VRPTW

One of the most used improvement heuristics in routing and scheduling is the *r-Opt* heuristic. Here r arcs are removed and replaced by r other arcs. A solution obtained using a r-Opt neighbourhood that cannot be improved further is called *r-optimal*. Usually r is at most 3. Using the *3-Opt* on the routes of a solution to the VRPTW problem is not without problems. For all possible *2-Opt* interchanges and some of the exchanges in the 3-Opt neighbourhood, parts of the route is reversed. This may very likely lead to a violation of the time windows.

In [PR95], Potvin and Rosseau presents two variants 2-Opt* and Or-Opt that maintain the direction of the route.

In Or-Opt a segment of the route, that is, l consecutive customers, are moved to another place on the route. An example of the Or-Opt is shown in figure 2.3. It is quite easy to see that Or-Opt's are a subset of 3-Opt's as we exchange 3 special arcs with 3 others. The size of the neighbourhood is although reduced from $O(n^3)$ to $O(n^2)$. Generally the size of a r-Opt neighbourhood is $O(n^r)$. The 2-Opt* is exchanging one segment of one route with a segment of another route. This is illustrated in figure 2.4. Again the size of the neighbourhood is $O(n^2)$. This neighbourhood operator is sometimes denoted crossover or simply cross.

The last part of either route will become the last of the other. Note that if (i, i_s) is the first arc on one of the routes and (j, j_s) is the last one on the other route the two routes will be merged into one. Note also that the crossover contains Or-opt as a special case.

The *relocate* operator moves a customer from one route to another as shown in figure 2.5. Here the edges $(i_p, i), (i, i_s)$ and (j, j_s) are replaced by $(i_p, i_s), (j, i)$ and (i, j_s).

The *exchange* operator swaps customers in different routes, thereby interchanging two customers simultaneously into the other routes (see fig-

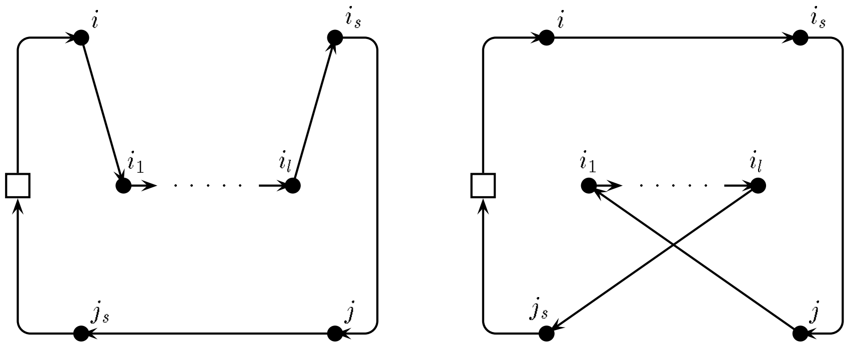


Figure 2.3: An example of an Or-Opt exchange. The figure on the left presents the route before the Or-Opt is performed, and the figure on the right is the route after the exchange. Note that orientation of the customers from i_1 to i_l remains the same. The square represents the depot.

ure 2.6). This idea may be extended to swap segments of routes between two routes.

The k-node interchange by Christofides and Beasley is modified by several authors to take time windows into account. Sequentially each customer i is considered and the sets M_1 and M_2 are identified. M_1 is defined as the customer i and its successor j. Then the elements of M_2 are found as the two customers closest to i and j but not on the same route as i and j (found by minimizing insertion cost calculated by the Euclidean distance). A neighbourhood is then defined by removing the elements of M_1 and M_2 and inserting them in any other possible way. As this neighbourhood is quite large, only the k most promising candidates are checked.

Another neighbourhood is the λ-interchange by Osman originally developed for the VRP. It is a generalization of the relocate operator. Here a subset of customers of size $\leq \lambda$ in one route is exchanged with a subset of size $\leq \lambda$ from another route. Typically there is also given an ordering in which the different set sizes are tested. For example using a 2-interchange scheme we would first try to move one element from one route to the other, and none

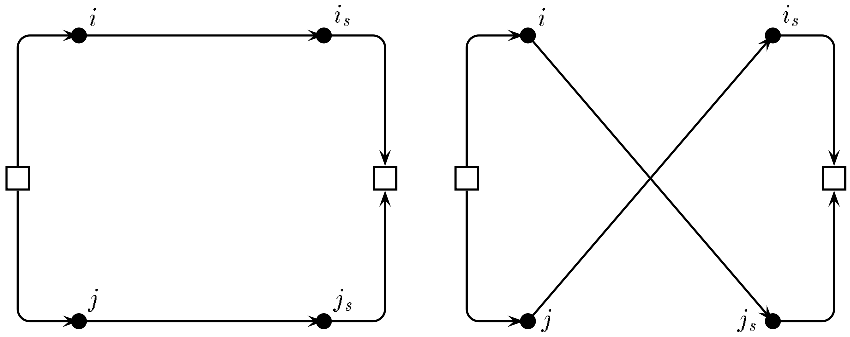


Figure 2.4: An example of an 2-Opt* exchange. The squares represent the depot. The left picture is before and the right is after the exchange. Note that there may be any number of customers on the path between the depot and i respectively j and again between i_s respectively j_s and the depot.

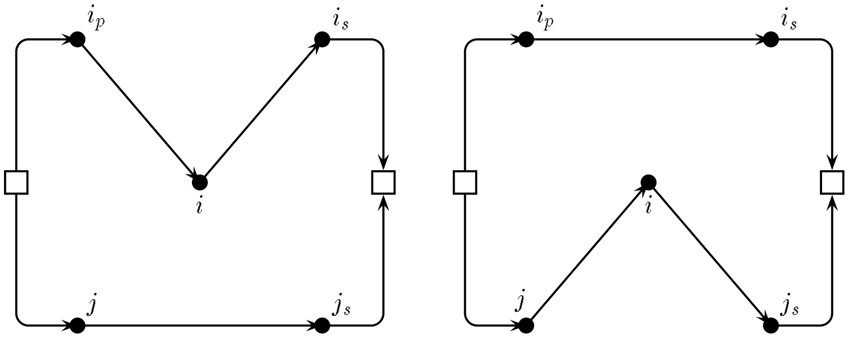


Figure 2.5: An example of an relocate operator. The square represents the depot. Again the left picture is before and the right is after the relocation. Note that there may be any number of customers on the path between the depot and i respectively j and again between i_s respectively j_s and the depot.

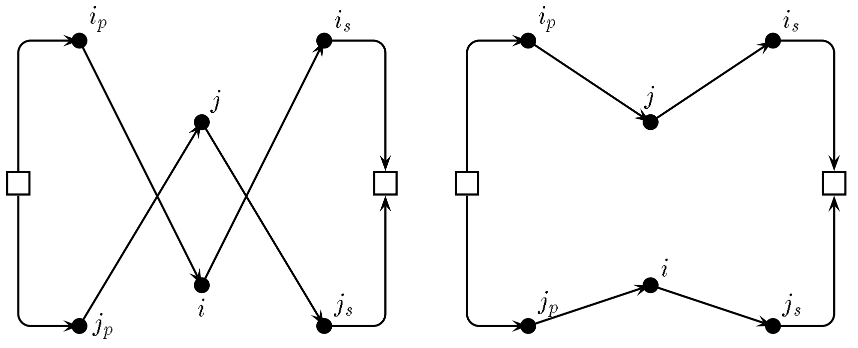


Figure 2.6: An example of an exchange operator. The picture on the left is before the relocate and the picture on the right is after the relocate is performed.

the other way. Then we would try the reverse situation, and then try to exchange one element from one route with one from the other etc. This would be written as $(1,0),(0,1),(1,1),(0,2),(2,0),(2,1),(1,2),(2,2)$.

Finally, a neighbourhood denoted *shift-sequence* is used by Schulze and Fahle in [SF99]. A customer is moved from one route to another then checking all possible insertion positions. If an insertion is made feasible by removing another customer j, it is removed and inserted in another route. This procedure is repeated until feasibility is restored. A summary of the neighbourhoods including am informal description is presented in table 2.1.

In the papers [Rus95] and [PR95] local search heuristics are developed. Whereas Russell in [Rus95] use the modified k-move interchange to improve the solution generated by his route-building heuristic described in the same paper, Potvin and Rousseau in the other paper introduces the 2-Opt* and uses it in conjunction with the Or-Opt neighbourhood.

Simulated Annealing

Simulated Annealing was one of the first metaheuristics developed. When using simulated annealing one does not search for the best solution in the

Operator	**Description**	**Reference**
Or-Opt	A continuous segment of customers is moved from one position on a route to another	[PR95]
2-Opt* *(crossover, cross)*	Changing one segment of a route with another segment from another route	[PR95]
Relocate	Move one customer from one route to another	[KPS97]
Exchange	Interchange two customers between two routes	[KPS97]
k-node interchange	Sequentially each customer i is considered. Customer i and its successor j and the two customers closest to i and j but not on the same route are removed. The neighbourhood is defined by trying to insert these four vertices in any other possible way. As this neighbourhood is quite large, only the k most promising candidates are checked.	[Rus95]
λ-interchange	A subset of customers of size $\leq \lambda$ is exchanged with a subset of customers of size $\leq \lambda$ from another route	[TOS94]
Shift-sequence	A customer is moved from one route to another checking all possible insertion positions. If an insertion is feasible by removing another customer j, it is removed and inserted in another route. This procedure is repeated until feasibility is restored.	[SF99]

Table 2.1: Summary of neighbourhood-operators.

neighbourhood of the current solution. Instead one simply draws at random a solution from the neighbourhood. If the solution is better it is always accepted as a new current solution, but if the solution is worse than the present current solution it is only accepted with a certain probability. The acceptance probability is determined by a "temperature" which is gradually decreased. By reducing the temperature the selection becomes more and more selective in accepting new solutions. The idea of simulated annealing comes from thermodynamics and metallurgy: when a metal in fusion is cooled slowly enough it tends to solidify in a structure of minimal energy.

Chiang and Russell develop in [CR96] three different simulating annealing methods: one using a modified version of the k-node interchange mechanism and the second one using the λ-interchange mechanism as proposed by Osman with λ set to 1.

The third algorithm borrows the concept of a tabu list (this algorithm reflects a recent trend within the metaheuristic community to combine features from different metaheuristic paradigms into *hybrid heuristics*) from the Tabu Search metaheuristic (explained later). Using simulated annealing with the λ-interchange mechanism the tabu list contains moves that will not be allowed for the time being.

The last two methods have faster convergence than the first, although the first yields slightly better results. The travel distances obtained by the three simulated annealing algorithms was between 7 and 11.5 percent from optimum.

In [TOS94] a non-monotone probability function is used. Thangiah et al. are using the λ-interchange with $\lambda = 2$ scheme to define the neighbourhood. The temperature is decreased after every iteration. In case the entire neighbourhood has been explored without finding any accepting moves the temperature is increased. This is called a "reset". The temperature is increased to the maximum of the temperature at which the best solution was found and half of the temperature at the last reset. After R resets without improving the best solution the algorithm terminates.

The quality of the solutions obtain in [TOS94] is about the same as those obtained by Chiang and Russell in [CR96].

Tabu Search

Just as simulated annealing, the *Tabu Search* heuristic is one of the "old" metaheuristics. It was introduced by Glover in two papers from 1989 and 1990 (for a tutorial see [Lau94a, HTd95]). At each iteration the neighbourhood of the current solution is explored and the best solution in the neighbourhood is selected as the new current solution. In order to allow the algorithm to "escape" from a local optimum the current solution is set to the best solution in the neighbourhood even if this solution is worse than the current solution. To prevent cycling visiting recently selected solutions is forbidden. This is implemented using a "tabu list". Often, the tabu list does not contain "illegal" solutions, but forbidden moves. It makes sense to allow the tabu list to be overruled if this leads to an improvement of the current overall best solution. Criteria such as this for overruling the tabu list are called *aspiration criteria*. The most used criteria for stopping a tabu search are a constant number of iterations without any improvement of the over-all best solution or a constant number of iteration in all.

In the paper [GPR94] by Garcia et al., two improvement heuristics *2-opt** and *Or-opt* are used to explore the neighbourhood. In order to restrict the amount of work, not all possible 2-opt* or Or-opt operation are carried out. The exploration of the neighbourhood is restricted to the exchange of arcs that are close in distance.

Solomon's I1 heuristic (one of the insertion heuristics developed in [Sol87]) is used to generate the initial solution. The algorithm shifts between the two strategies. When one has not made any improvement for a certain number of iterations the other strategy is used instead and vice versa. In order to minimize the number of routes the algorithm tries to move customers from routes with few customers to other routes. This is done using Or-opt.

The technical report [TOS94] describes a tabu search which used λ-interchanges. In addition the Tabu Search is combined with the Simulated Annealing algorithm (described previously) using the parameters of the Simulated Annealing to accept or reject solutions worse than the current solution.

In [BGG$^+$95], Badeau et al. first generate a series of solutions and then new solutions are composed by randomly selecting from the already generated routes. The selection is done biased to the good routes. When one route is selected the remaining routes servicing customers from this route is excluded. This process is continued until all customers are serviced by a route, or the algorithm runs out of routes. In the later case the remaining customers are added to the solution by the use of Solomons I1 heuristic.

The solution is now decomposed into groups of routes (the groups are generated by the use of polar angle and center of gravity). For each group a Tabu Search is performed using the exchange operator on segments. Furthermore segments of costumers are moved around within each route. In order to force the algorithm to make a thorough exploration of the search space, frequently performed crossovers are penalized.

Another tabu search algorithm with similarities to [GPR94] is developed by Potvin et al. in [PKGR96]. It is based upon the local search methods discussed in [PR95].

In [SF99] a feasible solution is first found by using the Solomon I1 heuristic. The neighbourhood is then defined by the shift sequence operator. The shift sequence with the highest gain is chosen. Furthermore *route elimination* is used on routes with few customers (trying to move the customers to other routes). Additionally the Or-opt exchange is used on every modified route.

Several papers have been written about parallelization of the Tabu Search heuristic (see [GPR94, BGG$^+$95, SF99]). Their efforts can be divided into three groups:

1. Partitioning of the neighbourhood.
2. Run parallel threads of Tabu Searches.
3. Decompose the problem into subproblems each solved by parallel Tabu Searches.

In [GPR94] the strategy 1 is used to parallelize the Tabu Search. One processor (called the "master") controls the Tabu Search, while the remaining processors (called "slaves") are used to explore the neighbourhood, which is partitioned among them. After exploration the best move from each processor is send to the master. Now exchanges independent of each other can be applied simultaneously to the current routes. The exchanges to be used are determined by a simple greedy heuristic. Note that the master waits for all slaves to deliver their exchanges, thereby making the algorithm highly synchronous.

Another parallel Tabu Search algorithm is developed in [SF99] where the sequential algorithm described above is parallelized. In [GPR94] the neighbourhood is partitioned on all but one processor. But instead of waiting for all slave processors to finish exploring the neighbourhood a threshold decides when enough shift sequences have been supplied by the slave processors for the master to start processing. The results from the remaining slaves are simply ignored.

The tabu search heuristic for the VRPTW reported in the technical report [BGG+95] by Badeau et al. is essentially a parallelization of the Tabu Search presented in [TBG+95]. Here a combination of strategy 2 and 3 is used. After the generation of a solution at the master processor, the solution is decomposed into groups of routes. At each slave processor an "independent" Tabu Search tries to improve the overall solution by improving the solution for the routes it received from the master processor.

Among the parallel implementations the algorithm developed in [SF99] performs slightly worse that the other parallel algorithms when working in problems with tight windows. As time windows get larger all algorithms perform equal with respect to the quality of the solution. As different parallel computers or network of computers are used it is difficult to compare the algorithms with respect to running time.

The *Reactive Tabu Search* was developed by the Battiti and Tecchiolli in order to strengthen the basic tabu search concept. In reactive tabu search the length of the tabu list is not fixed but can dynamically be varied during the search procedure in order to avoid limit cycles where we repeatedly visit the same sequence of solutions.

In [CR97] the reactive tabu search metaheuristic is applied to the parallel construction approach of [Rus95]. The tabu search route improvement procedure is invoked each time another 10 percent of the customers have been added to the emerging routes using the λ-interchange of Osman as the neighbourhood. If the same solution defined by

- number of vehicles,
- total accumulated distance, and
- total accumulated travel time

occurs to often (more than three times) the tabu list is increased by a constant factor. If waiting time is eliminated, the customer is fixed at its position for a number of iterations, and as in [BGG+95] too frequent switches of customers is penalized. On the other hand if no feasible solution is found by the tabu search, the size of the tabu list is decreased by (possibly different) constant factor. The paper presents the solution for two well-known standard test instances and report on a gap of around 3 percent. Generally it is among the best heuristics for the VRPTW. One of the conclusions in the paper is that diversification/intensification (the extension and contraction of the tabu list) is just as important in obtaining good solutions as variable length tabu list

The Genetic Algorithm

Genetic Algorithms is an iterative procedure that maintains a *population* of κ candidates (solutions). The population members can be seen as entities of artificial chromosomes (of fixed length with binary values). Each chromosome has a *fitness value* describing the "goodness" of the solution. Variation into the population is introduced by *cross-over* and *mutation*. Cross-over is the most important operator. Here some chromosomes undergo a two point cross-over and produce offspring for the next generation. *Mutation* prevents loss of important information by randomly mutating (inverting) bits in the chromosomes. The termination criterion is usually a certain number of iterations.

The genetic algorithm of Potvin and Bengio, presented in [PB96], operates on chromosomes of feasible solutions. The selection of parent solutions is stochastic and biased toward the best solutions. Two types of cross-over are used. Both only rarely produce valid solutions and the solutions therefore has to undergo a "repair phase", as the algorithm only works with feasible solutions. The reduction of routes is often obtained by the two mutation operators. The routes are optimized by an Or-opt-based local search every k iterations.

A genetic algorithm for the VRPTW is presented in [TOS94]. This algorithm uses the cluster-first route-second method mentioned earlier. Clustering is done by a Genetic Algorithm while routing is done by an insertion heuristic. The Genetic Algorithm works by dividing the chromosome into K divisions of B bits. The algorithm is based on dividing the plane by using the depot as origo and assigning the polar angle to each customer. Each of the divisions of the chromosome then represent the offset of the seeds of a sector. The seeds are polar angles that bound the sector and thereby determine the members of the sector. Further specialization of the algorithm is presented in [TOS], where the insertion heuristic is replaced by a local search phase using the λ-interchange with λ equal to 2, simulated annealing or tabu search.

Competitive Neural Networks

In two papers [PR99, PDR96] the insertion heuristic of Potvin and Rousseau (see page 30) is extended. Instead of selecting the customers farthest away as seed customers, the seed customers are selected by a *competitive neural network* and a genetic algorithm.

In [PR99] a special type of neural network called competitive neural network is used to select the seed customers. Competitive neural network is frequently used to cluster or classify data. For every vehicle we have a *weight vector*. Initially all weight vectors are placed randomly close to the depot. Then we select one customer at a time. For each cluster we calculate the distance to all weight vectors. The closest weight vector is updated by moving it closer to the customer. This process is repeated for all customers a number of times, each time the process is restarted the update of the weight vector becomes less sensitive. At the end of this phase the seed customers are selected as the customers closest to the weight vectors. Finally the insertion heuristic by Potvin and Rousseau is used to construct a solution. This algorithm is about 25% slower than the insertion heuristic of Potvin and Rousseau but the solutions generated are between 10 and 25% better.

The insertion heuristic is extended further is [PDR96]. In the insertion heuristic two constants determine the importance of travel cost and travel time in the score of each unrouted customer. Furthermore the "route form factor" of Clark and Wright (see page 28) is used. Which value to assign the constants is to some extend not determined by the problem. In [PDR96] a genetic algorithm is used to find values for the three constants that give the best results. For each class of problems from the Solomon test-set the genetic algorithm is run for about 15 minutes. The idea is that for instances with the same characteristics only one run of the genetic algorithm is needed as the settings obtained can be reused. The results are better compared to the using the insertion heuristic without the preprocessing but only fractionally (here the running time of the genetic algorithm is not included in the overall running time of the algorithm as the settings from the genetic algorithm can be reused on many instances). The seed customers are selected using the approach from [PR99].

It seems strange that the authors of [PR99] do not take time windows into account when they determine the seed customers. In VRPTW the time windows often result in routes that are not confined to one geographical region of the plane, therefore clustering should take time windows into account. In [PDR96] a discussion of how to identify two instances as being equal lacks. An important point in using the genetic algorithm is that the running time of 15 minutes is justified by the idea of reusing the obtained values, therefore a discussion of how two instances are "equal" is needed.

Miscellaneous metaheuristics

Another general heuristic paradigm used for the VRPTW is the GRASP approach described in [KB95]. GRASP stands for *Greedy Randomized Adaptive Search Procedure* and is a combination of greedy heuristics, randomization, and local search. First an initial pool of feasible solutions is constructed. Then at each iteration all feasible moves are ranked according to an adaptive greedy function. One of the k best solutions is chosen at random and a local search is applied. This process is repeated a certain number of times. The GRASP algorithm of Kontoravdis et al. constructs feasible solutions by an randomized version of the insertion algorithm presented in [PR93]: instead of selecting the best customer for insertion every time, one of the k best candidates is chosen. After generating five solutions this way the best is chosen for local search. In the local search part each route ρ is considered for elimination by moving all its customers into other routes, or if this is not possible, replacing a customer with another which is then moved to another route.

The *Guided Local Search* (GLS) approach has been used for the VRPTW in the paper [KPS97] by Kilby et al. GLS is a memory-based metaheuristic like tabu search. In GLS the cost function is extended by adding penalty term encouraging diversification, that is, escaping from local minima is done by penalizing particular solution features. The local search phase of Kilby et al. uses the 2-Opt on single routes, the relocate, the exchange and 2-Opt* neighbourhoods. In their heuristic, using the same arc over and over again is penalized. GLS performs quite well compared to two of its competitors. GLS is also used in [The97] from the GreenTrip[1] consortium, an EU-funded project focused on the development of flexible and effective routing software.

A quite different approach from the metaheuristics, which all use some way of escaping local minima, is the *Large Neighbourhood Seach* put foreward by Shaw in [Sha97]. This heuristic is a greedy heuristic and it does not try to escape a possible local minimum. Instead, the neighbourhood structure is made considerably larger in order to enhance the chances of delivering a high-quality solution. The initial solution is made up of routes supplying just a single customer.

Finally Kanstrup Kristensen uses a hybrid in his master thesis [Kri95]. Here the exact shortest path algorithm of the Branch-and-Price approach (to be explained in the next chapter) is replaced by a tabu search, thereby mixing exact and heuristic approaches. The results are generally acceptable in quality as most solutions only diverge less than 2-3 per cent from the best known solutions. The author does admits, though, that the approach is both time and memory consuming. In [CJR81] Cullen et al. also uses the set partitioning approach for a heuristic. An interactive heuristic is described, where the branching decision is guided by the user that selects the arcs to be used to branch on.

[1] Homepage: www.cs.strath.ac.uk/~ps/GreenTrip/

2.5 Overview of exact methods and heuristics

Since 1987 numerous researchers have been using the Solomon test-sets when they test their heuristics or exact algorithms. Unfortunately, there is not a general consensus on two important aspects: objective and calculation of distance (and travel times) between vertices (customers and the depot). The test-sets are described in more detail in section 7.1.

As can be seen from the tables 2.2, 2.3 and 2.4 several different objectives have been proposed. For the entries marked with (?) in the column "Distance and time" in the tables 2.2 and 2.3 it is assumed that the calculation methods are based upon the methods used in the papers describing the algorithm(s) used as reference algorithms as no indication of the calculation method is given otherwise. When solving the problems to optimality, the most widely used approach is to minimize the total traveled distance, while the heuristics in almost every case have several objectives, the primary usually being minimization of the number of vehicles used.

As the customers are given by a pair (x, y) in the plane the (Euclidean) distances have to be calculated. In most papers, real arithmetic is used, that is, the results are neither rounded or truncated, but simply used as they are. As noted in [AD95, PR93] this makes the solution and running times dependent on the chosen hardware and the chosen precision. Some authors round or truncate to one or three decimals. Table 2.2, 2.3 and 2.4 present the choices taken regarding objective and calculation of distance and travel times in papers known to the author, and where the developed heuristic or exact algorithm for the VRPTW is tested using the Solomon test-sets. There does exist some papers on heuristics, which do not use the Solomon test-sets. Among those are [vL88]. These papers will not commented upon here.

The difference in objective and calculation of time and cost makes it hard to make direct comparisons between the exact methods and the heuristics.

---

[1] The routing cost is defined as the sum of travel time, service time and waiting time.

[2] A remark concerning the problems of using real arithmetic suggests that something else may have been used, but the choice is not stated in the paper.

[3] This paper contains two algorithms.

Reference	*Objective*	*Distance and time*
[Sol87]	1. Min number of vehicles 2. Min schedule time 3. Min travel distance 4. Min waiting time	Real arithmetic
[PR93]	1. Min number of vehicles 2. Min route time	Real arithmetic
[FP93]	1. Min number of vehicles 2. Min travel distance	Real arithmetic (?)
[GPR94]	1. Min number of vehicles 2. Min routing cost[1]	Real arithmetic (?)
[TOS94]	1. Min number of vehicles 2. Min travel distance	Real arithmetic (?)
[PR99]	1. Min number of vehicles 2. Min route time	Real arithmetic
[KB95]	1. Min number of vehicles 2. Min travel distance	Real arithmetic
[Rus95]	1. Min number of vehicles 2. Min schedule time 3. Min travel distance	Real arithmetic
[AD95]	1. Min number of vehicles 2. Min routing cost[1]	?[2]
[BGG+95]	Min travel distance	Multiplied by 1000 and rounded to nearest integer
[RT95]	1. Min number of vehicles 2. Min travel distance	Real arithmetic
[SF96]	1. Min number of vehicles 2. Min travel distance	Real arithmetic
[PKGR96]	1. Min number of vehicles 2. Min route time	Real arithmetic
[PB96]	1. Min number of vehicles 2. Min route time	Real arithmetic
[PDR96]	1. Min number of vehicles 2. Min route time	Real arithmetic

Continued on the next page.

Table 2.2: Overview of strategies chosen for objective and calculation of distance for heuristic methods. (Part 1)

Continued from the previous page.		
Reference	*Objective*	*Distance and time*
[CR96]	1. Min number of vehicles 2. Min schedule time 3. Min travel distance	Real arithmetic
[Sha97]	1. Min number of vehicles 2. Min travel distance	Real arithmetic
[KPS97]	Min travel distance	Real arithmetic (?)
[The97]	Min travel distance	Real arithmetic (?)
[GTA99]	1. Min number of vehicles 2. Min route time	Real arithmetic (?)
[Tha]	1. Min number of vehicles 2. Min routing cost[1]	Real arithmetic (?)
[TOS]	1. Min number of vehicles 2. Min travel distance	Real arithmetic (?)

Table 2.3: Overview of strategies chosen for objective and calculation of distance for heuristic methods. (Part 2)

Reference	*Objective*	*Distance and time*
[DDS92]	Min travel distance	One decimal point and truncation (travel times no decimal point and truncation)
[Hal92]	Min travel distance	One decimal point and rounding
[FJM94][3]	Min travel distance	Real arithmetic
		One decimal point and rounding
[Koh95]	Min travel distance	One decimal point and truncation
[GDDS95]	Min travel distance	One decimal point and truncation
[KM97]	Min travel distance	One decimal point and truncation
[Kon97]	Min number of vehicles	One decimal point and truncation

Table 2.4: Overview of strategies chosen for objective and calculation of distance for exact methods.

The VRPTW community is using the Solomon test-sets extensively, therefore we need to find a common objective function and a common method for calculating time and cost.

Time and cost should be calculated to with some rounded or truncated precision and it seems that the method first used in [Hal92] is becoming a standard for exact methods. The wide spread use of real arithmetic without rounding or truncation in the heuristics must stop. Instead the method of [Hal92] should also be used here, making comparison between the algorithms easier.

The objective function should be to minimize the total travel distance (cost). This objective function can also incorporate the minimization of the number of vehicles by assigning the cost of using a vehicle to the $(0, i)$ arcs.

A number of heuristics seem to perform equally well. Using the idea of minimum-escaping as performed by the metaheuristics lead to significantly higher quality of the solution. The better solutions do not come for free. The running time of the metaheuristics are significantly higher than the route-contruction or route-building heuristics. A number of methods generate solutions of almost equal quality. As the algorithms are tested on different computers an exact comparison is very difficult. The methods generating the best quality solutions (around 5% from optimum) are the GRASP approach [KB95], Tabu Search-based approach [BGG+95, RT95, CR97] and the method developed by the GreenTrip project [KPS97].

Among the exact methods it is difficult to compare [Kon97] with the others as a different objective is used. Otherwise column-generation based methods seems to be best right now, closely followed by methods based on Lagrange relaxation. The dynamic programming based methods can not compete with the column-generation and Lagrange relaxation.

Chapter 3

The sequential algorithm for the VRPTW

In this chapter we describe the sequential VRPTW algorithm, which will be used as the basis for the parallel algorithm developed later. In the following, it is assumed that all vehicles have the same capacity.

The column generation scheme presented in this chapter is first used in 1989 in [AMS89] for the VRP problem, while Desrosiers, Soumis and Desrochers in [DSD84] and again in 1985 Desrosiers, Soumis, Desrochers and Sauvé in [DSDS85] used the column generation approach to solve the m-TSP with time windows (or VRPTW without capacity constraints depending on taste). In [DDS92], the column generation approach is used for the first time for solving the VRPTW, and in [Koh95] a more effective version of the same model with addition of valid inequalities solves more instances to optimality than in [DDS92].

3.1 A set partitioning model of the VRPTW

Like many routing problems, the VRPTW can be described as a *set partitioning problem* (SP) (see e.g. [Sal75]). In this model each column corresponds to a feasible route, while the rows in the constraint matrix corresponds to customers. For each column, the variable x_r is defined by

$$x_r = \begin{cases} 1, & \text{if route } r \text{ is used in the solution} \\ 0, & \text{otherwise} \end{cases}$$

and c_r denotes the cost (distance) of route r. The resulting model becomes:

$$\min \sum_{r \in \mathcal{R}} c_r x_r \quad \text{s.t.} \tag{3.1}$$

$$\sum_{r \in \mathcal{R}} \delta_{ir} x_r = 1 \qquad \forall i \in \mathcal{C} \tag{3.2}$$

$$x_r \in \{0, 1\} \tag{3.3}$$

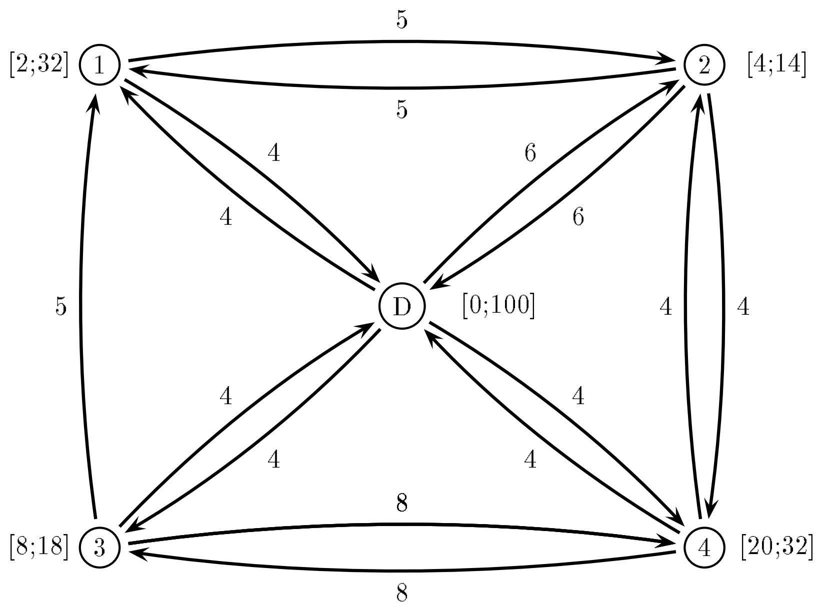
where $\mathcal{R}$ is the set of **all** feasible routes, and δ_{ir} is 1 if customer i is serviced by route r and 0 otherwise. Figure 3.1 shows an instance and the corresponding set partitioning model of the instance is shown in figure 3.2.

Note that the constraints concerning time windows, capacity and flow are assumed to be respected as $\mathcal{R}$ only contains feasible routes. This makes the model very versatile as other demands are hidden from the set partitioning formulation by the "route generator". Note also that a solution to the set partitioning formulation does not completely state a solution to the VRPTW as the order of service of the customers in each route is not given. This information has to be supplied separately, that is, two representations of the routes, one for the set partitioning problem, where we just indicate whether a customer is serviced on a given route or not, and one where the actual lay-out of the route is given, have to be maintained.

3.2 Preprocessing

In preprocessing the formulation is tightened before the actual optimization is started. This can be done by fixing some variables, reducing the interval of values a variable can take etc. The aim is to narrow the solution space.

In VRPTW the time windows can be reduced if as in our case the triangular inequality holds. Kontoravdis and Bard uses the triangular inequality



Feasible routes	
D – 1 – D (x_1)	D – 2 – 1 – D (x_5)
D – 2 – D (x_2)	D – 2 – 4 – D (x_6)
D – 3 – D (x_3)	D – 3 – 4 – D (x_7)
D – 4 – D (x_4)	

Figure 3.1: Instance graph of a routing problem. The time windows for each customer is shown beside the vertices. Here $c_{ij} = t_{ij}$ and service time for all customers is set to 10 time units. Demands are all 1 and capacities of the vehicles are set to 3.

$$
\begin{array}{lccccccccccccc}
\min & 8x_1 & +12x_2 & +8x_3 & +8x_4 & +15x_5 & +14x_6 & +16x_7 & \\
 & x_1 & & & & + \; x_5 & & & = 1 \\
 & & x_2 & & & + \; x_5 & + \; x_6 & & = 1 \\
 & & & x_3 & & & & + \; x_7 & = 1 \\
 & & & & x_4 & & + \; x_6 & + \; x_7 & = 1
\end{array}
$$

Figure 3.2: The set partition formulation of figure 3.1.

in [KB95] to strengthen the time windows. As the triangular inequality holds the earliest time a vehicle can arrive at a customer is by arriving straight from the depot and the latest time is by driving the fastest way to the depot, that is, straight back to the depot. So for a customer i the time window can be strengthened from $[a_i, b_i]$ to $[\max\{a_0 + t_{0i}, a_i\}, \min\{b_{n+1} - t_{i,n+1}, b_i\}]$.

A further reduction of the time windows can be achieved by the scheme developed by Desrochers et. al in [DDS92]. The time windows are reduced by applying the following four rules in a cyclic manner. The process is stopped when one whole cycle is performed without changing any of the time windows. The four rules are:

1. Minimal arrival time from predecessors:
   $a_l = \max\{a_l, \min\{b_l, \min_{(i,l)}\{a_i + t_{il}\}\}\}$
2. Minimal arrival time to successors:
   $a_l = \max\{a_l, \min\{b_l, \min_{(l,j)}\{a_j - t_{lj}\}\}\}$
3. Maximal departure time from predecessors:
   $b_l = \min\{b_l, \max\{a_l, \max_{(i,l)}\{b_i + t_{il}\}\}\}$
4. Maximal departure time to successors:
   $b_l = \min\{b_l, \max\{a_l, \max_{(l,j)}\{b_j - t_{lj}\}\}\}$

The first rule adjusts the start of the time window to the earliest time a vehicle can arrive coming straight from any other possible predecessor. The rule is illustrated in figure 3.3.

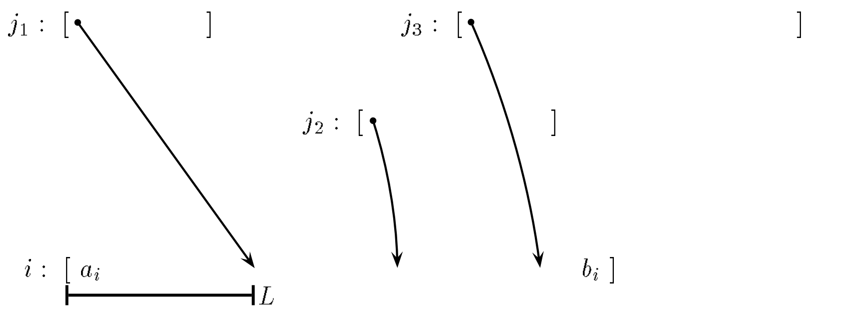


Figure 3.3: The start of the time window for customer i can be moved from a_i to L as this is the earliest possible arrival, when we start as early as possible from any of the 3 possible predecessor (j_1, j_2 and j_3).

In a similar fashion the second rule adjusts the start of the time window in order to minimize the excess time spend before the time windows of all possible successors opens if the vehicle continues to a successor as quickly as possible.

The two remaining rules uses the same principles to adjust the closing of the time window.

3.3 Branch-and-Price

For the small instance in the example given in figure 3.1 it was possible to generate all feasible routes (columns) and then solve the set partitioning problem. This becomes if not impossible then very cumbersome for even small instances. To illustrate this, consider an instance with 40 customers where the constraints make every route of up to 7 customers possible. As there exists $\binom{40}{i}$ routes of i customers there will be in order of

$$\sum_{i=1}^{6} \binom{40}{i} \approx 4.6 \text{ million}$$

routes that has to be generated.

The number of feasible routes quickly becomes very large, and in addition the set partitioning problem is $\mathcal{NP}$-hard. This makes the problem difficult to handle both with respect to memory and running time. This can be overcome by relaxing the integrality constraints. Then our integer program becomes a linear program, that is solvable by available software.

3.3.1 Column Generation

If a linear program contains too many variables to be solved explicitly, then we can initialize the linear program with a small subset of the variables (corresponds to setting all other variables to 0) and compute a solution of this reduced linear program. Afterwards, we check if the addition of one or more variables, currently not in the linear program, might improve the LP-solution. This check can be done by the computation of the reduced costs of the variables. In our case, a variable of negative reduced cost can improve the solution.

Column generation (an introduction to column generation can be found in [BJN$^+$94, BJN$^+$98] and in Danish in [MJ, chapter 7]) has turned out to be an efficient method for a range of vehicle routing and scheduling problems. In our case we have to drop the integrality constraints (3.3). This changes our our integer program to a linear denoted the *linear relaxation of the integer program*. The drawback is that we might end with a non-integral solution (if we are lucky the solution is integer and we are finished). To guarantee that we end up with an integer solution we use Branch-and-Bound to close the gap between the lower bound (the result of the relaxation) and the integer solution. Column generation used together with Branch-and-Bound is denoted Branch-and-Price (due to the analogy with

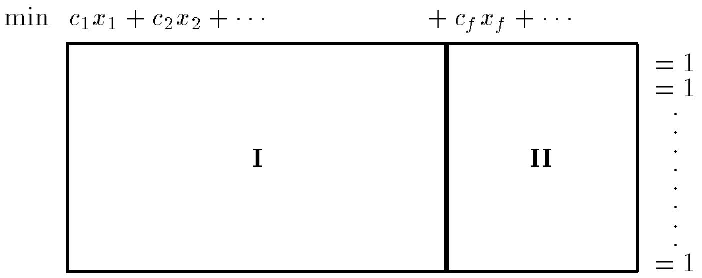


Figure 3.4: An illustration of the column generation approach. The columns in part I are present, whereas the columns in part II are "represented" by the pricing algorithm.

Branch-and-Cut where rows instead of columns are added successively to the problem).

In figure 3.4 part I of the constraint matrix are the routes actually generated and included while the remaining routes (part II) are "represented" by a *pricing algorithm*. If no variables have positive reduced costs then the current optimal solution can not be improved further, one normally says that "all variables price out correctly". In the case that the variables do not price out correctly we add the column(s) with negative reduced cost to the problem, re-optimize and run our pricing algorithm again. Sometimes, one applies a heuristic pricing algorithm instead of an exact pricing algorithm. Thereby the entire algorithm becomes a heuristic as it is not guaranteed that all variables price out correctly. As lower bound used by the Branch-and-Bound part of the algorithm is not guaranteed, optimal bounding is performed using values that may be larger than the optimal value, this may result in nodes being fathomed that would not have been fathomed if the optimal value of the lower bound was known. The Branch-and-Bound therefore also becomes a heuristic.

In our case, the subproblem (to be solved by the pricing algorithm) is an ESPPTWCC, as all constraints are handled here. The reduced cost r_j of a non-basic variable j corresponding to a LP-solution with dual variables

$\pi \in R^m$ is defined as:

$$r_j = c_j - \pi^T \delta_{.j}$$

where c_j is the cost of the route and $\delta_{.j}$ is the "route vector".

The column generation part of the Branch-and-Price is summarized in figure 3.5.

3.3.2 Branch-and-Bound

As previously described the column generation approach does not necessarily lead to an integer solution. Instead we may end up with a solution that is not feasible for the VRPTW at all. At this point we can apply another basic algorithmic technique called Branch-and-Bound.

The problem we have to solve is to minimize a function $f(x)$ over a set of variables $(x_1, x_2, \ldots, x_n)$ over a set of feasible solutions $\mathcal{S}$, that is,

$$\begin{aligned} \min \quad & f(x), \ s.t. \\ & x \in \mathcal{S}. \end{aligned}$$

Branch-and-Bound is a divide-and-conquer approach that dynamically searches through a search tree, where the original problem is divided into smaller problems (by adding more constraints). When dividing a problem into smaller problems the original problem is denoted the parent node and the smaller problems are usually denoted *subproblems, child nodes* or *subspaces*. Note that the terminology subproblem may conflict with the subproblem in the column generation approach. We will therefore use the work subspace. In the worst case all subspaces in the search tree have to be explored.

Before starting the exploration of the search tree, we either have to produce a feasible solution e.g. using an approximation algorithm or a heuristic, or we may simply set the upper bound sufficiently high ("∞"). It should be noted that the existence of a good initial feasible solution is important as it limits the number of subspaces that have to be explored. The solution is known as the *global upper bound* (or sometimes the *current best solution* or

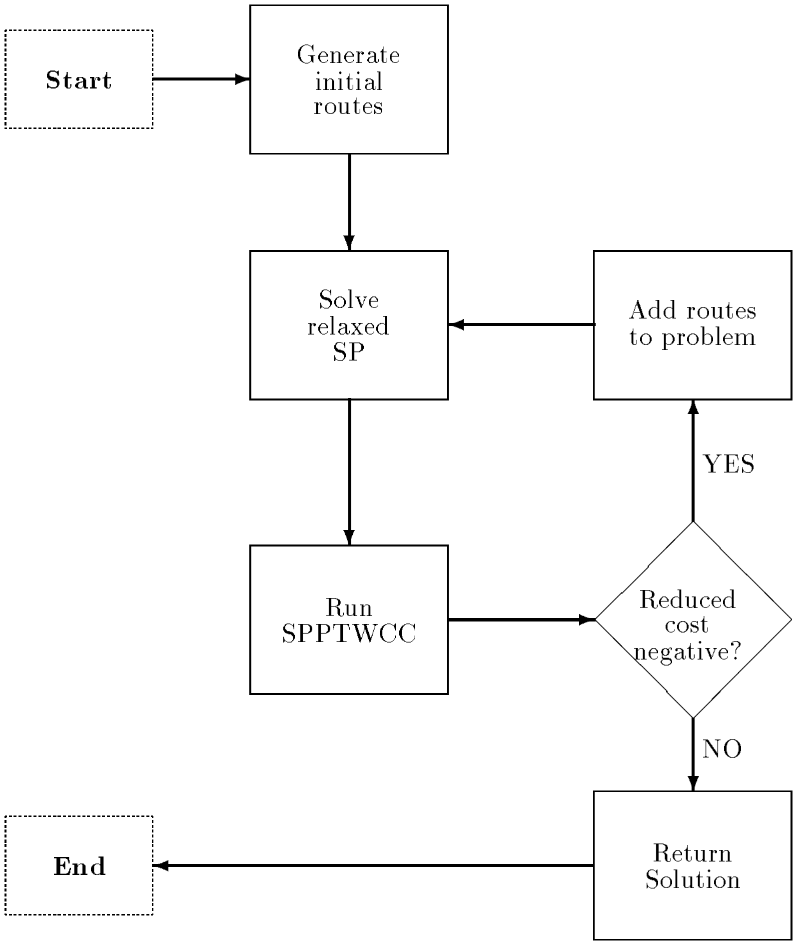


Figure 3.5: Diagram of the column generation part of the VRPTW algorithm.

the *incumbent*). It is possible however to start without an initial solution by setting the initial value of the global upper bound to either a large number that is definitely an upper bound or use the total accumulated distance of all *depot* - i - *depot* routes.

In each iteration of the Branch-and-Bound algorithm, an unexplored child node is chosen. The *bounding function* is then called and produces a *local lower bound.* If the global upper bound is smaller than the local lower bound the child node may be discarded (sometimes called *fathomed*), as no feasible solution contained in the set represented by the subspace can be better than the existing global upper bound. In the case where the local lower bound is smaller than the global upper bound, we check if the local lower bound is the value of a feasible solution. If that is the case, the local lower bound becomes the new global upper bound. If the local lower bound is smaller than the value of the global upper bound we perform a branching operation thereby splitting the child node into a number of even smaller subspaces, as there is a possibility for a better feasible solution in the subspace than the current global upper bound.

A typical Branch-and-Bound algorithm therefore consists of three major components:

1. Bounding
2. Selection
3. Branching

Bounding

In order to evaluate a given subspace, a bound value is computed. In our case, a lower bound is computed, that is, no feasible solution in a given subspace can attain a value lower than this bound. The bound used for the VRPTW is the LP relaxation of the IP model of the set partitioning problem, and as a part of the bounding process, the column generation is performed.

If a subspace has a lower bound larger than the current global upper bound, the subspace can be discarded as mentioned before. Two strategies for assigning lower bounds to subproblems are often used and investigated. In one, *lazy evaluation*, we postpone the computing of the actual bound as long as possible, and in the other, *eager evaluation*, we do it as soon as possible.

In lazy evaluation, the bound of the parent node is often used. As more restrictions are placed on the child node the bound of the parent node is also a lower bound for the child node albeit maybe a weak one. The advantage is that we avoid doing bound calculations if the parents lower bound is already worse than the global upper bound.

The idea behind eager evaluation is to bound the subspaces as soon as possible and thereby assign a stronger bound to the subspaces than in lazy evaluation. Of course the drawback is that more bound calculations have to be performed than in the lazy evaluation strategy.

Selection

During the execution of the Branch-and-Bound algorithm we have a set $\mathcal{U}$ of generated but unexplored subspaces. The selection of the next subspaces to be evaluated is performed by the *selection function* h, where we chose the subspace that has the minimum value of h.

In *Best-first Search* (BeFS) the subspace to be explored is the one that has the lowest lower bound, that is, $h \equiv$ bounding function. Here no superfluous computation is done once the optimal solution has been found. Even though this may seem like a good idea, serious memory problems may arise as the search tree grows exponentially as a function of the depth.

Another selection function, which exhibits the same memory problem, is the *Breadth-First Search* (BFS). Here $h \equiv L$, where L is the level of the subspaces. The level of a subspace is the depth of its position in the Branch-and-Bound tree, i.e. the root node is on level 0, its children on level 1 etc.

In the *Depth-First Search* (DFS) with $h \equiv -L$ (that is, the child node with the largest level is selected) the memory usage is only linear in the depth of the search tree. Additionally, it tends to be simpler to implement as the generated child nodes can be stored on a stack. The drawback is that a large number of subspaces may have to be explored if the gap left by the initial setting of the global upper bound is large.

Sometimes a combination of DFS and BeFS is used. Then DFS is used as the main principle of selection, and BeFS is used to select among subspaces on the same level.

Branching

Just as bounding, branching is a decision related to the problem we are trying to solve. During the years a number of different branching operations for the VRPTW has been proposed in the literature [DDS92, Hal92, Koh95, GDDS95]. We will describe the following methods:

1. branching on the number of vehicles,
2. branching on the flow variables (x_{ijk}),
3. branching on the sums of flow variables, and
4. branching on resource windows.

Branching on the number of vehicles

This branching rule was proposed by Desrochers, Desrosiers and Solomon in [DDS92]. If our bounding function returns a solution where the number of vehicles k is fractional it is natural to introduce a bound on the number of vehicles.

We branch on the number of vehicles by creating two child nodes equal to the current subspace but adding $\sum_{j \in \mathcal{C}} f_{0j} \geq \lceil k \rceil$ respectively $\sum_{j \in \mathcal{C}} f_{0j} \leq \lfloor k \rfloor$ to the child nodes in the master problem. Here f_{ij} denotes the "flow" of vehicles on the arc (i, j).

Branching on flow variables

Branching on a single variable x_{ijk} is only possible if each vehicle can be distinguished. In the column generation environment this can be done by solving a subproblem (SPPTWCC) for *each* vehicle and in the master problem we introduce an additional constraint

$$\sum_{p \in \mathcal{P}_k} y_p = 1$$

for each vehicle k, where $\mathcal{P}_k$ is the set of routes generated for vehicle k and y_p is either 0 or 1 ensuring that each vehicle drives exactly one of the generated routes. This ensures that each vehicle is used for one path only. As all our vehicles are identical this option will not be considered any further.

Instead of branching on a single variable x_{ijk} we can branch on the sums of flow. This can be done in two ways: either $\sum_j x_{ijk}$ or $\sum_k x_{ijk}$ (equivalent to f_{ij}).

Branching on $\sum_j x_{ijk}$ means fixing customers i to vehicle k, which implies that the subproblems of different vehicles are not identical. If we instead branch on $\sum_k x_{ijk}$ this can be done by fixing arcs (i, j) in the graph (that is, changing the graph for *all* vehicles). So the subproblems remain identical.

Branching on $\sum_k x_{ijk}$ is equivalent to branching on single flow variables f_{ij}. As all subproblems remain identical only one SPPTWCC must be solved in each iteration. The branch $f_{ij} = 0$ is imposed by removing the arc (i, j) from the graph. The other branch, $f_{ij} = 1$, is imposed by removing arcs originating in i (unless i is the depot) and arcs terminating in j (unless j is the depot) except for the (i, j) arc. This forces a route to continue directly to j after visiting customer i. We have used this branching scheme in our basic implementation.

The branching decision $\sum_j x_{ijk}$ was first proposed by Halse in [Hal92]. Imposing $\sum_j x_{ijk} = 0$ is done by removing all arcs originating or terminating in customer i for vehicle k, and $\sum_j x_{ijk} = 1$ is imposed by removing all arcs originating or terminating in customer i for all vehicles but vehicle k. This will "force" customer i to be visited by vehicle k.

In order to create more complex strategies the branching schemes can be mixed.

All branching rules discussed in this section will produce an integer solution in a finite number of steps.

Branching on resource windows

The idea of branching on resource windows is introduced by Gélinas et al. in [GDDS95]. In our VRPTW model we can branch on the time windows, but the capacity constraint can also be viewed as resource windows. We will only discuss branching on time windows, as the capacity is significantly less constraining in most cases. In [GDDS95] only branching on time windows is used.

In branching on time windows we branch by splitting a time window into two time windows. The branching has to be done to ensure that at least one route is infeasible in each of the two "sub windows".

In order to branch on time windows we have to make the following decisions:

1. How should we choose the node for branching?
2. Which time window should be divided?
3. Where should the time window be divided?

In order to determine where branching on time windows is possible we define *feasibility intervals* $[l_i^r, u_i^r]$ for all vertices $i \in \mathcal{N}$ and all routes r with fractional flow. l_i^r is the earliest time that service can start at vertex i on route r, and u_i^r is the latest time that service can start, that is, $[l_i^r, u_i^r]$ is the time interval during which route r must visit vertex i to remain feasible.

The intervals can easily be computed by a recursive formula. Let the fractional route r be defined by $r = \langle i_0, i_1, i_2, \ldots, i_e \rangle$ where $i_0 = 0$ and $i_e = n + 1$. The intervals can then be calculated by:

$$l_{i_k}^r = \begin{cases} 0 & \text{if } k = 0 \\ \max\{l_{i_{k-1}}^r + t_{i_{k-1}, i_k}, a_{i_k}\} & \text{if } k = 1, 2, \ldots, e-1 \end{cases} \tag{3.4}$$

$$u_{i_e}^r = \begin{cases} T & \text{if } k = e \\ \min\{u_{i_{k+1}}^r - t_{i_k, i_{k+1}}, b_{i_k}\} & \text{if } k = e-1, e-2, \ldots, 1 \end{cases} \tag{3.5}$$

If a route is visiting a customer more than once it results in a feasibility interval for *each* visit. Additionally we define

$$L_i = \max_{\text{fractional routes } r} \{l_i^r\}, \quad i \in \mathcal{N} \tag{3.6}$$

$$U_i = \min_{\text{fractional routes } r} \{u_i^r\}, \quad i \in \mathcal{N} \tag{3.7}$$

Now if $L_i > U_i$ at least two routes (or two visits by the same route) have disjoint feasibility intervals, i.e. the vertex is a candidate for branching on time windows. It should be noted that situations can arise where there are no candidates for branching on time windows, but the solution is not feasible. We can branch on a candidate vertex i by dividing the time windows $[a_i, b_i]$ at any integer value in the open interval $[U_i, L_i[$.

Here three different strategies are proposed in [GDDS95]:

Elimination of cycles: As will be discussed in chapter 4 there may be cycles in the routes generated. These cycles leads to lower bounds which are not as tight as if we did not allow for cycles. This strategy therefore tries to select a candidate vertex where a cycle would be removed if the time window is split.

Number of visits: Here the vertex selected is the candidate vertex i with a minimum number of visits. We thereby hope that the number of routes that remain feasible in the two new problems is small.

Flow values: Dividing a time window into two smaller ones results in elimination of flow in both part. Let α_1 be the flow eliminated by choosing the first part and α_2 be the flow eliminated by using the second part of the time window. Here we choose the vertex i where

$$\alpha_1 + \alpha_2 - |\alpha_1 - \alpha_2|$$

is maximized. Hence, we select the candidate vertex for which the total quantity of flow eliminated for the two new problems is both maximal and best balanced.

We will elaborate further on these criteria later.

After having chosen the candidate vertex i for branching we now have to choose an integer $t \in [U_i, L_i[$ in order to determine the division.

Now, let Γ_i be the total number of feasibility intervals for vertex i. To avoid too many indices we omit the route index r from now on. The feasibility interval $[l_i^\gamma, u_i^\gamma]$ is the γth visit to vertex i. The total amount of flow visiting every vertex is 1 and each of the Γ_i visits contributes with a certain amount. We now try to divide the time window of vertex i in a way to 1) balance the eliminated amount of flow from each side and 2) make it as large as possible.

In order to choose t, we determine the bounds the l_i and u_i, $t \in [u_i, l_i[$. Let $A_i(t)$ be the flow eliminated if the time window is restricted to $[a_i, t]$ and let $B_i(t)$ be the flow eliminated if the time window is restricted to $[t+1, b_i]$. We now determine l_i and u_i in order to satisfy:

$$|A_i(u_i) - B_i(l_i)| \text{ is minimized} \tag{3.8}$$

$$[u_i, l_i[\subseteq [U_i, L_i[\text{ and } [u_i, l_i[\neq \emptyset \tag{3.9}$$

$$l_i \text{ is chosen among the } l_i^\gamma\text{s} \tag{3.10}$$

$$u_i \text{ is chosen among the } u_i^\gamma\text{s} \tag{3.11}$$

$$l_i^\gamma \notin]u_i, l_i[\text{ and } u_i^\gamma \notin]u_i, l_i[\tag{3.12}$$

We want our solution to result in a split that balances the flow as mush as possible (3.8). Condition 3.9 ensures that the current optimal solution becomes infeasible in both the two new problems, while the next two conditions ensure that l_i and u_i corresponds to boundaries of feasibility intervals. The final condition 3.12 ensures that we eliminate as much flow as possible.

The bound l_i and u_i can easily be computed. All l_i^γs and u_i^γs (hereafter referred to as *points*) are sorted together in non-decreasing order (and l_i^γs are put before u_i^γs in case of equal value). Now we run through the list starting from the first element. Here $A_i(t)$ is equal to 1 and $B_i(t)$ is equal to 0. Every time we encounter a lower bound, $A_i(t)$ is decreased with the amount of flow it contributes, and every time an upper bound is encountered it is increased with the amount of flow it contributes. In case we meet an upper bound immediately followed by a lower bound a *candidate split* is found and its $|A_i(u_i) - B_i(l_i)|$ value is determined. After inspecting all points the best pair of lower and upper bound is returned.

In [GDDS95] the branching on resource constraints is compared to one (unspecified) strategy for branching on flow variables. In 15 out of 21 benchmark tests branching on time windows are fastest, while branching on flow variables is best in the remaining 6 cases. In 8 of the 21 instances branching on time windows in more than twice as fast as branching on flow variables.

3.4 Achieving tighter lower bounds

Often the only way to solve difficult integer programming problems is to use a relaxation of the original problem. Typically the optimal solution to the relaxed problem is not feasible in the original problem. To get an integer solution the most popular method has been to use Branch-and-Bound.

Another method is to try to improve the polyhedral description of the relaxed problem in order to get an integer solution or at least tighten the bound.

Let **C** be the set of constraints that are defining the VRPTW ((2.2) – (2.9)), and let **S** be the polytope defined by the same constraints, but now with the integrality constraints relaxed, that is, instead of $x_{ijk} \in \{0, 1\}$ we now have $0 \leq x_{ijk} \leq 1$.

If it is possible to describe the *convex hull* of **C** we can use ordinary linear programming to solve the problem if the description is "small enough". The LP with constraints defining the convex hull will automatically result in a feasible and optimal **integer** solution.

Alas, finding an inequality description of the convex hull of **C** is not easy (see [NW88]).

So, in general the optimal solution to **S** will be fractional. Therefore we try to identify an inequality (a cut) that separates the optimal solution from all the integer solutions. If such an inequality can be found it is added to the problem and the new enhanced problem is solved resulting in a new optimal solution. Generally this leads to an improvement in the objective function. As long as we can find new cuts this process can be continued iteratively.

There exists both general cuts and cuts specific for a given problem. One of the general cuts are the *Gomory-Chvátal* cuts (see [NW88, CCPS98]). The typical problem with the general cuts is that they are rather "weak", that is, only a very small part of the solution space is removed. Therefore attention has mainly been focused on the problem-specific cuts, especially after the success obtained by Padberg, Rinaldi, Grötschel and Holland using cuts to solve the TSP.

The "deepest" cuts are those associated with the *facets* of the convex hull of **S**. Algorithms that identify cuts are called *separation algorithms*; they play the same role for cuts as the pricing algorithm does for column generation. Often one distinguishes between heuristic separation algorithms and exact separation algorithms. An exact separation algorithm finds a cut if one exists, whereas if something is found by a heuristic separation algorithm it is a cut, but cuts may actually exist without being detected.

If the process of detecting and adding (and in more complex situations also managing) cuts is done in every (or almost every) node of the Branch-and-Bound tree we have a Branch-and-Cut algorithm.

We will now briefly describe the cuts used in our implementation of the algorithm for solving the VRPTW. Note that in identifying cuts for the VRPTW we can not construct valid inequalities in the variables of the master problem, that is, valid inequalities based on the Set Partitioning Problem. As there is no way of transferring the simplex multipliers from these constraints to the subproblem this is not compatible with the column generation context. We therefore have to detect the cuts on the basis of ((2.2) – (2.9)) and then "translate" the cuts into the Set Partition Problem formulation.

3.4.1 The Subtour elimination constraints

The sub-tour elimination constraints originate from the TSP. Let us therefore now consider the usual LP-formulation of the TSP. Given a graph $\mathcal{G} = (N, E)$, let c_e and x_e, $e \in E$, be the cost respectively the decision variable for arc e. If $x_e = 1$ the arc e is part of the solution, if $x_e = 0$ it is not. $\delta(S)$ denotes the set of arcs with exactly one endpoint in S. Finally the complement of set S is denoted $\bar{S}$. TSP can now be stated as:

$$\min \sum c_e x_e, \quad \text{s.t.} \tag{3.13}$$
$$\sum_{e \in \delta(i)} x_e \quad = 2 \qquad \forall i \in N \tag{3.14}$$
$$\sum_{e \in \delta(S)} x_e \quad \geq 2 \quad \forall S \subset N, S \neq \emptyset \tag{3.15}$$
$$x_e \in \{0, 1\} \tag{3.16}$$

The constraints (3.14) are denoted *assignment constraints*, while the constraints (3.15) are the *subtour elimination constraints* of which there are exponentially many. An LP-based solution procedure starts with only assignment constraints and (3.16) relaxed to $0 \leq x_e \leq 1$. Now the subtour elimination constraints are generated as they are needed.

For the VRPTW the flow out of a set S of customers (denoted $X(S)$ will be: $X(S) = \sum_{k \in V} \sum_{i \in S} \sum_{j \in N \setminus S} x_{ijk} = \sum_{i \in S} \sum_{j \in N \setminus S} f_{ij}$.

The weak form of subtour elimination inequalities are given by

$$X(S) \geq 1.$$

Solving the SPPTWCC, this inequality is not necessarily satisfied. Had we instead solved the ESPPTWCC in our effort to generate new columns, Kohl has shown in [Koh95] that the weak form of the subtour elimination inequalities would automatically be satisfied.

In the case of running SPPTWCC, violated valid weak subtour elimination constraints can be identified in polynomial time for the equivalent separation problem.

3.4.2 Comb-inequalities

Subtour-elimination constraints alone do not guarantee integrality of the LP solution. Comb-inequalities form another class of inequalities that are often added to further strengthen the LP.

A comb consists of a *handle* (denoted H) and an odd number (greater than 1) of *teeth* (denoted $W_1, W_2, W_3, \ldots, W_h$). The handle and the teeth are sets of vertices. The teeth are disjoint, and all teeth share at least one vertex with the handle and have at least one vertex that is not part of the handle.

The arcs in the comb are the ones with either both endpoints in the handle or both endpoints in one of the teeth. If an arc have both endpoints in the handle and a teeth it will be counted twice in the inequality shown below. A comb inequality is now given by:

$$\sum_{e \in E(H)} x_e + \sum_{i=1}^{h} \sum_{e \in E(W_i)} x_e \leq |H| + \sum_{i=1}^{h}(|W_i| - 1) - \frac{h+1}{2}.$$

A comb with one handle and three teeth is the smallest comb (see figure 3.6). The concept has been generalized by Grötschel and Pulleyblank to inequalities with several handles with associated teeth (called *clique trees*).

A polynomial separation algorithm for the special case where $|W_i| = 2$ is given by Padberg and Rao; an efficient implementation of this algorithm and a very fast heuristic is given by Grötschel and Holland. For more general classes of combs, no polynomial separation algorithms are known.

3.4.3 2-path inequalities

As mentioned earlier cuts especially for the VRPTW have been developed and discussed by Kohl in [Koh95] and by Kohl et al. in [KDM+99]. These cuts are also used in our implementation and are now briefly described. It should be noted that in [Koh95] the separation algorithm is only run in the root node of the Branch-and-Bound tree, therefore the implemented algorithm is not a Branch-and-Cut algorithm.

The basis of this set of cuts is the subtour elimination inequality in the strong form:

$$x(S) \geq k(S) \quad \forall S \subseteq \mathcal{C}$$

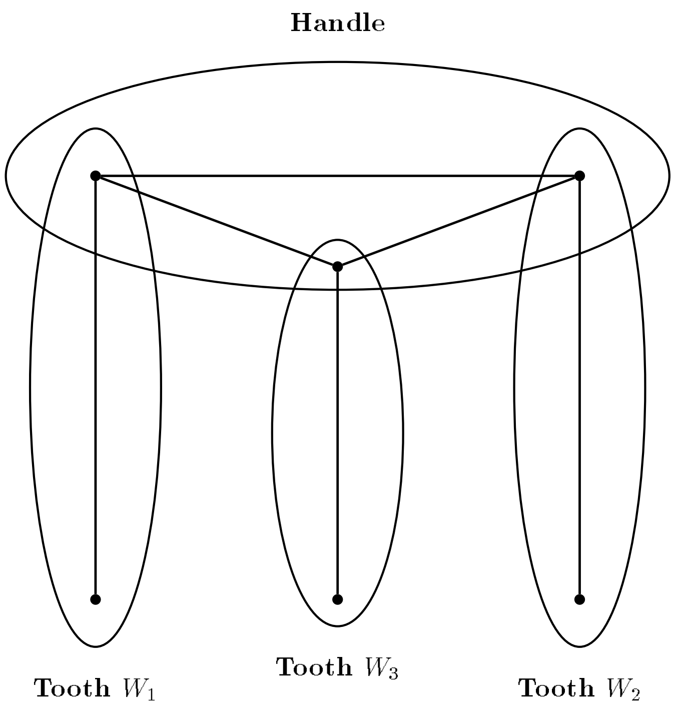


Figure 3.6: The simplest possible comb inequality.

where $x(S)$ is the amount of flow leaving the set S, and $k(S)$ is the minimum number of vehicles needed to service the customers in S. Now finding $k(S)$ is very hard, but using the fact that the travel times satisfy the triangle inequality we have

$$S_1 \subset S_2 \Rightarrow k(S_1) \leq k(S_2).$$

Using this fact one tries to find sets S that satisfy $x(S) < 2$ and $k(S) > 1$. As $k(S)$ is an integer, $k(S) > 1$ implies $k(S) \geq 2$. That is, we are trying to identify sets S that requires at least two vehicles to be serviced, but are currently (in the fractional solution) serviced by less than two vehicles.

For a given set S two checks have to be performed. First we check whether $k(S) > 1$, which is easily done, as we just have to check whether there is sufficient capacity in one vehicle. The second check is to test whether the customers of S can be serviced by a single vehicle, that is, solving a corresponding Traveling Salesman Problem with Time Windows *feasibility* problem. This problem is $\mathcal{NP}$-hard in the strong sense (proved by Savelsbergh in [Sav56]), but as the size of S will be rather small in our applications, it will remain manageable. This part is solved using dynamic programming. Note that the hardest problems are the infeasible ones as all possibilities have to be checked.

Finding the sets S for which $x(S) < 2$ are found using a heuristic. Starting with $S = \emptyset$ customers are added to S as long as $x(S) < 2$. If no additional customer can be added without violating $x(S) < 2$ then checking $k(S) > 1$ is performed.

Implementing the cuts in our set partitioning environment is by no means straightforward as the customers are not variables in the master problem of the column generation model. Our variables are the routes generated. If a route contains at least one customer that is in the detected set S the coefficient of that route in the constraint we wish to add must be non-zero. Now recall that $x(S)$ is the flow out of the set S. If the variable for a route is non-zero it represents a flow through that route. So if one of the customers in S is on such a route it is contributing to the flow out of S, unless it's successor is also a member of S. So the coefficient of a given route is the number of customers that is member of S minus the number

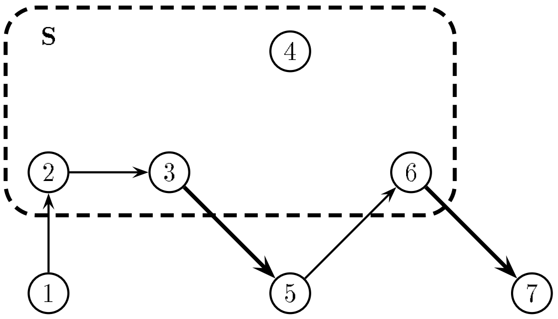


Figure 3.7: Determining the coefficient for a route/cut-pair. This route (1-2-3-5-6-7) twice leaves the set S therefore the coefficient is equal to two, which is equal to the number of customers in S minus the number of consecutive customers in S.

of consecutive pairs of members of S, as only the last customer in that sequence is contributing to $x(S)$ (see figure 3.7).

So for the root node we extend our column generation scheme depicted in 3.5 to also include the separation algorithm for cut generation (see figure 3.8).

While subtour- and comb-inequalities are valid no matter where in the Branch-and-Bound tree they are identified (we say they are *globally* valid), this is generally not the case for the 2-path cuts.

No matter where in the Branch-and-Bound tree a 2-path cut is found it is globally valid if $k(S) \geq 2$. If the underlying graph is changed the 2-path cut generated can be globally valid, but it is not guaranteed.

Branching on vehicles does not changes the underlying structure of the problem. Therefore a cut found in a Branch-and-Bound node where only vehicle restrictions have been imposed is also globally usable. If a TSPTW tour can not be found here it can not be found in the root node either and therefore the cut is global.

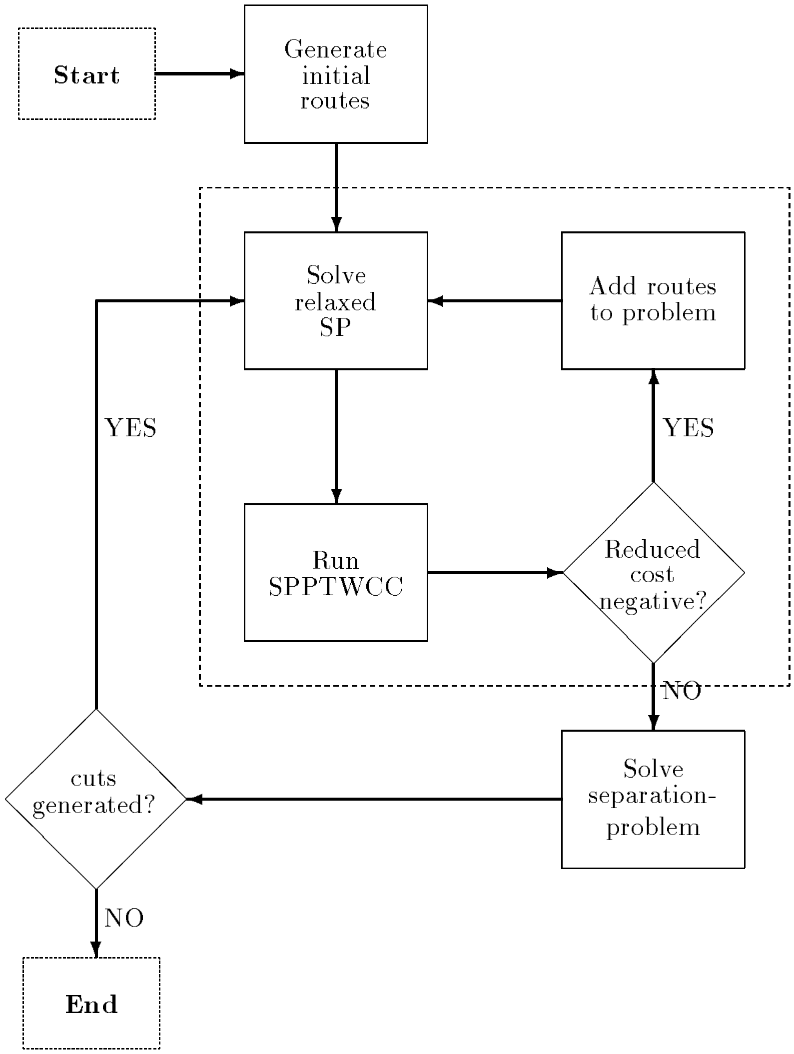


Figure 3.8: Diagram of the column generation part of the VRPTW algorithm.

If we branch on arcs (also denoted branching on flow) we have two situations:

- Lets assume we have deleted the arc (x, y) in the graph. If a cut $\mathcal{C}$ contains x but not y (or vice versa) the cut will be general applicable. As y is not among the customers in the set it will have no influence on the feasibility of the TSPTW problem.

  If $\mathcal{C}$ is a cut containing both x and y the cut is not necessarily general. The might exist a TSPTW tour in the root node by using the missing (x, y) arc.

- Now lets assume we have deleted all arcs leaving x and entering y but (x, y). Furthermore we assume that the cut contains x but not y. Clearly no TSPTW tour can be generated as there is no way we can leave x. Therefore it may be highly possible that a TSPTW exists in the original graph. Same arguments can be made in the symmetric case.

  In the case where both x and y are members of the cut we are forced to use the (x, y) arc. In the root this is not the case and it might therefore be possible to generate a TSPTW tour.

  So these cuts are not necessarily general but can be.

In the case of time window reduction it is quite obvious that a cut using a customer with reduced time windows might actually not be a cut in the root as the larger original time window might allow a TSPTW tour.

3.5 Putting it all together

This chapter has described the basic approach to solving the VRPTW using column generation and cuts. Figure 3.9 recaps the overall structure of the algorithm.

The master problem is initiated by generating the *depot* - i - *depot* routes for every i. The dashed box in figure 3.9 contains the parts of the algorithm that are only executed in the root. In the root if no more routes can be

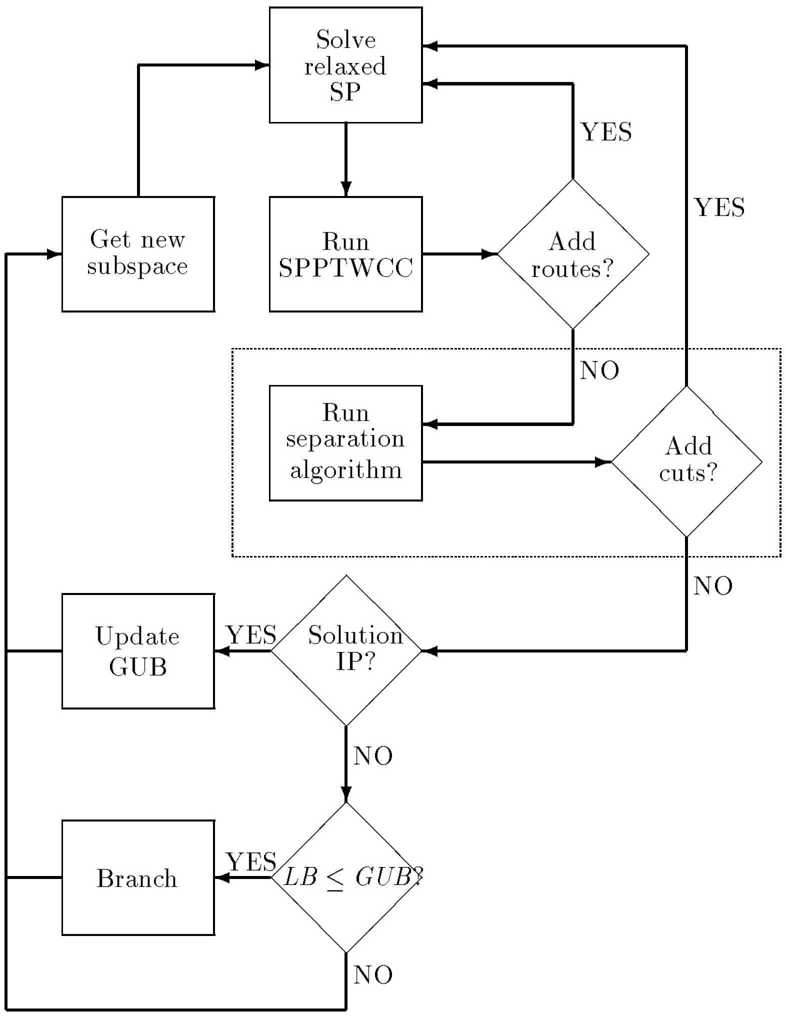


Figure 3.9: Overall structure of the algorithm for solving the VRPTW exact using column generation and cuts. The two segments of code in the dashed box is only executed in the root node.

generated the algorithm tries to identify some cuts that can strengthen the formulation. Hereafter the route generating algorithm is run again. Only when no more routes **and** no more cuts can be generated the lower bound of the Branch-and-Bound node is determined. In all other Branch-and-Bound nodes than the root the algorithm does not run the separation algorithm.

Now if the solution is integer we compare it with the current global upper bound (GUB). If the lower bound is not integer, and the value of the lower bound is higher than GUB then node is fathomed, otherwise the Branch-and-Bound node is divided into two new child nodes, and these are entered into the datastructure that holds the unsolved subspaces.

Finally the next subspace to be solved is fetched from the pool of unsolved Branch-and-Bound nodes. If the pool is empty the algorithm prints out the solution and terminates. The column generator is run at all nodes in the Branch-and-Bound tree.

Chapter 4

Shortest Path with Time Windows and Capacity Constraints

As seen in the previous chapter the ESPPTWCC must be solved as a subproblem. In this chapter we describe an algorithm for solving the subproblem of the column generation approach.

4.1 The mathematical model

Using the column generation approach introduced in section 3.1 with the set partitioning problem as the master problem, the subproblem becomes the following mathematical model:

$$\min \sum_{i \in \mathcal{N}} \sum_{j \in \mathcal{N}} c_{ij} x_{ij}, \qquad s.t. \tag{4.1}$$

$$\sum_{i \in \mathcal{C}} d_i \sum_{j \in \mathcal{N}} x_{ij} \leq q \tag{4.2}$$

$$\sum_{j \in \mathcal{N}} x_{0j} = 1 \tag{4.3}$$

$$\sum_{i \in \mathcal{N}} x_{ih} - \sum_{j \in \mathcal{N}} x_{hj} = 0 \qquad \forall h \in \mathcal{C} \tag{4.4}$$

$$\sum_{i \in \mathcal{N}} x_{i,n+1} = 1 \tag{4.5}$$

$$s_i + t_{ij} - K(1 - x_{ij}) \leq s_j \qquad \forall i, j \in \mathcal{N} \tag{4.6}$$

$$a_i \leq s_i \leq b_i \qquad \forall i \in \mathcal{N} \tag{4.7}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \in \mathcal{N} \tag{4.8}$$

Constraint (4.2) is the capacity constraint, constrains (4.6) and (4.7) are time constraints, while constraint (4.8) ensures integrality. The constraints (4.3), (4.4) and (4.5) are flow constraints resulting in a path from the depot 0 and back to the depot $n+1$. The $\hat{c}_{ij}$ is the *modified cost* of using arc (i, j), where $\hat{c}_{ij} = c_{ij} - \pi_i$. Note that while c_{ij} is a non-negative integer $\hat{c}_{ij}$ can be any real number. As we are now dealing with one route the index k for the vehicle has been removed.

As can be seen from the mathematical model above the subproblem is a shortest path problem with time windows and capacity constraints where each vertex can participate at most once in the path/route. For this problem (sometimes denoted the Elementary Shortest Path Problem with Time Windows and Capacity Constraints (ESPPTWCC)) there is no known efficient algorithm, making the problem unsolvable for practical purposes. Therefore some of the constraints are relaxed. Cycles are allowed thereby changing the problem to the Shortest Path Problem with Time Windows and Capacity Constraints (SPPTWCC). Even though there is a possibility for negative cycles in the graph the time windows and the capacity constraints prohibits infinite cycling. Note that capacity is accumulated every time the customer is serviced in a cycle.

Arcs can now be used more than once, and customers may therefore be visited more than once. This must be incorporated into the model above. We therefore replace decision variables x_{ij} and s_i with x_{ij}^l and s_i^l. Now x_{ij}^l is set to 1 if the arc (i,j) is used as the l'th arc on the shortest path, and 0 otherwise, and s_i^l is the start of service of customer i as customer number l, where $l \in \mathcal{L} = \{1, 2, \ldots, |\mathcal{L}|\}$, $|\mathcal{L}| = \lfloor \frac{b_{n+1}}{\min t_{ij}} \rfloor$ (remember that b_{n+1} is the closing time of the depot). The SPPTWCC can now be described be the following mathematical model:

$$
\begin{aligned}
&\min \sum_{l\in\mathcal{L}} \sum_{i\in\mathcal{N}} \sum_{j\in\mathcal{N}} \hat{c}_{ij} x_{ij}^l, && s.t. && && (4.9)\\
&\sum_{i\in\mathcal{N}} \sum_{j\in\mathcal{N}} x_{ij}^1 && = 1 && && (4.10)\\
&\sum_{i\in\mathcal{N}} \sum_{j\in\mathcal{N}} x_{ij}^l - \sum_{i\in\mathcal{N}} \sum_{j\in\mathcal{N}} x_{ij}^{l-1} && \leq 0 && \forall l \in \mathcal{L} - \{1\} && (4.11)\\
&\sum_{i\in\mathcal{C}} d_i \sum_{l\in\mathcal{L}} \sum_{j\in\mathcal{N}} x_{ij}^l && \leq q && && (4.12)\\
&\sum_{j\in\mathcal{N}} x_{0j}^1 && = 1 && && (4.13)\\
&\sum_{i\in\mathcal{N}} x_{ih}^{l-1} - \sum_{j\in\mathcal{N}} x_{hj}^l && = 0 && \forall h \in \mathcal{C}\ \forall l \in \mathcal{L} - \{1\} && (4.14)\\
&\sum_{l\in\mathcal{L}} \sum_{i\in\mathcal{N}} x_{i,n+1}^l && = 1 && && (4.15)\\
&s_i^l + t_{ij} - K(1 - x_{ij}^l) && \leq s_j^l && && \\
& && && \forall i,j \in \mathcal{N}\ \forall l \in \mathcal{L} - \{1\} && (4.16)\\
&a_i \leq s_i^l \leq b_i && && \forall i \in \mathcal{N} && (4.17)\\
&x_{ij}^l \in \{0,1\} && && \forall i,j \in \mathcal{N} && (4.18)
\end{aligned}
$$

(4.10) states that the first arc can only be used once, while (4.11) states that arc l can only be used provided that arc $l-1$ is used. The remaining constraints are the enhancements of the "original" constraints (2.3) to (2.9) with the additional superscript l and the related changes. Note that (4.10) is redundant as it is covered by (4.13), but it has been kept in the model as to indicate the origin of the problem.

4.2 A Dynamic Programming Algorithm

In this section we will present a pseudo-polynomial dynamic programming algorithm for the SPPTWCC.

4.2.1 The algorithm

The Shortest Path Problem (SPP) (without any additional constraints) is a fundamental problem in network optimization. Hundreds of papers on SPP have been written (see for example the surveys [GP88, CGR93]) and it must be considered one of the truly classical problems in algorithmics.

Even though *Dijkstras algorithm* for solving SPP might not seem useful in relation to SPPTWCC (arcs must be non-negative and it does not operate with additional constraints) it is a good starting point for building our dynamic programming algorithm for solving SPPTWCC.

The Dijkstra algorithm is very simple, and intuitively very easy to understand. Lets assume we have a graph with non-negative edge-costs (l_{ij}) and a vertex s as the origin of the shortest paths we are about to find. Initially we have a list of all vertices not yet visited (the algorithm starts by visiting s), and for each vertex i we have a label d_i which is the length of the shortest path from s to i found so far (initially $d_s = 0$ and $d_i = \infty$, $i \neq s$). In each step of the algorithm the vertex with the smallest label among all unvisited vertices is visited. When the algorithm visits a vertex i it checks all out-going arcs (i, j) whether $d_i + l_{ij} < d_j$ holds. If this is the case, d_j is updated to $d_i + l_{ij}$ otherwise d_j remains unchanged. Vertex i is the removed from the list of unvisited vertices, and unless the set of unvisited vertices is empty a new step of the algorithm is started. The Dijkstra algorithm can be described by the dynamic program:

$$\begin{aligned} d_s &= 0, \\ d_j &= \min_{(i,j)\in\mathcal{A}} \{d_i + l_{ij} \mid j \in V \setminus \{s\}\}. \end{aligned}$$

As described the Dijkstra algorithm builds up the shortest path by extending "good" paths, and in each iteration the algorithm tries to extend a "good" path to all possible successors. If all possible paths had to be checked the algorithm should in the worst case check and compare exponentially many paths. But by checking whether $d_i + l_{ij} < d_j$ holds or not we try to extend only the "good" paths, the remaining paths are not extended any further. The $d_i + l_{ij} < d_j$ is called a *dominance criterion* and this concept of discarding "bad" paths will be important in our effort to build an efficient SPPTWCC algorithm.

In order to build the SPPTWCC algorithm we have to make two assumptions:

1. Time is always increasing along the arcs, i.e. $t_{ij} > 0$.
2. Time and capacity are discretized.

While a state in Dijkstras algorithm only contained the vertex i, this state is now enlarged with the current time t of arrival and the accumulated demand d to:

$$(i, t, d).$$

The label is then defined as $c(i,t,d)$. The algorithm is based on the following simple extension of the dynamic programming behind the Dijkstra algorithm:

$$\begin{aligned} c(0,0,0) &= 0 \\ c(j,t,d) &= \min_i \{\hat{c}_{ij} + c(i,t',d') \mid t' + t_{ij} = t \wedge d' + d_i = d\}. \end{aligned}$$

States are treated in order of increasing time (t). Note that for each label i there may now exist more than one state. The number of states is given by

$$\Gamma = \sum_{i \in \mathcal{N}} (b_i - a_i)(q - 1).$$

This is nevertheless the upper limit, many of these states might even not be possible, and others will not be considered as they are dominated by other states.

In a straightforward implementation we maintain a set *NPS* of not processed states. Initially this set only has one member: the label $(0, 0, 0)$. As long as there exist unprocessed labels in the set the one with the lowest time is chosen and the algorithm tries to extend this to the successors of the vertex. States at vertex $n + 1$ are not processed and are therefore kept in a special set of "solutions", from which the best one is returned as the algorithm terminates. When a label has been processed it is removed from the set of unprocessed labels. The algorithm is described in pseudo-code in figure 4.1.

This algorithm is denoted a *reaching* algorithm by Desrochers in [Des88], since all successors of a label are treated in the same iteration. In the *pulling* algorithm also presented by Desrochers in the same technical report all predecessors of a given state are treated in the same iteration. We will only use the reaching algorithm in this thesis.

In the paper [DS88b] by Desrochers and Soumis a reoptimization algorithm for the SPPTW is proposed. After having computed the non-dominated paths a number of disjoint paths are produced one at a time. Instead of starting from scratch after having removed the vertices from the optimal path, a primal-dual function is maintained in order to reoptimize the existing paths to the new conditions.

Let Z^0 denote the problem for which the non-dominated paths were obtained originally. The vertices of the best path generated are removed from Z^0 (let us denote the problem Z^1) and the path is returned.

Central to the reuse of the old solution is the fact that the non-dominated paths for a given vertex $j \in \mathcal{N}$ of the solution for Z^0 can be divided into two categories:

⟨ Initialization ⟩
$NPS = \{(0,0,0)\}$
$c(0,0,0) = 0$

repeat
  $(i,t,d) = \mathrm{BestLabel}(NPS)$

  for $j := 1$ **to** $n+1$ **do**
    if $(i \neq j \wedge t + t_{ij} \leq b_j \wedge d + d_j \leq q)$ **then**
      ⟨ Label feasible ⟩
      if $c(j, \max\{t + t_{ij}, a_j\}, d + d_j) > c(i,t,d) + \hat{c}_{ij}$ **then**
        ⟨ New label better ⟩
        $\mathrm{InsertLabel}(NPS,\ (j, \max\{t + t_{ij}, a_j\}, d + d_j))$
        $c(j, \max\{t + t_{ij}, a_j\}, d + d_j) = c(i,t,d) + \hat{c}_{ij}$

until $(i = n+1)$
return

Figure 4.1: The algorithm for finding the shortest path with time windows and capacity constraints. *BestLabel* returns a label with vertex different from $n+1$ and minimal accumulated time if one exists. Otherwise a label with vertex $n+1$ is returned. *InsertLabel* inserts the newly generated label in *NPS* possibly overwriting an old label if it already exists.

1. $P_j = \{(T_j^k, C_j^k) \mid$ the path X_{0j}^k associated with (T_j^k, C_j^k) is feasible$\}$. The paths associated with labels in P_j are still feasible paths in Z^1. Furthermore these paths remain non-dominated paths in the solution to Z^1.

2. $D_j = \{(T_j^k, C_j^k) \mid$ the path X_{0j}^k associated with (T_j^k, C_j^k) is removed$\}$. The paths associated with labels in D_j have at least one vertex in the optimal path from the solution to Z^0. As the vertices in the optimal path of the solution Z^0 are removed the paths associated with labels in D_j are no longer feasible.

   The labels of D_j will be replaced in the reoptimization phase. The labels in D_j are lower bounds on the values of their replacements because the labels of D_j dominate their eventual replacements.

In the reoptimization phase two new sets are computed.

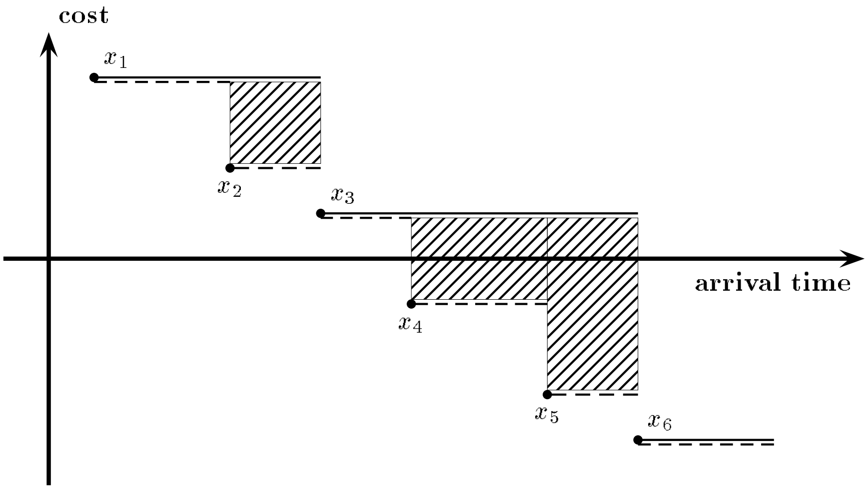


Figure 4.2: Solving problem Z^0 6 labels were generated at vertex j. After having removed the best path only 3 labels x_1, x_3 and x_6 are still feasible. The other labels were associated with paths that used at least one of the removed vertices. These labels are replaced in the reoptimization phase.

Consider figure 4.2. For a vertex j 3 (x_2, x_4 and x_5) out of previously 6 labels have been removed in the first phase. Now the new paths that are generated can only be extended to vertex j if their labels fall within the shaded area. If a new label is above a shaded area it is dominated by an existing label, and if it is below a shaded area it is not dominated and should also have been generated when we were solving problem Z^0.

The first set we calculate is the set of (non-dominated) labels that fall in the shaded area and are generated by finding all paths X_{pi} for which the extension by arc (i, j) produces a path X_{pj} with a label lying in the shaded area. This set is referred to as R_j.

The second set is obtained by calculating feasible solutions located in the identified zone. Feasible solutions are calculated by finding all feasible paths X_{pi} whose extension by an arc (i, j) produces a path X_{pj} whose label is placed within the shaded area. This set is denoted PR_j.

Now the replacement can take place. Start by finding new feasible labels ($\in PR_j$) to be added to P_j. Secondly add infeasible ($\notin PR_j$) but non-dominated labels ($\in R_j$) to D_j after eliminating the labels to be replaced. The change to D_j defines the new lower bound for future computations.

In our implementation we try to speed up computations by adding more than one route (path) to the set partitioning problem in each iteration. Kohl made a number of tests in [Koh95] indicating that it is worthwhile accepting more than one route to the set partitioning problem (as long as the reduced cost is negative, of course). Synergy effects between the routes typically lead to faster improvements and one reason for this is that several routes have one or more customers in common. Therefore we have not tried the reoptimization algorithm of Desrochers and Soumis in practice.

It should be noted that the SPPTW can be represented and solved as an ordinary shortest path problem in an enlarged graph by having one vertex for each pair of customer and feasible "time of service". Furthermore note that Spira and Pan in 1975 proved that the complexity of reoptimization is equal to the complexity of optimization for the shortest path problem.

4.2.2 Dominance criterion

In order to make the algorithm considerably more efficient we will (like in Dijkstra's algorithm) introduce a *dominance criterion*.

Assume that for a given vertex i we have two states (i, t_1, d_1) and (i, t_2, d_2) where $c(i, t_1, d_1) \leq c(i, t_2, d_2)$, $t_1 \leq t_2$ and $d_1 \leq d_2$. Clearly as long as the extensions based on (i, t_2, d_2) are valid the extensions based on (i, t_1, d_1) are also valid, and these will always be lower in cost (or at least not higher). Therefore the label (i, t_2, d_2) can be discarded. Formally we say that (i, t_1, d_1) *dominates* (i, t_2, d_2) (or $(i, t_1, d_1) \prec (j, t_2, d_2)$) if and only if all of the following three conditions hold:

1. $c(i, t_1, d_1) \leq c(i, t_2, d_2)$
2. $t_1 \leq t_2$
3. $d_1 \leq d_2$.

Each time a new label is generated we have to check with the other labels at the same vertex to see if the new label is dominated by some label or the new label dominates another label. This can be implemented quite effectively as will be describe later (see section 4.3).

We have not done any thorough experiments to measure the impact of the dominance criterion, but is seems to give a considerable speedup and also the amount of memory needed is much smaller.

Note that if a dominated label ℓ is not discarded, the labels generated based on ℓ will also be dominated (by the labels generated on the basis of the label that dominated ℓ). Therefore the label ℓ and labels generated on the basis of ℓ can never be part of the shortest path.

For the vertex corresponding to the depot $n+1$ the dominance relation is only dependent on the accumulated cost of the path/route, that is, label $(n+1, t_1, d_1)$ dominates label $(n+1, t_2, d_2)$ if and only if $c(n+1, t_1, d_1) < c(n+1, t_2, d_2)$.

4.2.3 Elimination of two-cycles

As we have relaxed the original constraints and thereby allowed for cycles one will typically find a lot of path cycling between two "good" vertices. These paths can be detected (and therefore avoided) by a method developed by Houck et al. and described in [Koh95].

In order to describe the algorithm, the labels are extended with on additional field *pred*, which denotes the predecessor of the vertex of the label. In addition to the extra field each label has a **type** being either *strongly dominant*, *semi-strongly dominant* or *weakly dominant*.

A label $(i, t, d, pred)$ is denoted *strongly dominant* if it is not dominated by any other label and at least one of the following conditions are satisfied:

1. $t + t_{i,pred} > b_{pred}$
2. $d + d_{pred} > q$.

This implies that a strongly dominant label can not participate in a two-cycle due to either time or capacity constraints (or both).

The label is called *semi-strongly dominant* if it is not dominated by any other label and none of the conditions

1. $t + t_{i,pred} > b_{pred}$
2. $d + d_{pred} > q$

are satisfied, which implies that a semi-strongly dominant label has the potential of being part of a two-cycle.

Finally, we call a label *weakly dominant* if it is only dominated by semi-strongly dominant labels, and the semi-strongly dominant labels have the same predecessor, and their common predecessor is different from the predecessor of the weakly dominant label.

It should be noted that the two-cycle elimination scheme does *not* change the computational complexity of the SPPTWCC algorithm. Let a state (i, t, d) be given, then exactly one of the following three cases is true:

1. There is no label for this state.
2. There is one strongly dominant label for this state.
3. There is one semi-strongly dominant label and **at most** one weakly dominant label for this state.

So the total number of labels is growing at most by a factor 2, thereby retaining the computational complexity.

As semi-strongly dominant labels can be part of a two-cycle, they are not permitted to be extended back to the predecessor. Instead we allow for the existence of weakly dominant labels. They are dominated by semi-strongly dominant labels **and** they can be extended "back" to the predecessor of the semi-strongly dominant labels. The weakly dominant label will only be extended to the predecessor of a semi-strongly label as it is dominated by a semi-strongly dominant label for all other possible extensions.

If a new label is dominated by an old label it can be discarded if:

1. The old label is not semi-strongly dominant. If the old label is not semi-strongly dominant it is either strongly dominant or weakly dominant. In both cases dominated labels are not allow, therefore the new label can be discarded.
2. The old label is semi-strongly dominant **and**
   (a) the old and the new label have the same predecessor.
   (b) the new label is dominated by two or more labels with different predecessors.
   (c) the new label can not be extended to the predecessor of the old label.

Case (2a) should be obvious. The reason for keeping weakly dominated labels is to be able to extend a path from the given vertex to the predecessor of a semi-strongly dominant label (in order to avoid 2-cycles it is not allowed to extend the path associated with the semi-strongly dominant label back to its predecessor). But if labels with different predecessors exist we do not need the weakly dominant label to be able to get back to a predecessor, and therefore the new label can be discarded in case (2b). If the new label cannot be extended back to the predecessor of the old label there is no reason for keeping it, as it will be dominated on all other extension, which proves cases (2c).

The same rules can also be applied if a new label is dominating an old label. Additionally a dominated label that is not discarded can change type to weakly dominant.

4.3 Implementation issues

A few question of a more technical nature are still open. These implementation issues will be discussed in this section.

4.3.1 Implementation of dominance

In order to allow for an effective check of dominance we maintain a list of labels for each vertex. These labels are sorted lexicographically according to arrival time, accumulated demand and accumulated cost. The list of labels at the same vertex is scanned checking whether the label we want to insert into the list is dominated. This process terminates as the lexicographically right position in the list is reached. If the new label is dominated it might either be discarded straight away (for example if the new label is dominated by a weakly dominated label) or "marked" as being dominated if it is possible to keep it as a weakly dominated label according to the dominance rules.

If the label is inserted in the list the remaining part of the list can not dominate the label, but instead the newly inserted label might dominate one or more of the remaining ones.

4.3.2 Generalized buckets

The procedure *BestLabel* returns the label with smallest accumulated time. Instead of just returning one label it is possible to return more labels that all can be processed independently. This is the idea realized with the *generalized bucket* datastructure.

In datastructures a "bucket" is typically a list of elements all within some pre-specified interval.

Now let $t_{\min}$ be the minimal "time-length" of any arc, that is, $t_{\min} = \min_{(i,j)}\{t_{ij}\}$. So whenever a label is extended, the time of the new label is at least increased by $t_{\min}$. If the new label is extended from a label with the presently smallest time t', it can not dominate any of the labels in the interval $[t', t' + t_{\min}[$. Therefore labels within this interval can not be deleted. So when *BestLabel* returns the currently best label it can also return the labels having an accumulated time within the mentioned interval. And all these labels can be processed without doing consecutive calls of *BestLabel*.

This can be accomplished in two ways. Either by a dynamic or a static approach.

In the dynamic approach, we keep track of the currently best label with respect to time (let us call its time t). Additionally we have a "pool" of labels that can be processed. This pool is initiated with the label $(0, 0, 0)$. When the pool is empty, the pool is refilled with labels with a "time stamp" within the interval $[t, t + t_{\min}[$. Hereafter all these labels can be processed etc.

Another way is chopping up the interval $[a_0, b_{n+1}]$ in smaller intervals of size $t_{\min}$ (starting with $[a_0, a_0 + t_{\min}[$) as depicted in figure 4.3. The labels are then inserted in the corresponding interval as they are generated. Due to the fact that the accumulated time of the newly generated label is a least $t_{\min}$ higher, the newly generated label will be inserted in a bucket succeeding the bucket of the label it was expanded from.

Here the processing simply starts from the first bucket containing $(0, 0, 0)$, and proceeds to a new bucket as soon as all the labels in the current bucket have been processed. The order in which the labels in a bucket is processed is unimportant.

For this implementation of the SPPTWCC algorithm the last "static" variant of the general bucket structure is chosen because it is the simplest approach. Regarding the running time of the two approaches they seem equally fast. The final algorithm is described in figure 4.4.

4.3.3 Miscellaneous remarks

Instead of just maintaining one label at vertex $n+1$ it may be advantageous to keep more than one. Often several routes with negative reduced costs will be found and can be returned and entered into the master problem. This does in practice speed up the column generation process.

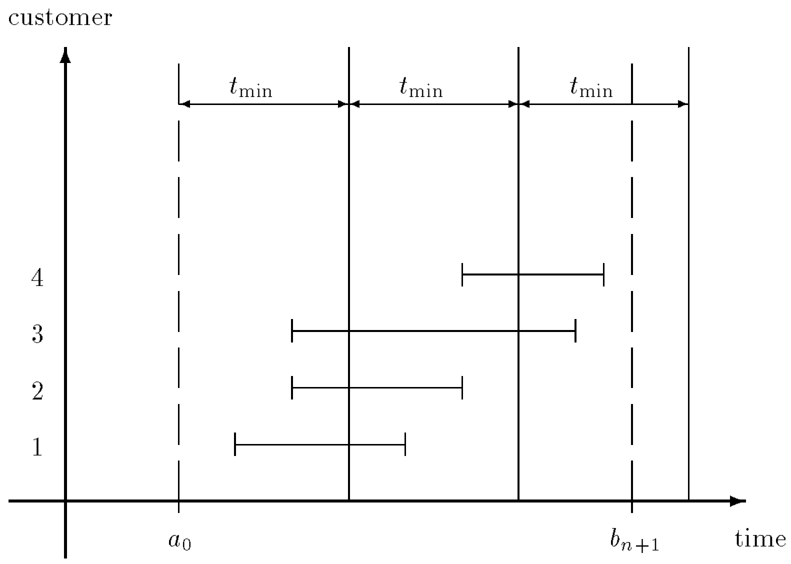


Figure 4.3: Division of the time span in smaller intervals and the time windows of the customers.

```
⟨ Initialization ⟩
t_min = min_(i,j) {t_ij}
Generate (b_{n+1} − a_0)/t_min buckets in structure Bucket
Bucket[0]= {(0, 0, 0)}
c(0, 0, 0) = 0
CurrBucket=0

while (CurrBucket ≤ ((b_{n+1} − a_0)/t_min))
    ⟨ Adjust CurrBucket to the next non-empty bucket ⟩
    while (Bucket[CurrBucket]=∅)
        CurrBucket++

    (i, t, d) = NextElement(Bucket[CurrBucket])

    for j := 1 to n + 1 do
        if (i ≠ j ∧ t + t_ij ≤ b_j ∧ d + d_j ≤ q) then
            ⟨ Label feasible ⟩
            if c(j, max{t + t_ij, a_j}, d + d_j) > c(i, t, d) + ĉ_ij then
                ⟨ New label better ⟩
                t' = max{t + t_ij, a_j}
                InsertLabel(Bucket[⌈(t' − a_0)/t_min⌉], (j, t', d + d_j))
                c(j, t', d + d_j) = c(i, t, d) + ĉ_ij

return
```

Figure 4.4: The algorithm for finding the shortest path with time windows and capacity constraints with the generalized buckets. *NextElement* returns a label from the specified bucket.

Chapter 5

Generalizations of the VRPTW model

A number of additional constraints or properties of more complex routing problems can be modelled using the framework just developed. In this section we will briefly discuss how to allow non-identical vehicles, work with more than one depot, multi-compartment vehicles, and the use of multiple time windows or soft time windows.

5.1 Non-identical vehicles

Vehicles can be non-identical in several ways. The typical way a heterogeneous fleet of vehicles is characterized is by the capacity, but it could also be different due to different arc costs for each vehicle, different travel times, time windows or other characteristics.

The fleet is made up of several groups of vehicles, where all the vehicles in a group are treated as identical. For each group of vehicles a SPPTWCC problem with different vehicle capacity, arc cost, travel times etc. has to be solved. If there exists an upper or lower limit on the number of vehicles available for each group this can be modelled in the master problem. Between each call of the master problem one can choose to run one or more SPPTWCC's. If the only difference between the groups are on vehicle capacity it is not necessary to solve one SPPTWCC for each group. Instead the underlying graph can be extended and only one SPPTWCC has to be solved. The extension is described in [Hal92, Koh95].

5.2 Multiple depots

In real-life problems there might be more than one depot. The VRPTW can be used to model situations where multiple depots exist. The customers are serviced by several depots, each depot having their own fleet of vehicles. Usually one assumes that vehicles must return to the same depot as they started from. In a relaxed form we only demand that the number of vehicles arriving at the depot equals the number of vehicles leaving it. In a further relaxation seldom used there are no constraints on which depots the vehicles should return to.

The Multi Depot VRPTW (MDVRPTW) has only recently be the focus of attention. In our column generation-based approach we can solve the MDVRPTW using the same master problem, as each customer still must be serviced by at least one vehicle. The modifications can be isolated to the SPPTWCC.

If the vehicles are based at different depots, that is, the vehicles must return to the depot they started from, one SPPTWCC must be solved for each depot. Bounds on the availability of vehicles are handled in the master problem, where we need one set of upper/lower constraints for each depot.

If we relax the problem and just want the number of vehicles that arrives at a depot to the same at the number of vehicles that leaves the depot it is only necessary to solve one SPPTWCC. As figure 5.1 demonstrate we introduce one "super leaving-depot" (SLD) and one "super arriving-depot" (SAD). A route now starts at the SLD continues to an ordinary depot then to the customers. The route ends by visiting an ordinary depot and the finally the (SAD). For each depot there is a constraint in the master problem ensuring that the number of vehicles arriving equals the number of vehicles leaving.

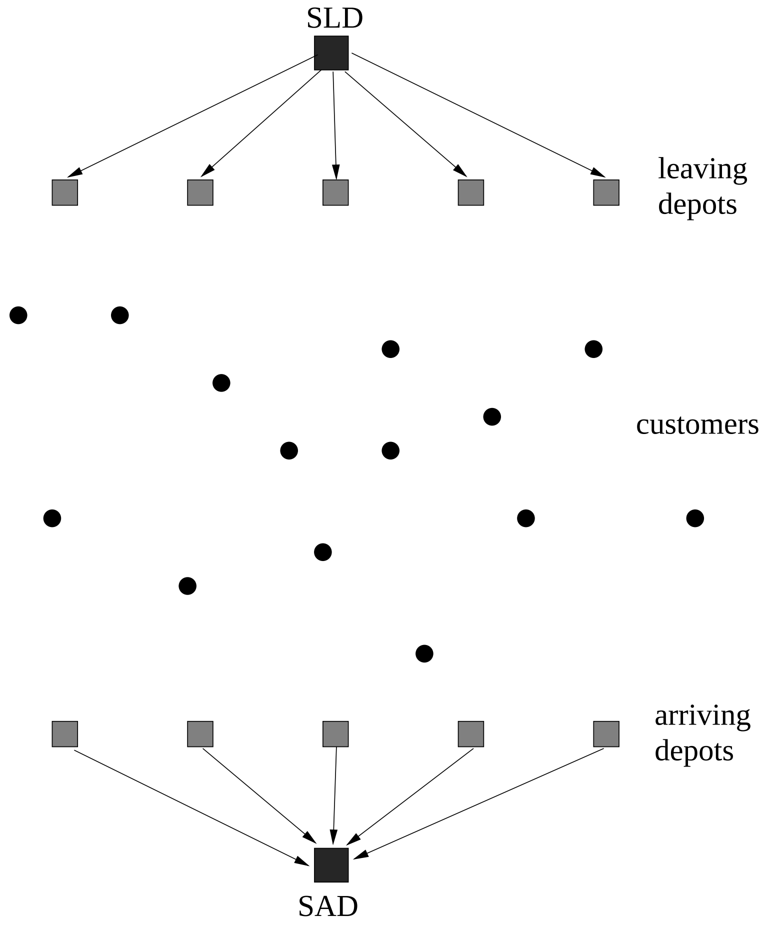


Figure 5.1: The underlying graph is extended by the SLD and the SAD. Now every route starts at the SLD and ends at the SAD.

The corresponding simplex multipliers modifies the cost of arcs originating in the depot and arcs terminating in the depot (with opposite sign).

5.3 Multiple Compartments

If the vehicles have two or more compartments the routing problem is known as a Multiple Compartment VRPTW (MCVRPTW). The use of multiple compartments is relevant, when the vehicles transports several commodities which must remain separated during transportation. An example is the distribution of oil products to service stations where the tank trucks are divided into a number of compartments in order to transport the different kinds of petrol.

The multiple compartments have no influence on the master problem. The overall structure of the SPPTWCC is not changed either. The multiple compartments are modelled by extending the number of states required. In the SPPTWCC the labels have four states: the label, accumulated cost, accumulated time and accumulated demand. If each vehicle has c compartments the capacity constraints must be modelled by c states instead of one. The general principle is the still the same but now we have to have one state for each compartment in order to keep track of the accumulated demand for each compartment. In this way we end up with a SPPTWMCC (Shortest Path Problem with Time Windows and Multiple Capacity Constraints).

In the same way we can extend the VRPTW model in order to handle multi-dimensional capacity constraints. In VRPTW the capacity is one dimensional. This dimension can be the weight, volume, value or pieces. However, the capacity constraints can be multi dimensional, for instance weight and volume in order to be able to handle cases where many large boxes do not violate the weight constraint but their volume is to large for one vehicle, or the other way around.

5.4 Multiple Time Windows

In the VRPTW each customer has one time window where the service must take place. Allowing customers to have multiple (and disjoint) time windows in which they can be serviced can be handled straightforward in the SPPTWCC. A vehicle that arrives between two time windows must wait until the beginning of the next time window. Note that preprocessing, dominance criterion etc. remains valid.

5.5 Soft Time Windows

Sometimes a cost $p(s_i)$ depending on the service time s_i of a customer i is introduced in order to penalize arrivals that are feasible but undesirable. The time window is then said to be *soft*. If the cost is non-decreasing with time, i.e. $s_i^1 \leq s_i^2 \Rightarrow p(s_i^1) \leq p(s_i^2)$ the dominance criterion remains valid and the soft time windows can be incorporated in our VRPTW. The case where the penalty function $p(\cdot)$ is a general function is not efficiently solvable.

5.6 Pick-up and delivery

In VRPTW we either pick-up goods at the customers or goods are delivered to the customers. In pick-up and delivery the vehicles can perform both tasks. In the simple *backhauling* version (sometime denoted VRPBTW – the Vehicle Routing Problem with Backhauling and Time Windows) of pick-up and delivery the vehicles must be completely empty before the pick-up phase starts.

In this simple case the customers can be divided into two classes of customers: a set of delivery customers and a set of pick-up customers. Now by removing all arcs from pick-up customers to delivery customers we ensure that it is not possible to service a delivery customer after a pick-up customer. Two capacity labels, one for delivery and one for pick-up, are handled in the SPPTWCC.

In [GDDS95] Gélinas et al. demonstrates that only one capacity resource is sufficient. Now let q_i be the load of the vehicle if customer i is a delivery customer. If customer i is a pick-up customer q_i denotes the load of the vehicle before customer i is visited. Let d_j be the amount of goods to be delivered to customer j, if (i, j) is an arc between two delivery customers the quantity of goods to be delivered to customer j is placed on arc (i, j), that is, $q_{ij} = d_j$. For an arc between two pick-up customers the amount to be picked up at customer i is placed in arc (i, j), that is, $q_{ij} = p_i$, where p_i is the amount to be picked up at customer i.

As all deliveries have been accomplished the load of the vehicle is reset to zero. This is done by setting $q_{ij} = -q$ on every arc going from delivery customers to pick-up customers. So going from a delivery customer to a pick-up customer we get:

$$Q_j = \max\{Q_i + q_{ij}, 0\} = \max\{Q_i - q, 0\} = 0,$$

as capacity is not allowed to become negative.

In the more general backhauling problem, where customers, pick-up or delivery, may be serviced in any order requires two resources to ensure that the capacity constraints of the vehicles are satisfied.

As previously q denotes the capacity of the vehicles. The use of resource $k, k \in \{1, 2\}$ on arc (i, j) is called r_{ij}^k. As a customer is either a pick-up customer or a delivery customer at least one of r_{ij}^1 and r_{ij}^2 are 0. If j is a pick-up customer r_{ij}^1 will denote the amount to be picked up (for all i). If j is a delivery customer then r_{ij}^2 is equal to the amount to be delivered. A label in the shortest path subproblem consists of 4 states (i, t, r^1, r^2), where i is the customer, t the accumulated time and r^1 and r^2 are the accumulated pick-ups and deliveries. The accumulated cost of the label is $c(i, t, r^1, r^2)$.

The dominance criterion remains the same (although extended by one label), but the updating becomes more complicated. Now let label (i, t, r^1, r^2) be extended to vertex j. The new label will then be

$$(j, t + t_{ij}, r^1 + r_{ij}^1, \max\{r^2 + r_{ij}^2, r^1 + r_{ij}^1\}).$$

The label for the quantity picked up is updated as usual. For the delivery label the updating formula becomes more complex. Note that the label is never allowed to become smaller than r^1, as the vehicle is maximally loaded, when the difference between the two resource labels are maximal. A positive difference means that there is less available for deliveries. This is controlled by increasing r^2 up to r^1.

5.7 Additional constraints

A large number of extra "minor" constraints from real-life applications can be incorporated without problems.

An upper or lower limit on the length (in cost or time) of the routes can be modelled using additional resources. Using additional resources also makes it possible to set a limit on the number of customers that can be serviced. It is also possible to only allow specific vehicles to use certain arcs or service certain customers.

Note that it is also possible to introduce time-dependent travel speed for example in order to take account of rush hour traffic. If time-dependent travel speeds are introduced it is not guaranteed that the triangle inequality holds.

Chapter 6

Parallel Implementation

In this chapter the efforts undertaken and the results thereof with respect to developing a parallel algorithm based upon the sequential Branch-and-Price code described in chapter 3 are described. First, a short introduction to the field of parallel computing is presented, and then the description and analysis of making the parallel program follows.

6.1 Parallel Programming

Parallel computing enjoys significant interest both in the scientific community and industrially due to an increasing demand for computational power and speed and the decreasing cost/performance ratio for parallel computers.

Alas, the new possibilities do not come without a price: parallel programming is inherently more difficult.

In contrast to sequential programming where almost everyone can agree on how a sequential computer is viewed, the world of parallel programming is not so clear-cut. The sequential paradigm views the computer as a single processor which can access a certain amount of memory. This paradigm is in the literature also called *RAM* – *R*andom *A*ccess *M*achine. The RAM is equipped with an instruction set that includes basic operations for writing to and reading from the memory and a number of arithmetic and logic operations (depicted in figure 6.1(a)). The success of this model is due to its simplicity and the great similarity with real-life von Neumann-type computers.

Even though modern computers allow more than one user at the same time on the same machine, the view of the programmer is still the same; he/she views the computer as a single-user machine.

In parallel programming the situation is unfortunately not that simple; here there does not exist a commonly accepted model (see for example [vD96]).

There exists theoretical models, but the theoretical models and the real-world parallel computers are very far apart from each other. The most general model is the *PRAM* - *P*arallel *R*andom *A*ccess *M*achine. This model is purely theoretical and consists of an unlimited number of processors all directly connected to one shared memory (which consists of an unlimited number of memory cells). Sometimes a local memory is added to each processor. The processors communicate with each other through the shared memory, which can be accessed in unit time, that is, just as fast as e.g. a multiplication operation. This model is often used for evaluating parallel algorithms with respect to their asymptotic time-complexity.

In [KLK89] Kindervater et al. gives three reasons why parallel computers have not broken through yet. The main obstacle is the lack of uniformity in available architecture as a number of new aspects are introduced by parallelism. As mentioned below this obstacle has now been removed to a large extent.

The remaining two obstacles are that more realism will be required in theoretical models of parallel computation. We need models that include factors like for example location of memory and processor topology (some current efforts are *LogP* [CKP+96] and the *Distributed Memory Model (DMM)* described in [MDF+95]), and finally more formal techniques for the design and implementation of efficient parallel algorithms are needed.

The performance and usability of the different parallel models depend on factors like computational concurrency, processor topology, communication, location of memory (local or shared), scheduling and synchronization. These factors also highly influence which kind of algorithms/programs that are suitable for a specific parallel computer.

A general introduction to parallel programming with a theoretical approach can be found in [JáJá92], while a more practical approach can be seen in [Fos94, CT96].

The different qualities of the parallel computers have led to several different classification schemes for parallel computers, where the *Flynn's taxonomy* [Dun90, KL86] must be regarded as the most widely used. In Flynn's taxonomy computers are divided into four groups:

SISD or *Single Instruction-stream Single Data-stream* computers: This is the normal stand-alone sequential computer also commonly known as the *von Neumann* architecture. This type is depicted in figure 6.1(a).

SIMD or *Single Instruction-stream Multiple Data-stream* computers: This type of computers are sometimes also denoted *vector computers*. An array of (normally fairly simple) processors are controlled by an external host via a controller. Here only one instruction at a time is performed, but on different data-elements. Either a processor performs the instruction given or it sits idle. This type of computers normally operates on problem instances where $p \in \Omega(n)$. As this type of computer can be emulated by the more general MIMD class computer (see below) these computers are more often seen as special components instead of independent computers.

Real-life examples are the CM-2 (Connection Machine 2), the MasPar and the CPP DAP Gamma.

MIMD or *Multiple Instruction-stream Multiple Data-stream* computers: The MIMD is the most versatile architecture. Here the operation performed by each processor is independent of each other.

This class can be subdivided according to the character of the memory:

Shared Memory This subclass is often also called *multiprocessors*. All processors share the same memory space. As in the PRAM model, communication between the processors are carried out via the memory. It can easily be seen that the memory access quickly becomes a sever bottleneck of the system as the number of processors are increased. Therefore some shared memory systems are a hybrid between this subclass and the distributed memory subclass (see below) as the memory is physically distributed (as in the distributed memory subclass) but hardware emulates a globally addressable memory space (as in this subclass). This type of parallel computer is sometimes called *virtual shared memory* and is normally placed in the shared memory subclass as this is the way it acts seen from the programmers view (depicted in figure 6.1(b)).

An example of the virtual shared memory is the KSR-1 (there does also exists software libraries like TreadMarks [ACD+96] that simulates shared memory on a distributed memory computer). Examples of "real" shared memory machines includes Cray J90- and T90-series, Convex C4 series and the Silicon Graphics Power Challenge.

Distributed Memory This subclass is sometimes also referred to as *multicomputers*. Processors have their own local memory. Accessing the memory of another processor can therefore only be done by *message-passing* (depicted in figure 6.1(c)). Examples here include Cray T3E, the Paragon, the Meiko CS-1 and CS-2, and the Parsytec.

A network of workstations could also be classified as a MIMD Distributed Memory architecture. In order to distinguish them from "dedicated" parallel computer systems with a very fast highly specialized network the network of workstations is usually referred to as *loosely coupled* and the "dedicated" parallel computer as *tightly coupled*.

The physical topology of the network used to be a major issue when designing parallel algorithms. Processors that had to communicate during the execution of a parallel algorithm should preferably be neighbouring processors in order to obtain good performance. The programmer had to remember how the processors physically were connected, and spend time considering how to setup the parallel program. But in systems of today almost any topology can virtually be imposed, and the performance difference between exploiting the physical topology and using the imposed virtual topology is, if not insignificant, then very small [Ant96]. This is mainly due to dedicated communication processors in today's parallel computers.

MISD or *Multiple Instruction-stream Single Data-stream* computers. Here several different kinds of action are performed on the same data. This class is usually included for completeness only as no computer of this type has ever been built.

A comprehensive overview of real-life parallel computers can be found in [vD96, DS96].

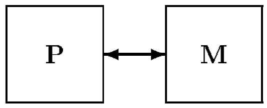
As stated earlier the different set of communication procedures, setup commands etc. for each parallel computer used to be a major obstacle. This made it very costly to port a program from one parallel computer to another even if the underlying architecture was identical. This impediment has to a great extent been removed by different interfaces using the message passing paradigm, where the different processors exchange information by sending messages around the communication network.

Among a number of different interfaces two have emerged as de facto standards: MPI (*Message-Passing Interface*) and PVM (*Parallel Virtual Machine*). Efforts to merge them are described in [FD96]. Presently work is in progress to melt the two standards into one (codename: "PVMPI"). We have used MPI ([ABMS94, Mal96, MMHB, Pac95]) in our efforts to parallelize the sequential code.

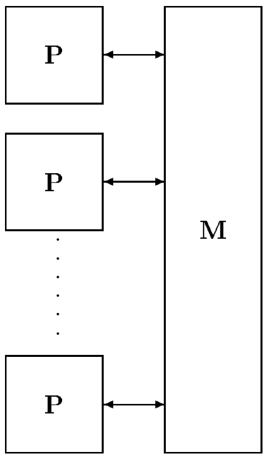
6.2 The IBM SP2 computer at UNI•C.

The parallel experiments conducted during this project were all carried out on the IBM SP2 computer owned by UNI•C[1], the Danish center for high-performance computing.

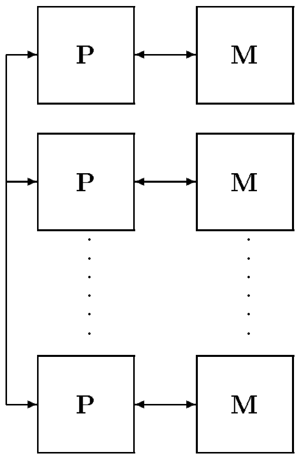
[1]Homepage: *www.uni-c.dk*.



(a) Picture of a sequential computer where P is the processor and M is the memory



(b) The Shared Memory model of a parallel computer.



(c) The Distributed Memory model of a parallel computer.

Figure 6.1: Diagram of the three computer models from Flynn's taxonomy that exists in the real world. Here **P** refers to a processor, while **M** is an independent block of memory.

Using parts originally designed for the RS/6000 workstations IBM developed the *RS/6000 Scalable Powerparallel System*, or just SP2 parallel computer. It is a full-fleshed MPP (Massively Parallel Processing) computer having the memory distributed among the processors, making the SP2 resemble a network of workstations. The SP2 starts with 2 nodes, expandable to 512. The SP2 at UNI•C (called "jensen") consists of 88 nodes in 4 main categories:

- **4 nodes for interactive use:**
  66 MHz Power2 CPU, 512 Mb RAM.
- **12 nodes for test jobs:**
  160 MHz P2SC CPU, 512 Mb.
- **64 nodes for parallel batch:**
  120 MHz P2SC CPU, 33 of which have 512 Mb RAM the rest are equipped with 256 Mb RAM.
- **20 nodes for serial batch:**
  10 135 MHz P2SC CPU, 2 Gb RAM.
  10 160 MHz P2SC CPU, 1 Gb RAM.

All running with a 1:1 ratio of processor speed vs. memory speed. Note that the memory of the 64 processors in the parallel batch adds up to 32 Gb. We might be able to speed up the computations if it is possible to get several processors to solve the problem collectively, but we may also be able to solve larger problems as we have a total of 32 Gb of memory at our disposal.

All nodes are interconnected by a very fast *High Performance Switch*. The 4 interactive SP nodes also act as fileservers for the 64 Gb user disk area.

Each node is equipped with a 128Kb data cache and a 32Kb instruction cache. The data cache is four-way set associative.

An SP2 system contains two main building blocks: nodes and switchboards. A switchboard consists of 32 communication chips (hereof 16 are "shadow-chips" for fault recovery). Chips work in pairs (one handling input one handling output). These pairs are then wired to construct a four way by four way cross-bar implementing a 16 bidirectional multistage interconnection network. This switchboard together with Power2 processors constitute the "SP2 frame" – a unit with at most 16 processors and their memory and the switchboard for intra- and inter-connection within the SP2. In systems with more than 80 nodes, intermediate switchboards are required.

Messages can be sent between two processors either using Ethernet or the high performance switch. The high performance switch can be used with two different protocols, Internet protocol and User Space protocol. In our case we have used the latter as it is according to UNI•C considerable faster.

When sending small messages the communication time is approximately linear in the size of the message. The communication time depends on two parameters: τ_S is the *start-up latency* and τ_B is the *bandwidth*. The start-up latency is the time it takes to set up the connection, while the bandwidth is the rate at which data can be transferred when the connection is established. Using the high performance switch with the User Space protocol the values are given in table 6.1

Message size		**Latency**	**Bandwidth**
0 – 4096	bytes	53 μs	53.1 Mb/s
4096 – 55900	bytes	118 μs	67.8 Mb/s
55900 –	bytes	363 μs	96.5 Mb/s

Table 6.1: Communication speed for the high performance switch using the User Space protocol.

The processors used for parallel batch jobs each have a peak performance of 480 Mflops.

Running batch jobs is carried out by the *LoadLeveler*, that maintains four different queues: tiny (max. 4 processors), small (max. 8 processors), medium (max. 16 processors) and large (max. 32 processors). Using more than 32 processors requires a special arrangement with UNI•C. When running production runs of programs on the SP2 the processors allocated to the program are not shared with other programs or users.

6.3 A parallel VRPTW algorithm

Our Branch-and-Price model contains roughly four main parts:

1. Solving the subproblem (SPPTWCC).
2. Solving the master problem (relaxed set partitioning).
3. Branching.
4. Bounding.

We will examine each of these components and try to estimate the potential of parallelization.

As we focus on the MIMD architecture with distributed memory this will be the basis of our analysis.

6.3.1 Solving the subproblem (SPPTWCC)

Parallelism can be introduced by trying to parallelize the SPPTWCC subroutine. Indeed, some simple parallel schemes are fairly obvious. For example an initial pool of labels is generated on one processor, then the pool is distributed among all processors and thereafter each processor can run independently on the basis of the assigned labels. This procedure stops when all processors have finished generating labels on the basis of the initial labels assigned. Another idea would be to stop when a number of routes have been generated and sent to the master. Note that communication is needed in order to resolve dominance between labels on different processors. Running without dominance checks between processors would lead to a large increase in the number of labels, potentially "drowning" the processors in work.

We are not guaranteed to receive the best routes from the subproblem, but the main point is that the routes the SPPTWCC subroutine is returns are "good" ones (i.e. routes with negative reduced costs).

The idea results in an algorithm depicted in figure 6.2. Note that the only place where parallelism is introduced is the SPPTWCC subroutine, as a designated master processors executes the remaining tasks (branching, solving the master problem etc.).

$\langle$ run the SPPTWCC code until k labels are generated $\rangle$
$\langle$ distribute $\frac{k}{p-1}$ to each of the $p-1$ slave processors $\rangle$
while ($\langle$ stopping criterion not meet $\rangle$)
    for each $\langle$ slave processor $\rangle$ **do**
        $\langle$ run the SPPTWCC code on the basis of the labels $\rangle$
        return $\langle$ generated routes to the master $\rangle$

Figure 6.2: Parallelization of the SPPTWCC code. The stopping criterion may be a simple one stopping when all slaves are finished, or a more complex for example stop all slaves when a certain number of routes are generated.

Even now with a possible design of a parallel SPPTWCC subroutine the question is whether it is worthwhile implementing. Preliminary test runs of our sequential VRPTW algorithm suggests that this is not the case. We have collected information on a sample of runs of our sequential VRPTW algorithm in table 6.2. The high percentage of running time spent in the SPPTWCC subroutine favours the parallelization but two other observations indicates performance problems.

As one can see the running time spent in the SPPTWCC subroutine is not spent in one or two calls but in many calls, which results in low average time spent in one call of the SPPTWCC subroutine. And even though some calls of the SPPTWCC subroutine takes longer time than others, the average time spend in the SPPTWCC subroutine does not exceed 15 seconds. Only in 2 out of the instances presented in table 6.2 is the average running time of the SPPTWCC subroutine above 0.5 seconds. This is clearly not enough to parallelize on a MIMD machine. As a considerable amount of time is used for setup and initialization before actually doing any “useful” work it is important that the time not used for “useful” work can be regained later. With these small running times these gains can not be achieved. We will therefore not consider this idea further.

Instance	Customers	Overall time	Time in SPPTWCC	No. of Calls	Average time	largest time
R101	100	20.60	2.19	88	0.025	0.08
R102	100	111.32	100.22	46	2.179	25.82
R105	100	331.78	26.70	369	0.0724	1.03
C103	100	1090.25	1025.05	89	11.517	224.25
C105	100	65.21	8.67	87	0.097	0.59
C107	100	53.26	7.73	74	0.131	0.59
C108	100	59.11	18.28	68	0.290	2.23
RC101	100	46.97	5.47	94	0.058	0.27
RC105	100	166.56	34.28	211	0.162	2.32

Table 6.2: Time used running in the SPPTWCC and the total number of calls for a selected number of Solomon problems.

6.3.2 Solving the master problem

The master problem is a set partitioning problem which is relaxed and solved by an LP-solver. The topic of parallelizing the computation of an LP-solver is ongoing research. As far as the author knows there is still a long way to efficient parallel LP-solvers, so we will focus on parallelizing the other part of the algorithm. This is also a topic outside the scope of the Ph.D. project.

6.3.3 Branching and bounding

Branch-and-Bound is often computationally very intensive. There are two basic ways of reducing the execution time of Branch-and-Bound:

- Improve the effectiveness of the bounding rule, and
- Parallelize the computation.

While the first item requires knowledge of the problem to be solved, the second requires skills in parallel programming.

There exist a number of papers on parallel Branch-and-Bound [PL90, GKP, Cla96, Cla97, LRT95, Rou96, GC94, TP96]. As branching and bounding are dependent on each other they will be analyzed together in this section.

In [GC94] the authors distinguish between three kinds of parallelism. In *parallelism of type 1* the parallelism is introduced on the generated subproblems. The parallelism discussed in section 6.3.1 and 6.3.2 is an example of parallelism of type 1. Note that this type of parallelism has no influence on the general structure of the Branch-and-Bound tree and that the parallelism therefore is dedicated to the problem solved.

Solving more subproblems simultaneously is *parallelism of type 2*. Here the processing of the Branch-and-Bound tree is reflected by the decisions made during analysis and design of the parallel algorithm.

Finally *parallelism of type 3* consists of building more than one Branch-and-Bound tree in parallel, thereby using different operations (for example different branching operations or bounding functions) in each tree. The global upper bound can still be shared among the Branch-and-Bound trees, and maybe, even other structural results can be exchanged during the computation. Here the aim is to take advantage of the strength of different operations.

The parallelism of type 2 has been subject of intensive research. This will also be the type of parallelism we introduce. Besides, many concepts and ideas discussed with respect to parallelism of type 2 is also applicable with respect to parallelism of type 3. Parallelism of type 1 is already discussed and will not be considered further.

Parallelism of type 2 is very well suited for implementation on MIMD structured parallel computers.

A further classification criteria seen in several surveys ([GC94, PL90, Rou96, Cla96]) is determined by the placement of the work pool. Generally these schemes are referred to as the *master-slave paradigm* and *distributed loading* (in [PL90] they are referred to as *central list* respectively *distributed list*, and in [Rou96] as *centralized* respectively *distributed*).

In the centralized master-slave paradigm one processor is designated as the master and the remaining processors are named “slaves”. This approach is the oldest and originates from the times of the old SIMD computers where a "normal" computer was acting as front-end of the system. This computer was also responsible for sending the commands to the processors of the system.

The master processor is responsible for maintaining the work pool, and the slave processors perform the processing of subspaces distributed to them by the master. In this way it is ensured that the slaves always work on subspaces that from a global point of view look "promising". Work loading is then not a problem, and also termination detection is easily solved. The principle is depicted in figure 6.3.

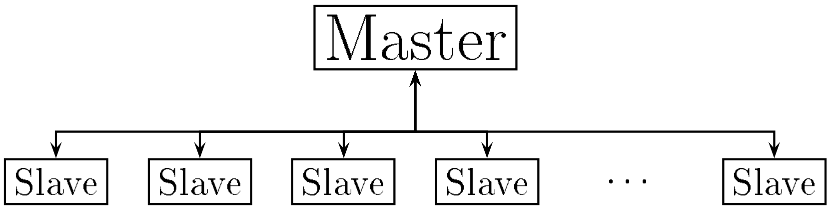


Figure 6.3: The Master processor is feeding the Slave processors with problems that needs to be solved. In the opposite direction the slaves send solutions or requests for more work to the master.

A major drawback of this approach however is the communication between the slaves and the master [TO89]. With increasing number of processors this quickly becomes a bottleneck [Cla90, Cla96]. This is documented in [dRT88] by de Bruin et al., where a series of experiments lead to the conclusion that the bottleneck appears between 8 to 16 processors, i.e. this approach has *scalability* problems. Scalability can be overcome to a certain extent by implementing several levels of master processors and "distribute" the global work pool. Instead of all slaves connected to one master, the processors are structured in a tree. This principle is shown in figure 6.4.

The approach can be improved by doing a series of Branch-and-Bound operations before communicating with the master. As the Branch-and-Bound

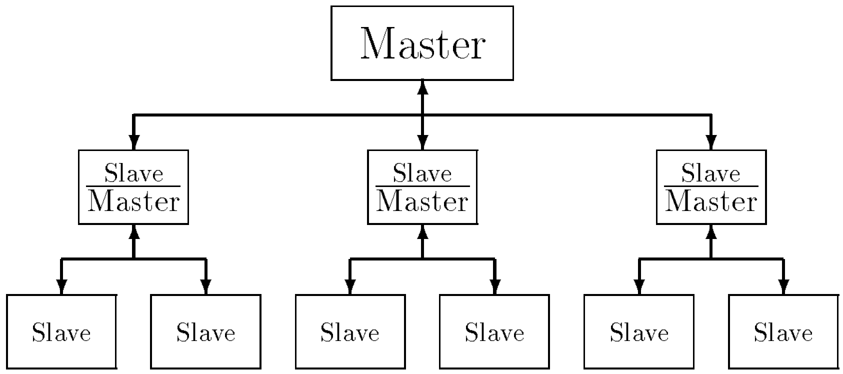


Figure 6.4: An example of a "distributed master" setup. 10 processors are placed in three layers. The six processors on the bottom level all work as slave-processors only, while the top processor only acts as a master-processor. The three processors in the middle layer acts both as slave-processor with respect to the master-processor in the top level, but also as a master-processor for the two slaves in the level below. As in figure 6.3 arrows pointing down symbolizes the transfer of unsolved Branch-and-Bound nodes and arrows pointing upwards is a indication of new global upper bounds and requests for more work flowing the other way.

operations are usually not equally fast this may distribute the communication better.

Our parallel algorithm will be implemented using the distributed loading approach. In the distributed approach all processors are "equal" after an initial distribution of unsolved Branch-and-Bound nodes.

In *static* load distribution (or static parallel Branch-and-Bound) the initial distribution is the only distribution of unsolved Branch-and-Bound nodes. Hereafter each processor only handles the Branch-and-Bound nodes from the initial distribution or the child nodes generated hereof. The advantage of this approach is its simplicity. This strategy has been investigated by Laursen in [Lau94b]. The approach looks to be quite effective, but it is very sensitive to the assignment of subspaces to processors in the initial distribution phase.

We have therefore implemented the *dynamic* load balancing approach. The processors are set up in some topology (possibly depending on the physical topology of the parallel computer), and work is then passed around between neighbouring processors. The strength of this approach is the scalability. On the other hand work load distribution and termination detection is not that simple any more. As subspaces are generated and dealt with independently on the different processors, the efficiency of the load balancing scheme becomes crucial with respect to efficiency for the implementation of the parallel Branch-and-Bound.

Two problems have to be addressed in order to obtain an efficient workload distribution algorithm:

- Idle time for each processor has to be minimized.
- Each processor is only allowed to spend a (small) fraction of time on the load distribution scheme compared to the time used working on the Branch-and-Bound-nodes.

Workload distribution is done with a *local* strategy in our algorithm, that is, the distribution only depends on the workload between two neighbouring processors. Each time information is passed along from one processor to another, be it a number of new unsolved Branch-and-Bound-nodes or a new global upper bound, information on the size of the work pool is also passed along. It is vital for this approach that the communication is effective in two ways: we do not wish to use too much time communicating vs. the time used on computation, and we prefer to send "promising" work around, in order to keep every processor busy working on "promising" subspaces. Among possible strategies proposed in the literature are:

1. Sending the most recently generated Branch-and-Bound nodes.
2. Sending the most recently generated Branch-and-Bound nodes with the lowest lower bounds.
3. Sending the Branch-and-Bound nodes with the lowest lower bounds.

4. Sending Branch-and-Bound nodes with "low" (not necessarily the lowest) lower bounds among all Branch-and-Bound nodes in the work pool.

5. All newly generated Branch-and-Bound nodes are sent to other processors.

As we do not have a designated master in the distributed approach the termination detection becomes more difficult than in the centralized approach. A number of different termination detection strategies have been suggested (see for example [Tel94, chapter 8]). We have implemented the termination detection algorithm by Dijkstra, Feijen and van Gasteren [DFvG83], which will be briefly described.

In the termination detection algorithm one processor (let's say processor 0) is responsible for detecting the stable state were all processors have terminated (i.e. are passive). Often processor 0 is denoted the "master", but note that it is not supervising the solution of subspaces as the master processor in the master-slave paradigm did. Each processor has an associated colour – it is either white or black. Additionally we have a token that also has a colour (again either white or black). Initially all processors are white, but a processor changes its colour to black as it transmits one or more messages, thereby indicating to the token that the processor has been active since it was previously visited by the token. Upon receiving the token an active processor keeps it until it becomes passive, and then it sends it along to the next processor in the ring. If the processor is black the token is coloured black before it is sent otherwise the token is passed on without changing the colour. As the token is passed on, the transmitting processor becomes white.

The detection scheme is initiated by processor 0. It transmits a white token to the next processor in the ring (thereby making itself white).

By this scheme a token returning to processor 0 still being white indicates that all processors are passive. If the token returns to processor 0 being black a new "probe" is initiated.

6.4 Implementational details

The sequential algorithm for the VRPTW described in chapter 3 and 4 is used as the starting point for the construction of the parallel algorithm. As stated earlier, MPI was chosen as the interface providing the communication framework of the message-passing scheme.

6.4.1 The initial phase

In the initial phase only the master processor is active. The remaining processors are waiting to get their initial set of unexplored subspaces. The master processor runs the sequential algorithm for the VRPTW. The process is stopped if the problem is solved (a signal is then broadcasted from the master to the remaining processors to stop work on all processors) or if $k \cdot p$ unexplored subspaces have been generated, where k is a pre-specified constant and p is the number of processors.

So in the interesting cases, k problems for each processor are generated. It should be fairly obvious that it is of paramount importance that the running time spent in this initial phase is kept as small as possible in order to get all processors working on the problem as quickly as possible. Still assuming that the problem is not solved the master processor broadcasts problem characteristics (number of nodes, the geographic position of the nodes, the time windows etc.) and generated cuts to the other processors. Hereafter each processor receives k unsolved problems from the master. This ends the initial phase.

6.4.2 The parallel phase

In this part of the execution of the algorithm all processors solve different Branch-and-Bound nodes. The master processor is in charge of termination detection and returning the result at the end of the execution of the algorithm. Otherwise there is no difference between the master and the remaining processors. Every processor is running a slightly modified version of the sequential algorithm for the VRPTW.

Parallelism introduces one major difference to the sequential algorithm for the VRPTW. Via the message-passing interface new unexplored Branch-and-Bound nodes arrive and new global upper bounds may also be passed along.

Subspaces with a value higher than the value of the global upper bound are denoted *passive*, while the remaining subspaces are called *active*. As the heap of unsolved subspaces during computation might contain subspaces that are not longer necessary (as their value is greater than or equal to the global upper bound) the number of nodes in the heap is not giving a correct picture of the workload on each processor. This can be dealt with in several ways:

1. Do nothing. We take care not to send passive subspaces to other processors but otherwise we only adjust the number of nodes when the heap no longer contains active subspaces (in this case the heap is emptied).
2. Every time the global upper bound gets updated some subspaces in the heap might change type from active to passive. Therefore the number of passive subspaces is counted and subtracted from the total number of subspaces in the heap. This will give an accurate measure of the workload of the heap, as no new passive subspaces will be put on the heap.
3. As an extension of the previous idea we can actually delete passive subspaces whenever the global upper bound is lowered. Hereafter the total number of subspaces in the heap, is also the total number of active subspaces in the heap.

We have opted for setup number 1 in order to spend as little time as possible in the load balancing phase (see next section). When the heap does not contain any more Branch-and-Bound nodes with a selection value smaller than the present global upper bound it is an easy task to delete the remaining Branch-and-Bound nodes and continue with an empty heap. It is reasonable to assume that the number of “bad” subspaces is evenly spread throughout the processors, which means that each heap has approximately the same number of “bad” subspaces.

The load balancing phase is only entered if the processor is currently out of work or after a constant number of subspaces have been processed. It should happen often enough to keep the workload fairly distributed, ensuring that every processor is doing meaningful work, but not too often. If load balancing is done too often time that could have been spend better solving subspaces is used for insignificant load balancing operations.

6.4.3 The message passing/load balancing framework

This part of the parallel algorithm is identical for each processor and is only run after a certain number of subspaces have been processed or the processor is out of work. It takes care of receiving, handling and sending messages to the neighbouring processors.

The overall design is based on the design of a message passing/load balancing framework presented by Perregaard and Clausen in [PC98].

Each processor allocates memory for two buffers for each neighbour: one for incoming messages and one for outgoing messages. Every message received is acknowledged by returning a confirmation message to the sending neighbour. By allocating one buffer for incoming messages for each neighbour and by not sending more messages before a confirmation is received, the processors do not have to sit idle while transferring messages. If a message has not been acknowledged, the processor simply ignores that particular neighbour with respect to sending new information along.

Deadlock occurs when two processors wait for a message from each other thereby halting them. This cannot occur in our implementation. If a processor has sent message to one of its neighbours it does not halt until an acknowledgement arrives. Instead it regularly checks whether the acknowledgement has arrived from the receiving processor. If this is not the case the processors continues to work on other tasks (load balancing or solving subspaces). Meanwhile no further message is sent in the direction of the receiving processors. Sooner or later an acknowledgement will arrive (unless the network breaks down, but recovering from network/link breakdown is entirely outside the scope of this project).

There are 3 types of messages: new subspaces, update of global upper bound and termination message. Upon receiving a termination message the processors passes the termination message to the next processor and terminates. The first termination message is sent from the master as soon as it detects that all processors are idle.

A message with a new global upper bound consists of the value of the global upper bound and the routes that constitute the global upper bound. Only the master problem records the routes. The remaining processors send the routes along, but do not keep track of them.

Finally a message with new unsolved subspaces can be received. The message might actually not contain any message. Such a message indicates that the sending processor has no subspaces left to process. In order not to flood the network with these "out of work" messages, a flag is set so that the next "out of work" message can not be sent before the processor has received at least one unsolved Branch-and-Bound node. In case the message contains Branch-and-Bound nodes these are extracted from the buffer, checked against the value of the global upper bound and inserted in the heap. With each batch of subspaces, the senders heap size is included, and the local estimate on the senders heap size is hence only updated when the receiver received a batch of Branch-and-Bound nodes or a special "out of work" message. The token in the termination detection process is integrated in the messages with new unsolved subspaces.

Sending unsolved Branch-and-Bound nodes to a neighbour is thus based on an estimate of the size of the neighbours heap. This estimate is the latest received size. Let n be the heap size of the sending processor and n' the estimate of the heap size of the receiving processor. If the receiving neighbour is out of work, or $n > 2$ and $n - n' > \frac{n}{2}$ at least one subspace is transfered. The number of subspaces transfered is determined by the formula

$$\max\{\frac{n-n'}{3}, 1\}.$$

This formula is also used by Perregaard and Clausen in [PC98]. Among a number of different load balancing criteria this formula resulted in the best performance for the job-shop scheduling problem. Whether this criteria is also best for VRPTW has not been investigated, but would be interesting to study.

6.5 Summary

The parallel algorithm for the VRPTW developed is principally a number of sequential VRPTW algorithms splitting the work of solving the Branch-and-Bound nodes among each other.

Initially at least one subspace per processor is generated. If the problem is not solved during this initial phase the subspaces are divided among the processors.

The processors are set up in a ring, where it is possible to send and receive messages from both neighbours. Every time a new better global upper bound is determined it is communicated to the other processors via the ring. If the difference between the workload of two neighbouring processors becomes too large, the processor with a light load receives a number of unsolved subspaces from the heavy loaded processor. For every message received the receiving processor sends an acknowledgement back to the sending processor. It is not possible to send a message to a processor that has not yet acknowledged the last message that was send.

Termination detection is implemented using the algorithm described by Dijkstra et al. in [DFvG83].

A setup of 11 processors with 1 "master" and 10 "slaves" is depicted in figure 6.5.

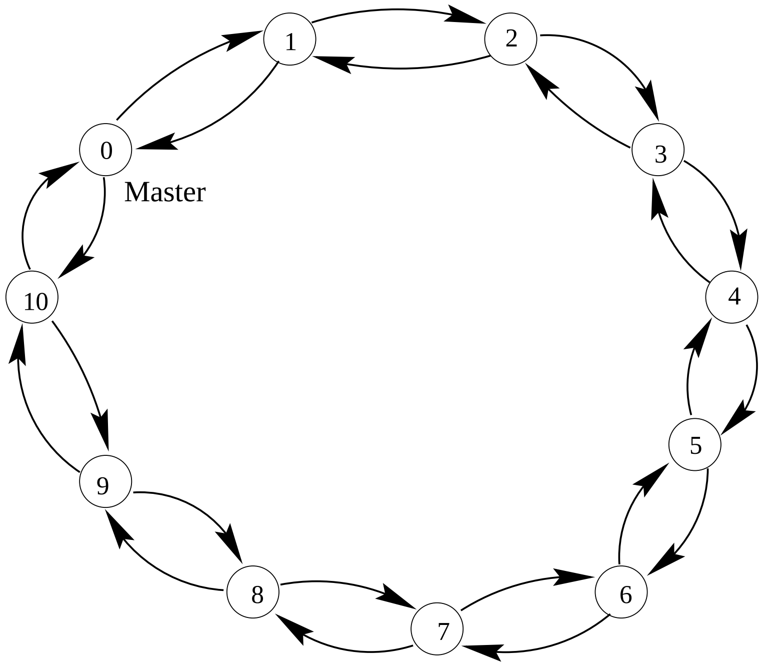


Figure 6.5: A example of the doubly-connected ring topology that the processors are organized as. Each processors can send and receive unsolved subspaces and new global upper bounds to and from both of its neighbours, while tokens in the termination detection process are only send one way (clock-wise).

Chapter 7

Sequential computational experiments

This chapter discusses the experiments made with the sequential code. The tests were run on two Hewlett-Packard HP 9000 series 800 model K460 computers called `serv2` and `serv3` at IMM. Both are equipped with PA-8000 processors running at 180 MHz. While `serv2` has 928 Mb of memory `serv3` only has 608 Mb. From SPEC[1] the HP-9000 K460 has a SPECint95 index of 11.8 and a SPECfp95 index of 22.2 (approximately the same performance figures are achieved by PC-systems based on the Intel Pentium II 300MHz systems). All running times presented in this chapter are presented in seconds. In order not to use too much of IMM's computer resources the algorithm was stopped after 50000 columns were generated or 2000 Branch-and-Bound nodes checked.

The algorithms are coded in C. We have used the LP-solver CPLEX version 3.0 for solving the master problem.

[1]Homepage: *www.spec.org*.

7.1 The Solomon test-sets

Getting involved in research on solving the VRPTW (in some older papers also referred to as VRSPTW – Vehicle Routing and Scheduling Problem with Time Windows) means that you sooner or later meet the Solomon test-sets first put forward by Solomon in [Sol87]. The test-sets, which can be downloaded from the WWW at

`http://dmawww.epfl.ch/~rochat/rochat_data/solomon.html`,

consists of sets of instances based on data from some of the problems used by Christofides et al. in [CMT79] for the standard routing problem. The test-sets reflects several structural factors in vehicle routing and scheduling as geographical data, number of customers serviced by a single vehicle and the characteristics of the time windows (e.g. tightness, positioning and the fraction of time-constrained customers in the instances). Customers are distributed within a $[0, 100]^2$ square.

The instances are divided into 6 groups (test-sets) denoted R1, R2, C1, C2, RC1 and RC2. Each of the test-sets contain between 8 and 12 instances. In R1 and R2 the geographical data is randomly generated by a random uniform distribution (see figure 7.1). In the test-sets C1 and C2 the customers are placed in clusters (see figure 7.2), and finally in the RC1 and RC2 test-sets some customers are placed in clusters while others are placed randomly (see figure 7.3). In the test sets R1, C1 and RC1 the scheduling horizon is short permitting approximately 5 to 10 customers on each route. The R2, C2 and RC2 problems have a long scheduling horizon making routes with more then 30 customers feasible. This makes the problems very hard to solve exactly and they have until now not been used to test exact methods. The time windows for the test sets C1 and C2 are generated to permit good, maybe even optimal, cluster-by-cluster solution. For each class of problems the geographical position of the customers are the same in all instances whereas the time windows are changed.

Each instance has 100 customers, but by considering only the first 25 or 50 customers smaller instances can easily be generated. It should be noted that considering 25 or 50 customers for the RC-sets the customers are

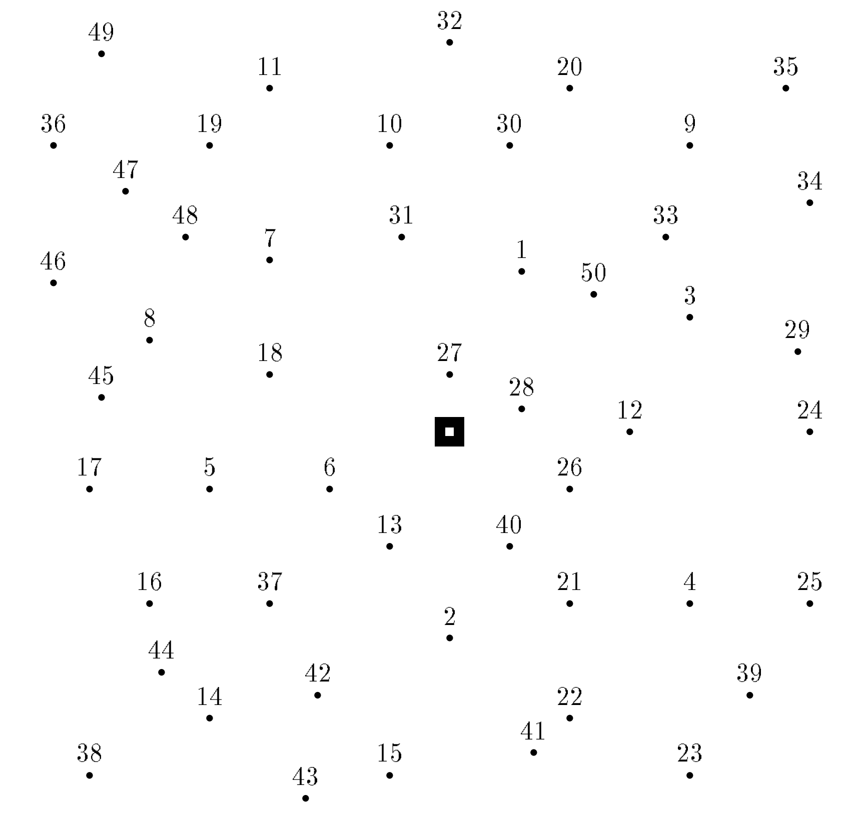


Figure 7.1: Example of the geometry of R-instances (R102 with 50 customers) where the depot is marked as a black square.

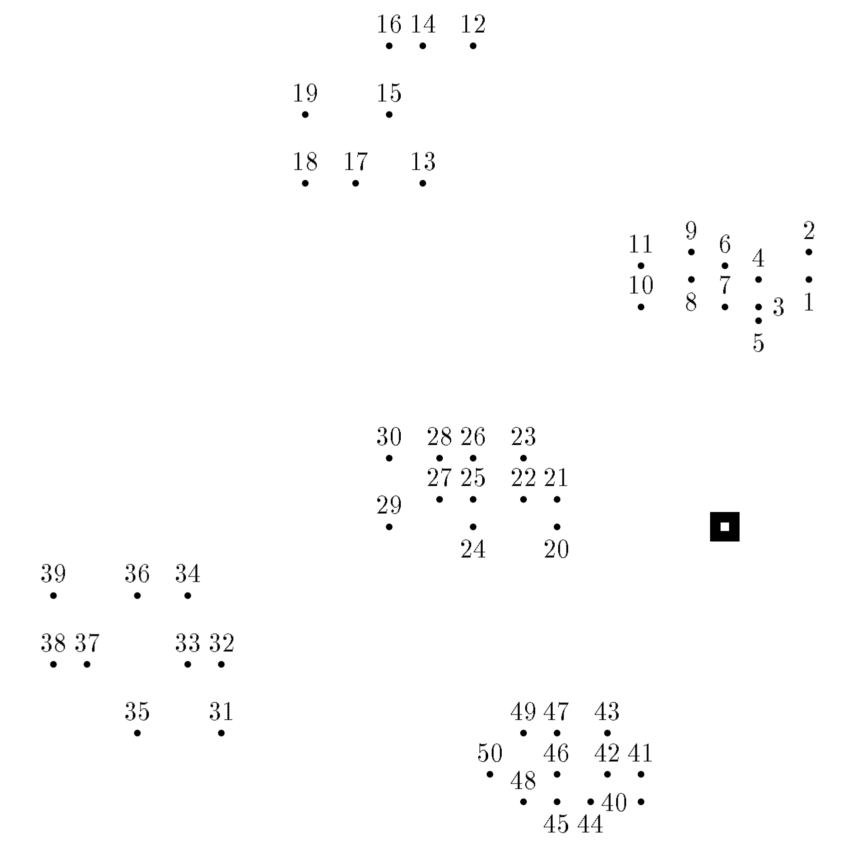


Figure 7.2: Example of the geometry of C-instances (C101 with 50 customers) where the depot is marked as a black square.

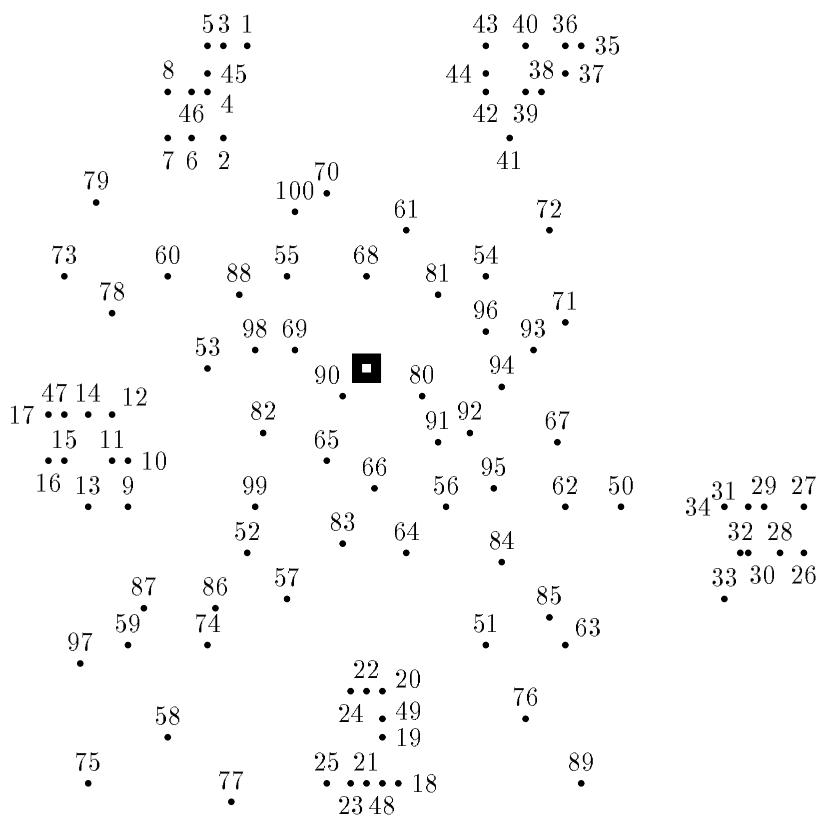


Figure 7.3: Example of the geometry of RC-instances (RC101 with 100 customers) where the depot is marked as a black square.

clustered as the clustered customers appear in the beginning of the file. Travel time between two customers is usually assumed to be equal to the travel distance plus the service time at customer we are starting from.

7.2 Using the Ryan-Foster branching rule in VRPTW

During a 5 month visit to the Department of Engineering Science of the University of Auckland the possibility of using the Ryan-Foster branching rule [RF81] instead of the arc branching rule that has traditionally been used within the vehicle routing community was investigated under the supervision of professor David M. Ryan.

The Ryan-Foster rule has been used in several papers solving crew scheduling and rostering problems. In crew scheduling, a number of flights (called legs) are to be manned (as legislation and union-contracts are different between pilots and cabin-crew, scheduling is done for each group independently). This is done by linking legs to form rosters that start at a crew-base and end at the same crew-base. Hence, customers in VRPTW corresponds to legs in crew scheduling, the depot in VRPTW becomes the crew-base in crew scheduling, and a route in VRPTW corresponds to a roster for an unnamed pilot/cabin crew. It is therefore obvious to try to use the experience gained from the crew scheduling problem in solving VRPTW.

In VRPTW terms the Ryan-Foster rule amounts to selecting two customers i_1 and i_2 and generating two Branch-and-Bound nodes: one in which i_1 and i_2 are serviced by the same vehicle and one where they are serviced by different vehicles. This is slightly different from the arc-branching rule normally used. In arc-branching i_2 must follow immediately after i_1 in one of the nodes, and in the other Branch-and-Bound node i_2 is not allowed to follow immediately after i_1. In the Ryan-Foster rule there is no constraint on the order of the customers. This should give a more balanced search tree, as opposed to arc-branching, where one arc is removed in one of the Branch-and-Bound nodes while up to $2(n-1)$ arcs are removed in the other Branch-and-Bound node. The drawback is that the branching decisions can not be implemented by changing the underlying graph.

Therefore, we have to add an array to each label containing two entries for each pair of customers that is in the "branching-path". If for instance, a branch is made in which (i_1, i_2) must be serviced by the same vehicle and (i_3, i_4) are not allowed to be serviced by the same vehicle, an array of size 4 needs to be allocated with each label to monitor whether the customers has been visited or not. This corresponds to adding new resources.

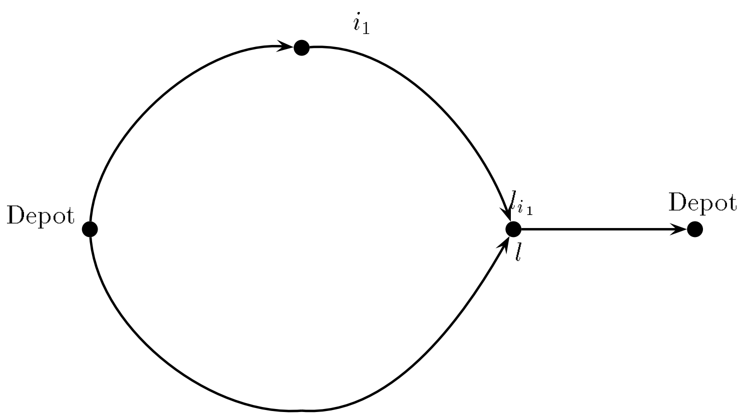


Figure 7.4: Comparison of two labels: The label l_{i_1} represents a partial route that contains customer i_1, while label l represents a partial route that does not contain customer i_1.

Consider figure 7.4 and assume that we have branched on the customer-pair (i_1, i_2) and we are working on the branch where both customers have to appear on the same route. Now in case the partial route containing i_1 dominates every other label at customer j, and no extension of the partial route to the depot containing customer i_2 results in a route with negative accumulated costs, no route will be available for insertion in the master problem. Therefore the dominance criteria must be extended if we want to use the Ryan-Foster rule. In the case where two customers have to be on the same route the labels representing a partial route containing both i_1 and i_2 dominates other configurations as long as they dominate with respect to distance/accumulated cost and time. Between the remaining types of partial routes no dominance can be established. The same considerations has to be applied to the branch where i_1 and i_2 can not be part of the same route.

The initial test with small problems looked promising, but when we turned our attention to medium and large scale problems the running times deteriorated drastically. When the depth of the Branch-and-Bound-tree became 5 or 6 (which is not unusual) the weaker dominance criterion decreased the performance substantially, and the scheme was not competitive.

We did not perform a large test of the VRPTW algorithm using the Ryan-Foster branching rule. The method had problems solving instances where the Branch-and-Bound trees were deep and therefore it did not seem worth to continue the work. To illustrate the problem we consider two instances. The R101 with 100 customers has a relatively small Branch-and-Bound tree and therefore the running time of the two branching rules (Ryan-Foster and arc-branching almost alike). Using the Ryan-Foster rule 15 Branch-and-Bound nodes are needed to obtain a solution in 22.24 seconds, while the algorithm with the arc-branching rule returned a solution after 25 Branch-and-Bound nodes and 27.78 seconds. In the case of R103 with 50 customers a solution was found in 42.16 seconds after examining 51 Branch-and-Bound nodes using arc-branching. Here the Ryan-Foster rule needed 312.11 seconds and 41 Branch-and-Bound nodes. Clearly the added states in the labels weakens the dominance criteria too much.

One of the important differences between vehicle routing and crew scheduling is the depth of the Branch-and-Bound trees. The depth in Branch-and-Bound trees of a crew scheduling problem is typically significantly smaller in vehicle routing. Another important aspect is that in crew scheduling cycles are not possible (it is not possible to fly the same leg twice). We have therefore not pursued the idea further.

7.3 Experiments with branching on resources

In [GDDS95] Gélinas et al. described how to branch on resource constraints. In the VRPTW model resource constraints are either capacity constraints or time windows.

The computational tests in [GDDS95] are made for a special variant of the VRPTW: the VRPTW with backhauling. In the backhauling variant of the VRPTW, customers are divided into two groups – a group of pick-up points and a group of delivery points. A restriction that all delivery points must be visited before any pickup point is furthermore imposed in [GDDS95]. With this model the following conclusion is drawn:

> *This strategy of branching on resource variables proved much more effective than branching on flow variables.*

The implemented branching scheme uses branching on the number of vehicles, branching on time windows and branching on arcs (branching on flow). As capacity windows seldom are constraining we do not try to use them to branch on. The arc-branching is only used in cases where branching on time windows cannot be applied. Hence, whenever branching on time windows is possible it is used. The branching scheme is tested against a scheme which branches only on the number of vehicles and arc-branches. Out of 21 instances, branching on time windows performed best in 17 cases, but only significantly (more than a factor 2) better in 9 cases. These results do not seem impressive, hence we set out to try to combine the branching on time window and arc-branching in a more intelligent way, trying to develop a branching scheme depending on the present geography, customer distribution, time window size etc.

In many papers it seems to be a foregone conclusion to use a best-first eager evaluation approach. Before evaluating branching on time windows the two selection functions best-first and depth-first in combination with the two bound evaluation schemes eager evaluation and lazy evaluation were tested. In eager evaluation, bounding of the Branch-and-Bound nodes is done before they are returned to the data structure handling the selection, while in lazy evaluation each Branch-and-Bound node is inserted with the bound value of for example the parent and bounding takes place immediately before branching.

In an initial phase we developed four versions of the branch-and-price algorithm for the VRPTW: two using best-first for node-selection and two using depth-first for node-selection. For each node-selection strategy we implemented a lazy and eager bound evaluation approach (as discussed by Clausen and Perregaard in [CP]).

From the rich set of instances in the Solomon test-sets we selected 8 instances (table 7.1), which are fairly difficult to solve and have different characteristics. These were used as a test-bed in the initial phase of the experiments. First of all we wanted to confirm the results of Gélinas et al. We therefore ran the 4 developed algorithms with:

- Setup 1:
  - Branching on the number of vehicles.
  - Branching on time windows.
  - Branching on arcs.
- Setup 2:
  - Branching on the number of vehicles.
  - Branching on arcs.

For each setup, 4 runs of the test-bed problems were made. The results are shown in table 7.2 and 7.3. The NoCo ("No more columns") entry indicates that the max. number of 20000 columns allowed were exceeded.

First thing to notice in table 7.2 is that the stack-implementation using eager evaluations is never the best. Even though the stack-implementation using lazy evaluation is the best on 3 instances it is only by a small margin, and when it is not the best it is way off the best running times. On the remaining instances the heap-implementation using lazy evaluation performs better. Note that the performance of depth-first based Branch-and-Bound can be improved by using an initial solution value better than ∞. A fast heuristic that can generate a good quality solution can significantly

Problem	No. of Cust.	Description
R101	100	All time windows are of size 10
R103	50	Half of the time windows are of size 10, the other half almost not constraining
R105	100	All time windows are of size 30
R107	50	Half of the time windows are of size 30, the other half almost not constraining
R110	50	All time windows are big (at least size 30)
R111	50	All time windows are big (at least size 30)
RC104	50	10 time windows of size 30, the rest are almost not constraining
RC108	50	All time windows are big (at least size 50)

Table 7.1: Description of the test-bed. For all instances the scheduling horizon is $[0; 230]$.

reduce the depth of the Branch-and-Bound tree. For a best-first Branch-and-Bound a good initial solution value can help reduce the number of Branch-and-Bound nodes stored in the heap.

For the setup where branching on time windows is not used (table 7.3) the results are even more clear. Again the stack-implementation using eager evaluation is never the best. Among the remaining setups the heap-implementation using lazy evaluation edges in front. Five times it is the best and it is never worse than than second.

So in our further tests of the time window branching scheme an implementation using a heap to store unsolved Branch-and-Bound nodes (best-first strategy) and lazy evaluation (storing each Branch-and-Bound node in the heap according to the bound of the parent).

Now comparing our two chosen implementations it is noteworthy that branching on time windows is better in 5 out of 8 instances, but only in 2 of the 5 instances is the difference really significant (both in R107 with 50 customers and RC108 with 50 customers branching on time windows is about a factor 2 faster).

		Heap						Stack					
		Lazy			Eager			Lazy			Eager		
Problem		time (s)		BB	time (s)		BB	time (s)		BB	time (s)		BB
	Customers	Veh.	Arc	TW	Veh.	Arc	TW	Veh.	Arc	TW	Veh.	Arc	TW
R101	100		28.79	41		35.51	20		25.33	41		36.63	21
		1	7	14	1	5	13	1	7	12	1	7	12
R103	50		34.24	31		35.82	15		50.19	81		38.29	19
		1	0	23	1	0	13	1	0	39	1	0	17
R105	100		817.56	381		1048.98	190		1432.21	789		2489.84	383
		8	0	242	6	0	183	9	0	385	10	0	372
R107	50		36.51	41		38.12	16		696.52	797		1591.77	404
		2	0	27	1	0	14	5	2	391	2	5	396
R110	50		26.87	11		26.02	6		4199.32	4193		2740.84	945
		1	0	5	1	0	4	100	6	1990	46	12	886
R111	50		568.01	629		525.24	254		1912.278	1615		7522.68	1144
		38	0	386	18	0	235	64	13	730	62	10	1071
RC104	50		187.74	9		186.30	5		185.33	9		194.112	5
		1	0	3	1	0	3	1	0	3	1	0	3
RC108	50		354.19	167		358.49	80		255.41	177		414.26	83
		1	0	112	1	0	78	1	0	87	1	0	81

Table 7.2: Using time windows (setup 1) on the test-bed. The first line for each instance contains the running time and the number of Branch-and-Bound nodes that was necessary to solve the problem. The second line displays the number of branchings performed on the number of vehicles, on arcs and using time windows.

		Heap						Stack					
		Lazy			Eager			Lazy			Eager		
Problem		time (s)		BB	time (s)		BB	time (s)		BB	time (s)		BB
	Customers	Veh.	Arc	TW	Veh.	Arc	TW	Veh.	Arc	TW	Veh.	Arc	TW
R101	100		18.82	25		24.25	13		16.76	25		24.87	14
		1	14	0	1	11	0	1	11	0	1	12	0
R103	50		54.12	49		54.99	21		56.15	69		70.48	32
		5	33	0	3	17	0	3	31	0	3	28	0
R105	100		680.92	327		920.02	166		NoCo			NoCo	
		20	151	0	17	148	0						
R107	50		75.48	67		95.11	34		101.33	117		138.91	58
		2	52	0	2	31	0	2	56	0	2	55	0
R110	50		26.89	13		27.23	7		NoCo			9669.23	1472
		2	5	0	2	4	0				3	1468	0
R111	50		545.64	559		713.80	275		472.83	555		689.31	270
		31	327	0	14	260	0	15	262	0	16	253	0
RC104	50		244.79	25		242.15	13		248.17	41		295.50	22
		1	14	0	1	11	0	1	19	0	1	20	0
RC108	50		712.47	341		902.78	171		NoCo			NoCo	
		3	235	0	1	169	0						

Table 7.3: Not using time windows (setup 2) on the test-bed.

In order to evaluate the branching schemes we selected three instances R105, R107 and R111 for further tests. In R105 all time windows are of size 30 which is small compared to the scheduling horizon. In R107 half of the time windows are of size 30 while the remaining half are significantly larger than size 30 and in R111 all time windows are significantly larger than size 30.

Now all time windows (except the time window of the depot) are changed from $[a_i, b_i]$ to $[a_i + r, b_i - r]$ so for positive r the time windows are reduced symmetrically around the center of the interval, while negative values of r enlarges the time windows accordingly. The experiments were carried out for $r = -10, -5, -3, -1, 1, 3, 5, 10$ and results are depicted in table 7.4.

Our first expectation was that reducing the time windows would lead to smaller running times, and, obviously, enlarging the time windows would lead to larger running times. Furthermore the idea was that as the time windows get larger, branching on time windows would become more efficient than branching on arcs. Finally we wanted to confirm the results of [GDDS95].

Making the time windows smaller did not however reduce running times. Looking at both the runs where we used branching on time windows and those were we did not use it, the running time may actually go up even though the time windows are smaller, e.g. as time windows are reduced by one third going from $r = 0$ to $r = 5$ in R105 running time goes up by a factor 4.5 from 823.49 to 3682.59 using branching on time windows, and by around a factor 2 from 665.32 to 1428.41 if not. There is no general trend in the relationship between size of time window and the running time in these data.

The reason for this behaviour must be found elsewhere, and in fact the reason is quite straightforward. Table 7.3 shows the gap in percent between the initial LP relaxation and the optimal IP solution. As the time windows are changed we unfortunately also change the general structure of the problem and this does in some cases lead to a weaker LP relaxation. In this model of the VRPTW the impact is significant as the gap is usually only closed in very small steps.

In the 24 new tests we ran only one (R107, $r = -10$) did not produce a result for any of the two branching strategies. In only 10 of 23 tests something is gained using branching on time windows, and only half of these actually led to a significant decrease (by more than a factor 2) in the running time.

As the change of every time window clearly did change the problem too much we tried on R105 and R107 to leave the "big" time windows untouched and only change part of the remaining ones; the philosophy being that changing the big time windows with the relative small values we are using does not change the structure significantly. R111 is not used in these tests because all time windows are relatively large. For the same set of possible r-values we tried only to change every second and every fourth "small" time window. The result of these runs are shown in the tables 7.6 and 7.7.

Again these results are not encouraging at all. Our problem is still that we cannot control the gap between the LP and the IP. Therefore, even small changes may result in quite drastic changes in running times. Out of the 32 new instances branching on time windows only performed better than branching on flows in 12 instances (1 instance could not be solved by any setup).

The conclusion from these tests are that our initial hypothesis that as time windows grow larger, branching on time windows helps can not be justified.

In fact, the picture is not clear. It is presently not clear to us in which situations branching on time windows yields better results than branching on arcs. Further investigations have to find relations between problem characteristics and the performance of branching on time windows.

7.4 Speeding up the separation algorithm for 2-path cuts

The (heuristic) separation algorithm for the 2-path cuts is developed by Kohl in [Koh95]. It contains a number of points where tuning is possible. In this section we report on ideas for tuning the generation of S sets and the checking whether the vertices can generate a TSPTW tour.

Problem	No. of Cust.	r	Using TW time (s)	Using TW BB	Not using TW time (s)	Not using TW BB
R105	100	0	823.49	381	665.35	327
R105	100	1	2320.12	1025	1331.28	732
R105	100	3	11959.71	4961	2193.08	1087
R105	100	5	3682.59	1745	1428.41	749
R105	100	10	18.30	29	14.59	21
R105	100	-1	2515.54	1097	2955.46	1177
R105	100	-3	18411.63	4389	8735.96	2523
R105	100	-5	NoCo		6569.79	1723
R105	100	-10	NoCo		8028.87	2675
R107	50	0	39.48	41	74.51	279
R107	50	1	62.74	53	36.51	33
R107	50	3	4080.12	1775	NoCo	
R107	50	5	460.64	507	1243.58	891
R107	50	10	26.11	43	38.83	75
R107	50	-1	354.22	321	115.23	111
R107	50	-3	220.78	229	182.96	167
R107	50	-5	138.32	151	176.72	215
R107	50	-10	NoCo		NoCo	
R111	50	0	561.84	629	550.37	559
R111	50	1	323.14	461	220.69	267
R111	50	3	115.43	123	114.66	111
R111	50	5	24.82	29	48.18	35
R111	50	10	11.83	15	15.20	23
R111	50	-1	1729.87	1487	548.32	587
R111	50	-3	4303.42	2201	NoCo	
R111	50	-5	383.18	297	1425.56	589
R111	50	-10	556.49	439	1062.48	591

Table 7.4: Testing branching on time windows on the same geography but different time windows.

Problem	No. of Cust.	r	LP	IP	gap in %
R105	100	0	13461.422	13553	0.68
R105	100	1	13591.179	13722	0.96
R105	100	3	14043.772	14207	1.16
R105	100	5	14389.533	14547	1.09
R105	100	10	16602.143	16672	0.42
R105	100	-1	13319.500	13431	0.84
R105	100	-3	13030.200	13189	1.22
R105	100	-5	12643.756	12814	1.35
R105	100	-10	12025.317	12144	0.99
R107	50	0	7044.380	7111	0.95
R107	50	1	7111.175	7164	0.74
R107	50	3	7236.800	7397	2.21
R107	50	5	7374.133	7492	1.60
R107	50	10	7899.500	7965	0.83
R107	50	-1	6977.321	7108	1.87
R107	50	-3	6950.300	7057	1.54
R107	50	-5	6874.659	6970	1.39
R107	50	-10	NoCo		
R111	50	0	6918.122	7072	2.22
R111	50	1	7047.800	7165	1.66
R111	50	3	7197.714	7276	1.09
R111	50	5	7241.641	7304	0.86
R111	50	10	7395.500	7453	0.78
R111	50	-1	6901.500	7072	2.47
R111	50	-3	6787.745	6980	2.83
R111	50	-5	6515.462	6689	2.66
R111	50	-10	6405.750	6561	2.42

Table 7.5: The gap between the LP relaxation and the optimal IP value.

Problem	No. of Cust.	r	Using TW time (s)	Using TW BB	Not using TW time (s)	Not using TW BB
R105	100	0	823.49	381	665.35	327
Every second changed						
R105	100	1	85.95	59	56.44	31
R105	100	3	62.60	45	39.52	21
R105	100	5	3546.56	1651	2337.59	1225
R105	100	10	35.26	39	23.24	17
R105	100	-1	7023.80	2505	3482.01	1433
R105	100	-3	1373.31	657	11484.09	3299
R105	100	-5	7473.87	2207	10906.47	3063
R105	100	-10	NoCo		NoCo	
Every fourth changed						
R105	100	1	340.35	203	210.29	145
R105	100	3	454.73	273	259.38	167
R105	100	5	135.47	93	123.19	85
R105	100	10	19.98	9	28.71	17
R105	100	-1	5588.24	2177	3782.05	1485
R105	100	-3	2949.82	1167	2574.70	1041
R105	100	-5	759.88	447	779.67	411
R105	100	-10	4072.78	1419	1464.65	697

Table 7.6: Not every time window is changed for R105.

Problem	No. of Cust.	r	Using TW time (s)	Using TW BB	Not using TW time (s)	Not using TW BB
R107	50	0	39.48	41	74.51	279
Every second changed						
R107	50	1	113.76	115	54.13	51
R107	50	3	214.34	157	242.19	119
R107	50	5	1608.23	887	469.26	379
R107	50	10	25.80	25	43.31	51
R107	50	-1	44.27	43	70.27	63
R107	50	-3	133.51	147	80.76	81
R107	50	-5	22.72	5	22.18	5
R107	50	-10	119.09	141	94.03	103
Every fourth changed						
R107	50	1	51.02	21	40.75	29
R107	50	3	92.60	49	70.25	79
R107	50	5	42.97	19	19.60	11
R107	50	10	791.43	395	788.70	515
R107	50	-1	59.35	31	72.59	67
R107	50	-3	110.42	55	67.37	61
R107	50	-5	37.95	3	29.04	11
R107	50	-10	33.65	5	24.94	13

Table 7.7: Not every time window is changed for R107.

7.4.1 A new way of generating the S sets

To generate candidates for 2-path cuts a number of sets of customers S are generated. Starting with S being the empty set customers are added as long as $x(S) < 2$. The question is which customers to add when. The order in which they are added may influence the cuts that are found and thereby how much the gap between the LP-value and the IP-value can be tightened.

Kohl et al. [KDM+99] uses the numerical order of the customers to determine when to add which vertices to the set S, which essentially means random order. We analyzed how the cuts that where found by Kohl's code were geographically distributed. The next four figures (7.5 – 7.8) show how the sets that were recognized as 2-path cuts were positioned. They are typical for the more than 20 tests we made.

It is quite evident that cuts are clusters of customers close to each other. Looking at the four presented figures (and additional plots all show the same characteristics as the ones in the thesis) the closest neighbour within a cut was one of the five closest customers on an over-all basis.

A systematic search for cuts will make the search more effective. We therefore propose the following scheme:

1. Start adding customers to the cuts on a nearest-neighbour basis. The search is started with the set $\{i\}$ for all customers i.
2. Check only the closest neighbours. It seems that the customers in the cuts are close to each other, therefore it does not make sense to try to extend the set S with all possible customers.

   The number of customers checked is limited to the nearest 5 not yet in the set.

Additionally we introduced a success criteria. If the number of sets S generated that leads to cuts gets below a certain threshold we backtrack without investigating the branch further. A number of preliminary tests lead to a threshold of 3%.

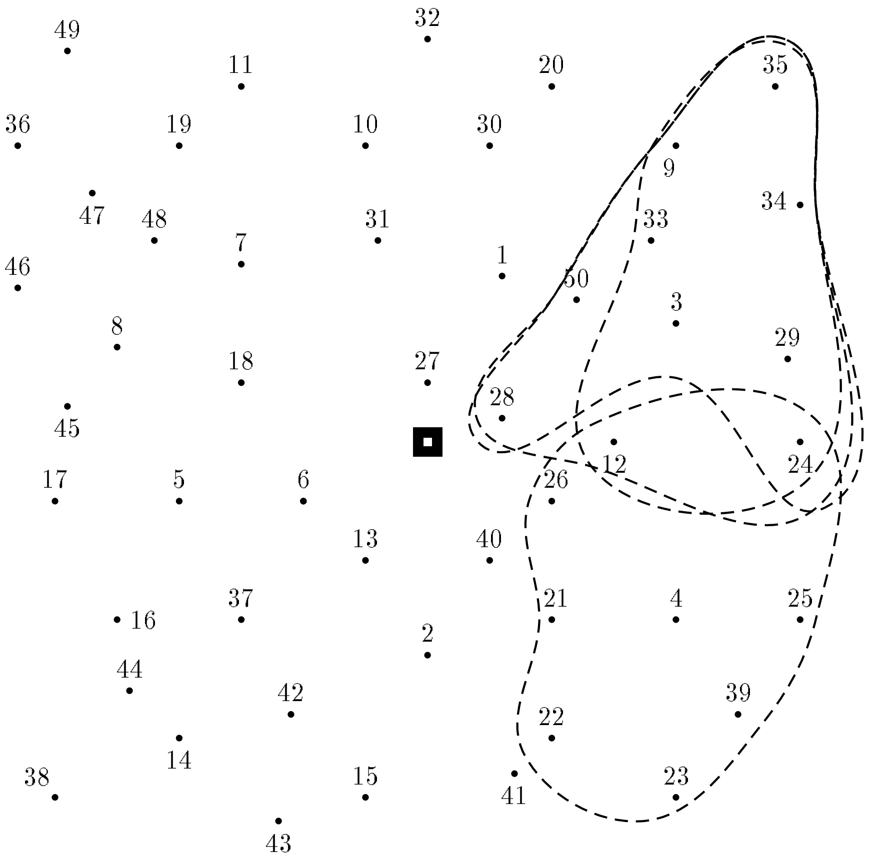


Figure 7.5: R103 with 50 customers. The dashed lines represents the sets that did become 2-path cuts.

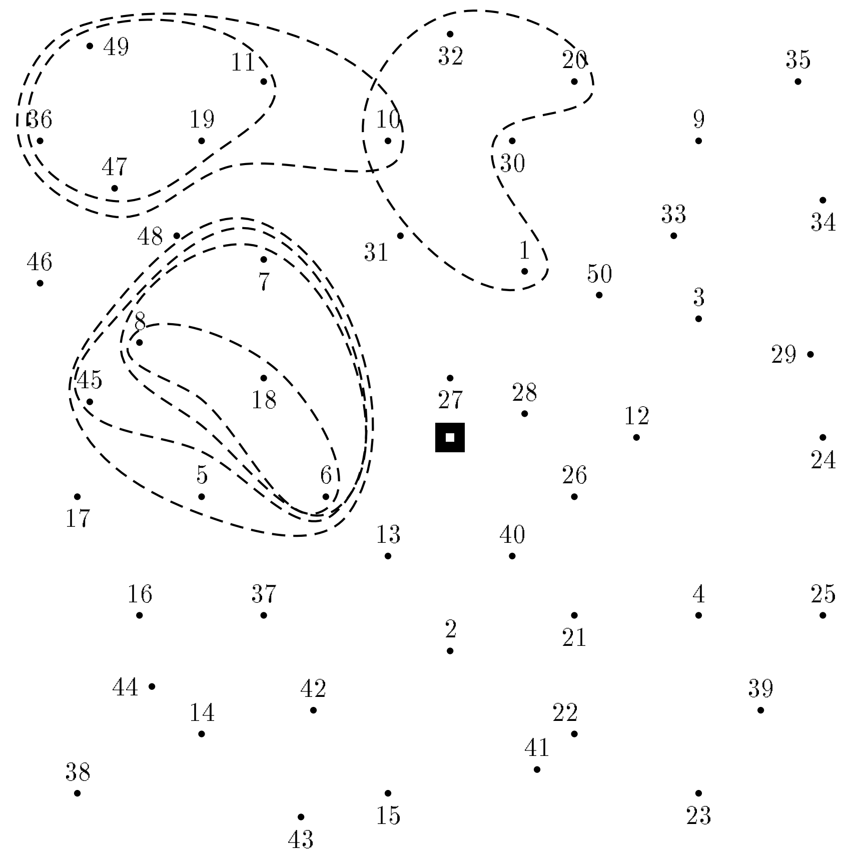


Figure 7.6: R105 with 50 customers. The dashed lines represents the sets that did become 2-path cuts.

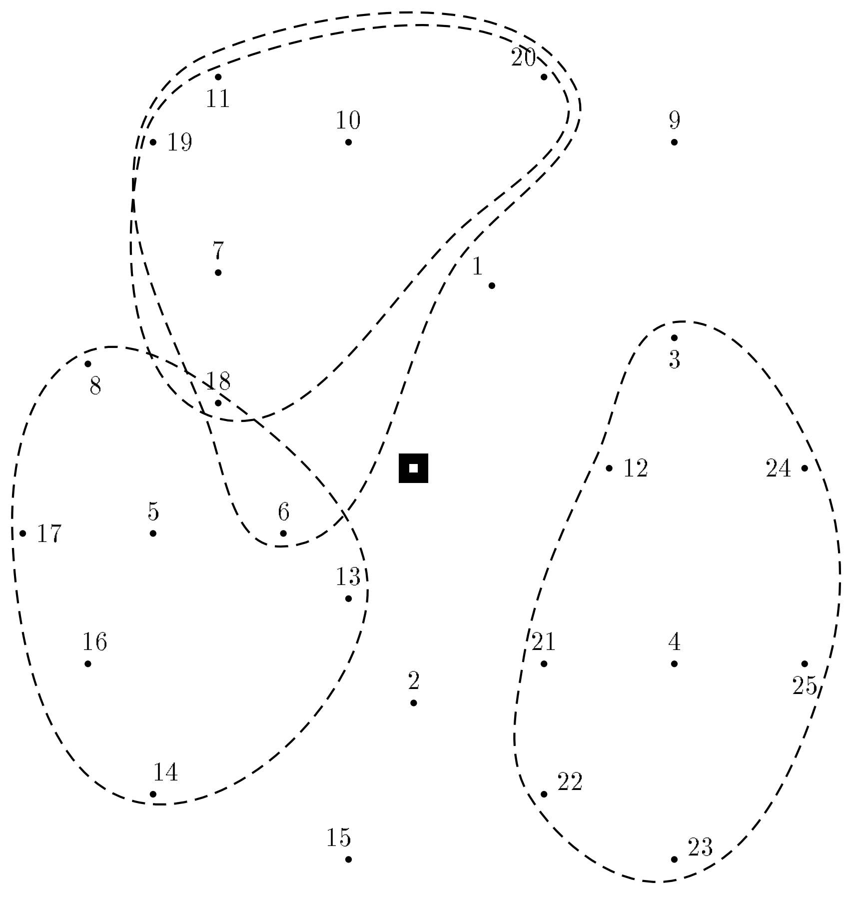


Figure 7.7: R110 with 25 customers. The dashed lines represents the sets that did become 2-path cuts.

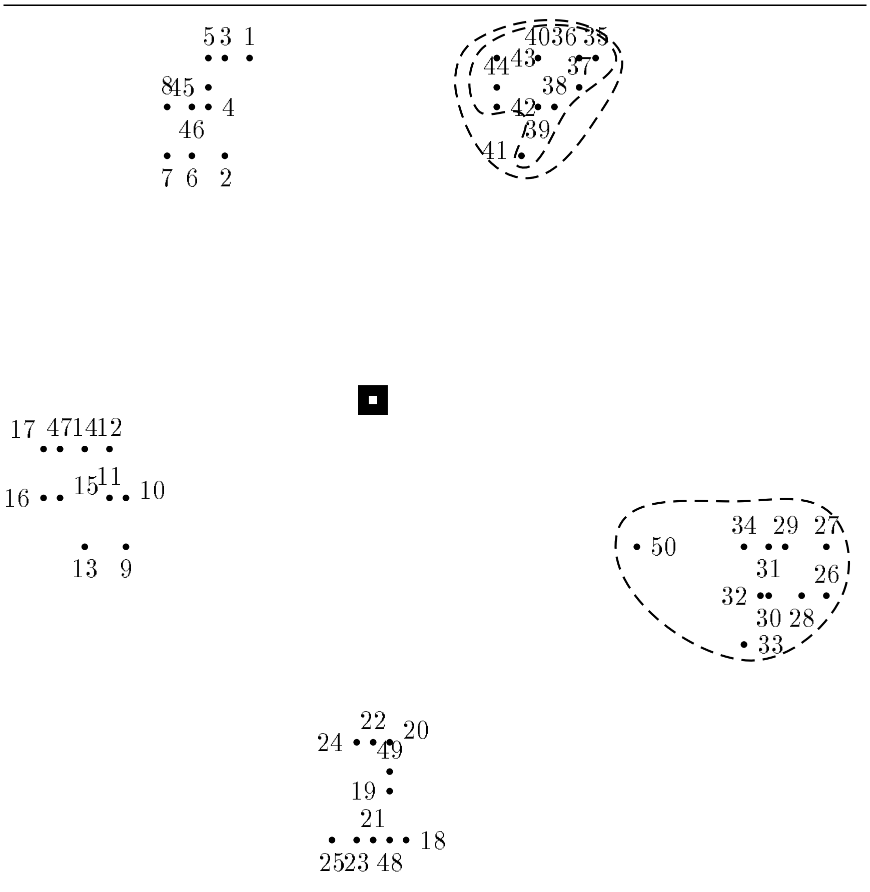


Figure 7.8: RC103 with 50 customers. The dashed lines represents the sets that did become 2-path cuts.

This scheme was tested using 49 instances from the Solomon test-sets and our instances developed in section on resource-branching (section 7.3) The results are show in the tables 7.8 and 7.9. Both loss in quality and gain in time are percentages, where a negative number in the "loss in quality"-column indicate a tighter bound in the new scheme and an negative number in the "Time gain"-column indicate that the old scheme was faster.

All in all, the loss in total quality was a mere 0.01% and we achieved a scheme that totally was 174% faster. Looking at the numbers for the individual runs it is remarkable that the largest gains are made where the running times where largest and these gains are achieved without any decrease in quality. In fact, the bounds achieved by the new scheme are better (see instance 28, 34, 38, 41 and 42). The instances 43 – 49 are the RC1 problems with 50 customers and it is noteworthy that in 5 out of the 6 cases where the new scheme took longer time to run are among those problems – further investigation might reveal why that is the case.

7.4.2 Heuristic for the "feasibility TSPTW" problem

The TSPTW problem has only been the subject of attention in a few papers. In the first paper on the subject ([Sav56]) by Savelsbergh, a heuristic is developed for the TSPTW. More important for our purpose is the proof by Savelsbergh that even the problem of finding a feasible solution to the TSPTW is $\mathcal{NP}$-hard. Savelsbergh therefore develops a heuristic for finding a feasible solution before he applies his TSPTW heuristic. The heuristic for finding a feasible TSPTW is using the same principles as the 2-stage insertion heuristic for the VRPTW presented by Solomon in [Sol87].

In the 2-stage insertion sort, the first stage is used to find the best insertion point for each unrouted customer. The "attractiveness" is determined by a function f_1. A function f_2 then determines which one of the insertions is to be performed.

In the implementation of the 2-path cuts by Kohl, a dynamic programming algorithm is used to check whether there is a TSPTW tour or not. The algorithm terminates when all possibilities have been checked or a TSPTW tour has been found. This algorithm is not polynomial but it is relatively fast as the number of vertices is rather small.

No.	LP bound	Old S Scheme		New S Scheme		Loss in qty.	
		Bound	Time	Bound	Time		Time gain
1	13419.50	13452.02	7.32	13443.19	5.47	0.07	33.77
2	12125.32	12132.40	13.14	12130.68	8.12	0.01	61.86
3	13130.20	13165.59	8.13	13165.25	6.17	0.00	31.81
4	12743.76	12779.34	7.81	12755.05	5.77	0.19	35.34
5	13691.18	13716.41	8.36	13760.86	5.95	-0.32	40.42
6	16702.14	16727.33	4.60	16708.67	3.70	0.11	24.31
7	14143.77	14246.13	7.13	14242.38	5.57	0.03	28.02
8	14489.53	14600.25	8.20	14600.25	5.87	0.00	39.78
9	7027.32	7034.56	2.35	7033.83	1.66	0.01	41.52
10	6747.95	6750.71	2.87	6748.28	2.37	0.04	21.01
11	7000.30	7005.15	2.44	7004.71	1.43	0.01	71.01
12	6924.66	6924.82	2.21	6924.82	1.53	0.00	44.31
13	7161.18	7171.60	2.21	7171.60	1.62	0.00	36.30
14	7949.50	7953.83	2.31	7953.83	2.07	0.00	11.59
15	7286.80	7293.25	1.88	7293.25	1.41	0.00	33.19
16	7424.13	7433.25	2.20	7431.68	1.76	0.02	25.26
17	6951.50	6956.37	3.47	6956.37	2.09	0.00	66.09
18	6455.75	6460.23	5.51	6460.14	2.84	0.00	94.08
19	6837.75	6842.49	5.09	6842.49	2.31	0.00	120.26
20	6565.46	6569.43	3.82	6569.43	1.74	0.00	119.66
21	7097.80	7100.25	2.10	7100.25	1.63	0.00	28.68
22	7445.50	7497.61	1.73	7497.61	1.74	0.00	-0.69
23	7247.71	7256.62	2.27	7257.64	1.54	-0.01	47.08
24	7291.64	7304.90	3.35	7291.68	1.33	0.18	151.35
25	16411.50	16440.00	5.86	16440.00	4.64	0.00	26.39
26	12163.12	12163.81	8.18	12163.81	5.51	0.00	48.47
27	9591.25	9606.80	85.06	9606.80	31.80	0.00	167.49
28	13561.42	13584.93	7.87	13587.08	5.71	-0.02	37.82
29	12364.40	12374.04	8.62	12369.54	5.63	0.04	53.20
30	11405.87	11426.34	10.94	11426.59	6.47	0.00	69.11
31	10584.82	10599.39	18.27	10599.39	11.18	0.00	63.45
32	10420.28	10420.75	13.35	10420.28	5.92	0.00	125.43
33	9291.92	9315.68	538.64	9319.14	112.45	-0.04	378.99
34	15940.94	16273.93	16.66	16272.76	8.83	0.01	88.65
35	14136.46	14478.48	27.35	14470.00	19.36	0.06	41.25
36	12284.95	12472.98	74.32	12472.63	30.36	0.00	144.80
37	11043.33	11223.54	872.90	11223.52	412.38	0.00	111.67
38	14811.60	15198.00	22.55	15193.75	9.10	0.03	147.73
39	13187.81	13432.83	81.47	13425.10	21.82	0.06	273.33

Table 7.8: Comparison between the old scheme for S sets by Kohl and our new search scheme. Gains and losses are measured in %. (Part 1)

No.	LP bound	Old S Scheme		New S Scheme		Loss in qty.	
		Bound	Time	Bound	Time		Time gain
40	11806.89	11901.61	254.90	11904.24	120.74	-0.02	111.12
41	10730.11	10904.53	961.22	10911.21	205.09	-0.06	368.68
42	8550.21	9466.60	0.67	9466.60	0.57	0.00	17.89
43	7249.02	7994.97	1.04	7994.97	0.56	0.00	86.02
44	6481.33	7136.82	1.02	7136.82	1.42	0.00	-28.27
45	5468.00	5487.50	8.33	5487.50	21.82	0.00	-61.84
46	7594.43	8453.46	0.82	8453.46	0.83	0.00	-0.72
47	6694.33	7209.79	11.95	7209.79	5.27	0.00	126.88
48	5964.76	6373.36	6.64	6373.36	8.36	0.00	-20.53
49	5439.57	5998.89	11.92	5998.89	22.16	0.00	-46.20
	Total	494313.57	3161.04	494313.57	1153.67	0.01	174.00

Table 7.9: Comparison between the old scheme for S sets by Kohl and our new search scheme. Gains and losses are measured in %. (Part 2)

Running a heuristic for sets with only a small number of vertices does not seem reasonable but for some of the larger instances encountered this may be a way to reduce the running time.

In the heuristic implemented by Savelsbergh in [Sav56], feasibility is the most important but not the only criteria. His second criteria is to generate a good starting solution for the second phase of his algorithm. Our sole purpose is to find a feasible solution. Otherwise we have to run the exact code of Kohl.

In our heuristic for the "feasibility TSPTW" problem we reuse the two-stage concept, but focus only on maintaining as much flexibility as possible.

In order to explain how the heuristic constructs a solution a few terms have to be defined. For every **routed** customer i variable s_i is the service time of the customer. Now let PFS_i (Push Forward Shift) be defined:

$$PFS_i = b_{n+1} - (s_i + \sum_{k=i}^{n} t_{k,k+1}).$$

So PFS_i is the difference between the earliest time we can arrive at the depot disregarding time windows and the closing time of the depot. As time windows are not regarded computing PFS_i the difference is an upper bound on the time available until the depot closes.

Another measure is $em_{i,u,j}$ which denotes the extra mileage in time units needed to go from customer i to customer j via customer u instead of going directly from i to j.

For each routed customer i lt_i denotes the latest time is it possible to arrive at customer i in order for the route to remain feasible, that is, lt_i can be calculated by

$$lt_i = \min\{lt_{i+1} - t_{i,i+1}, b_i\}$$

and is maintained throughout the algorithm. Note that lt_i only has to be recalculated for customers between the depot and the newly inserted customer (see figure 7.9).

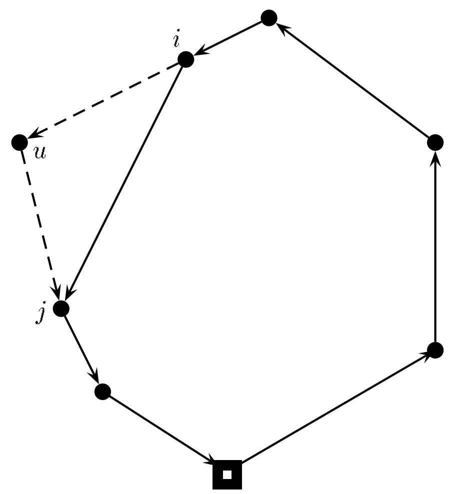


Figure 7.9: Only the latest arrival time of the customers from the depot to customer i has to be recalculated. The customers from j to the depot do not get their latest arrival time changed by the insertion of customer u.

In stage 1 the best possible insertion position for every unrouted customer is found. The point of insertion chosen is the position between two routed customers that maximize

$$\min\{lt_u - \max\{s_i + t_{iu}, a_u\}, PFS_j - em_{i,u,j}\}.$$

By calculating $lt_u - \max\{s_i + t_{iu}, a_u\}$ we get a measure of how much of the time window is left if we insert u between i and j. As PFS_j is a measurement of the flexibility of the path from j to the depot, $PFS_j - em_{i,u,j}$ is an estimate of how much flexibility that remains in the path when we insert customer u between i and j. In order to retain as much flexibility as possibly we maximize the minimum of the two numbers.

Whereas the insertion points selected in the first stage only depends on how much of the flexibility of the inserted customer is destroyed, the selection in the second stage focuses with a more "global" objective.

Among the best insertion points for each unrouted customer we choose the one that minimizes lost flexibility of the predecessor k' and the successor k'', that is,

$$\min\{(lt_{k'}^{\text{old}} - lt_{k'}^{\text{new}}) + (s_{k''}^{\text{new}} - s_{k''}^{\text{old}})\}.$$

Furthermore small time windows are preferred to large time windows. This forces the heuristic to insert the customers with small time windows first. Small time windows are less flexible, whereas customers with large time windows are likely to fit into the route at several insertion points. In this heuristic a time window smaller than half the size of the present best candidate is always chosen independently of the insertion quality.

We ran 3 tests using the "TSPTW feasibility" heuristic using a test-bed of 27 instances. The 27 instances were selected among the Solomon test cases and the instances generated for the test of the scheme for branching on time windows.

The 27 instances were selected to be "fairly hard" to solve, that is, they would at least require 2 minutes of computing time and should require at least 50 Branch-and-Bound nodes to be solved. Of course changing the algorithm does also change the running time and the number of Branch-and-Bound nodes to be investigated but the instances where determined by our first running version of the algorithm (without cuts, branching on time windows etc.). For completeness the instances are listed in table 7.10

No.	Name	Instance			Customers
1	S105-1	R105	-1		100
2	S105-10	R105	-10		100
3	S105-3	R105	-3		100
4	S105-5	R105	-5		100
5	S1051	R105	1		100
6	S1053	R105	3		100
7	S1055	R105	5		100
8	S1073	R107	3		50
9	S1075	R107	5		50
10	S111-10	R111	-10		50
11	S111-5	R111	-5		50
12	S111-1	R111	-1		50
13	S1055-2	R105	5	2	100
14	S105-1-2	R105	-1	2	100
15	S105-3-2	R105	-3	2	100
16	S105-5-2	R105	-5	2	100
17	S105-10-2	R105	-10	2	100
18	S105-1-4	R105	-1	4	100
19	S105-3-4	R105	-3	4	100
20	S105-5-4	R105	-5	4	100
21	S105-10-4	R105	-10	4	100
22	–	R103			100
23	–	R104			50
24	–	R105			100
25	–	R111			50
26	–	RC102			50
27	–	RC107			50

Table 7.10: The 27 instances in our test-bed.

In the first test we used the heuristic for every possible candidate set. In the two remaining tests we only used the heuristic for sets containing more than 8 respectively 11 customers. In case the heuristic can not find a feasible route we need to also run the optimal method described by [Koh95] therefore it seems to be a good idea to limit the use of the heuristic. For the set with a small number of customers we only use the exact method. The results are show in table 7.11.

It should be noted that we have also run a version of our VRPTW algorithm not incorporating the "TSPTW feasibility" heuristic, but it returned performance results almost identical to the "Above 11" test from table 7.11.

The numbers in the table are almost identical. The heuristic achieves a quite good success-rate for the test where we used it on every possible candidate set. But running the optimal method in case the heuristic does not succeed is not a problem as the candidate sets are small. For the "Above 8" and "Above 11" tests the heuristic is called too few times to make a significant impact on the running time. In the light of the identical running times we have not investigated this idea further.

7.5 Using the "trivial" lower bound

As we are using branching on vehicles we can almost without any cost strengthen the lower bound on the number of vehicles. Summing the demands of each customer and dividing the number by the capacity of the vehicles gives us a (fractional) lower bound on the number of vehicles required. By taking the ceil ($\lceil \cdot \rceil$) we get an integer lower bound. That is instead of starting with 0 as a lower bound we can use $\lceil \sum_{i \in \mathcal{C}} d_i / q \rceil$. In table 7.12 running times with and without this lower bound is compared.

In both tests, cuts were inserted in the root node, using our new scheme for finding the S sets for the 2-path cuts.

In 10 out of 27 instances using the trivial lower bound actually makes the algorithm perform worse than without it (that is with lower bound equal to 0). Only in instance 4 the deterioration is significant which is partly due to an increase in the number of Branch-and-Bound nodes.

	Every time			Above 8			Above 11		
No.	Time	TSPTW	Succ.	Time	TSPTW	successes	Time	TSPTW	successes
1	786.43	209	155	787.26	3	0	791.90	0	0
2	10035.18	225	155	10138.99	15	0	10084.01	0	0
3	3486.40	227	175	3487.86	6	1	3490.15	0	0
4	4858.22	226	179	4794.65	4	1	4883.50	0	0
5	238.73	214	173	242.77	4	0	237.14	0	0
6	362.44	222	167	365.84	1	0	360.64	0	0
7	606.63	277	205	612.05	1	0	605.69	0	0
8	2224.62	97	76	2182.89	6	0	2205.47	0	0
9	508.81	123	98	508.30	7	0	515.64	0	0
10	832.36	96	66	821.83	21	2	834.38	1	0
11	595.38	56	41	591.83	8	0	597.85	0	0
12	531.79	99	77	521.23	4	0	526.41	0	0
13	920.09	190	142	915.51	1	0	919.85	0	0
14	1660.32	236	195	1619.20	3	0	1688.93	0	0
15	2011.45	235	184	1951.10	6	1	2013.45	0	0
16	3628.71	209	162	3552.67	5	1	3675.66	0	0
17	6626.88	219	164	6546.74	6	1	6529.29	0	0
18	1220.05	210	170	1193.57	5	0	1212.19	0	0
19	1417.15	199	168	1384.99	3	1	1433.97	0	0
20	679.64	221	165	673.36	3	1	674.21	0	0
21	907.23	235	171	894.04	3	1	924.16	0	0
22	1602.40	149	125	1526.49	9	1	1532.76	0	0
23	834.61	105	68	1113.23	15	0	1128.58	1	0
24	231.25	219	182	229.94	3	0	232.06	0	0
25	291.32	116	92	288.29	7	0	293.31	0	0
26	1852.57	51	39	1835.95	7	2	1843.45	0	0
27	146.48	29	24	147.71	8	4	149.09	0	0

Table 7.11: Performance results of using the “TSPTW feasibility” heuristic.

No.	No lower bound Time (s)	BB	Using lower bound Time (s)	BB
1	551.24	181	545.86	183
2	9938.41	2051	9517.41	1967
3	2245.05	667	2118.98	673
4	4142.96	1249	5039.44	1297
5	298.10	97	286.20	97
6	382.77	113	391.53	113
7	309.39	65	284.87	63
8	1980.50	915	2190.60	907
9	417.79	389	392.69	407
10	1370.65	551	1278.18	527
11	797.15	335	696.77	335
12	648.30	533	599.57	525
13	970.43	401	1019.45	409
14	1433.01	521	1377.95	518
15	1298.80	441	1362.69	443
16	3241.34	875	3245.92	929
17	16746.09	2981	17524.91	3279
18	1150.14	402	1246.37	407
19	954.19	333	932.56	337
20	363.22	119	351.30	119
21	1097.26	393	972.43	345
22	2550.17	53	2082.70	53
23	1306.25	191	1232.92	199
24	296.80	101	284.06	99
25	363.41	339	378.42	345
26	1826.05	1645	1948.98	1681
27	276.93	75	257.21	79

Table 7.12: Comparison of the running times of the VRPTW algorithm with and without the "trivial" lower bound.

Generally the two sets of running times are not significantly different, although with a slight advantage to the version of the algorithm using the lower bound.

It is quite interesting to calculate the running time per Branch-and-Bound node. In the 10 problems where the trivial lower bound results in a worse performance the running time per Branch-and-Bound node is also worse except for problem 15 and 16, and in the remaining 17 instances the running time per Branch-and-Bound node is better using the trivial lower bound except for problem 21. This means that the worse performance is not (only) because of an increased number of nodes that has to be examined, but also the average time for each node itself is larger.

Further research might reveal why the trivial lower bound is not that helpful, and under which conditions it is. The trivial lower bound will be used in the code from now on.

7.6 Generating cuts outside the root node.

As discussed earlier in section 3.4.3 generating 2-path cuts in other subproblems than the root does not necessarily produce cuts that are globally valid. In [Koh95], Kohl generates cuts as long as no arc-branching has been performed.

Running the separation algorithm as long as no arc-branching has been performed makes it possible to look for 2-path cuts and use them globally if any are found. Therefore, in a first small step towards running the separation algorithm in additional Branch-and-Bound nodes the benefit of running the separation algorithm after branching on the number of vehicles were tested (see table 7.13). In the "Root only" test the separation algorithm was only used in the root, while it was used additionally after branching on the number of vehicles in the "Root and more" test.

Generally the results in table 7.13 do not present a clear picture. The majority (17 out of 27) of tests indicate that running only the cuts in the root is better, but the advantage tends to be a fraction larger when using the separation algorithm as long as possible. The running times are with

No.	Root only		Root and more	
	Time (s)	BB	Time (s)	BB
1	528.94	183	463.73	147
2	9622.31	1967	10922.35	2079
3	2048.29	673	1549.99	513
4	4968.14	1297	5298.81	1399
5	287.71	97	290.98	97
6	387.82	113	399.95	100
7	282.33	63	271.49	33
8	2177.73	907	2346.82	933
9	391.40	407	388.87	355
10	1274.21	527	1072.56	517
11	695.54	335	703.17	335
12	584.91	525	567.67	479
13	1008.26	409	991.34	381
14	1406.80	518	1768.50	603
15	1363.43	443	1496.79	415
16	3301.59	929	3045.77	805
17	17418.91	3279	7844.59	1521
18	1261.14	407	1308.73	414
19	933.61	337	953.66	337
20	353.78	119	348.62	113
21	986.22	345	1042.31	337
22	2229.15	53	2419.49	55
23	1220.46	199	1226.28	199
24	284.09	99	321.05	99
25	377.82	345	338.57	291
26	1952.21	1681	1950.56	1681
27	256.06	79	263.75	79

Table 7.13: Comparison of the effects of generating cuts as long as possible, that is, until we do arc-branching.

the exception of instance 17 not far apart – the difference between the two tests is never greater than 25%. As no clear conclusion can be drawn of these tests we continue to use the setup of Kohl generating cuts as long as no arc-branching has been performed.

Even though the 2-path cuts are not guaranteed to be globally valid when generated in Branch-and-Bound nodes the information can be used locally to fathom the Branch-and-Bound node. We therefore ran a version of our algorithm where 2-path cuts and subtour elimination cuts were detected also in the Branch-and-Bound nodes. The result is depicted in table 7.14. We have used the algorithm only generating cuts in the root as reference. As the difference in running time between the reference algorithm and the version using the cuts locally is fairly small we have given the difference in running time in table 7.14 in percentages. If the value is positive the reference algorithm is faster.

In none of the 27 instances do the cuts lead to smaller running times. In 6 of the instances fewer Branch-and-Bound nodes needed, but the running time is not better than the reference algorithm, as the separation algorithms have to be run on every Branch-and-Bound node.

A natural extension is to keep the generated cuts in a “cut pool” in order to insert them in the problem whenever possible. This involve the development of a cut management module to the existing program. Based on the tests made this does not seem worthwhile.

7.7 Reducing the number of columns

Each call to the route generator (the SPPTWCC function) typically generates more than one route, and as mentioned earlier experiments made by other authors suggest that if more than one route is available they should be entered into the master problem (the relaxed set partitioning problem). As also suggested by other authors we impose an upper limit on the number of columns that are entered into the master problem after each call to the route generator. In order to rank the columns, the accumulated costs are used.

No.	Ref. BB	Local cutting BB	Local cutting ZeroCut	Diff. in running time
1	183	183	117	1.15
2	1967	1967	1112	1.21
3	673	668	485	1.01
4	1297	1291	991	1.19
5	97	97	71	1.11
6	113	113	69	1.18
7	63	63	43	1.07
8	907	907	862	1.14
9	407	407	353	1.19
10	527	527	310	1.14
11	335	335	210	1.14
12	525	522	504	1.03
13	409	408	332	1.09
14	518	518	401	1.12
15	443	443	320	1.18
16	929	929	844	1.21
17	3279	3276	2941	1.16
18	407	407	361	1.22
19	337	337	275	1.09
20	119	119	110	1.11
21	345	345	299	1.12
22	53	53	31	1.23
23	199	199	181	1.22
24	99	96	45	1.07
25	345	345	237	1.04
26	1681	1681	1551	1.16
27	79	71	49	1.10

Table 7.14: Generating and using cuts locally. Column 2 contains the number of Branch-and-Bound nodes used by the reference algorithm, while column 3 contains the numbers used by the algorithm using the cuts locally. The column labelled 'ZeroCut' contains the number of Branch-and-Bound nodes where no cuts where generated. Last column is the difference in running time.

After a number of Branch-and-Bound nodes have been processed, the number of columns can grow very large and some of the columns might never have been part of the basis. Removing these columns would leave room for some new ones. Additionally this might speed up the LP-code and the work we have to do each time we start working on a new Branch-and-Bound node (for example checking whether any of the arcs in the path of a route are removed from the problem in the current Branch-and-Bound node). So here we have two goals: being able to handle larger problems and solving problems faster.

To check if anything can be gained, our 27 problems from the test-bed where used. The program was extended with a possibility to keep track of which columns that did participate in the basis at some point. The result was that as few as 10% and as many as 28% of the columns had been in the basis some time during the execution of the program. For the bulk of the problems the percentage was approximately 16 to 18. So only approximately 1 out of 5 entered columns ever became part of the basis. At least from a memory point of view there is something to be gained. Whether there is something to be gained in speed is more questionable. A possible scenario is to remove the obsolete columns after every k Branch-and-Bound nodes. Todays state-of-the-art LP-solvers (as CPLEX) are very fast (our problems of more that 10000 columns are by no means large in relation to solvable LP-problems) so it is not clear whether the contribution by removing columns is significant enough to produce faster running times. A side effect, as mentioned earlier, is that with fewer columns in the set partitioning few columns have to be checked every time we start working on a new Branch-and-Bound node. These effects have to counterbalance the time used on removing obsolete columns and time used generating them again in case they are needed in future Branch-and-Bound nodes.

To test the possibilities we ran four versions of our algorithm with column removal added. One version deleted obsolete columns after every Branch-and-Bound node, one after every 5 Branch-and-Bound nodes, one after every 10 Branch-and-Bound nodes and finally one after every 20 Branch-and-Bound nodes. Table 7.15 presents the results of running the algorithm without column deletion for reference. The results for our 27 problems in the test-bed are shown in table 7.16 - 7.19 (for reference columns 2 to 3 display the two most important columns from the reference algorithm data (column 2 and 3 in table 7.15) for running the code without column deletion). In the tables the column "BB" is the number of Branch-and-Bound nodes needed in order to solve the problem, the column "Calls" indicates the number of calls to the SPPTWCC subroutine during the execution of the algorithm, and "RD" is the number of routes deleted during all column deletion calls. In the test the trivial lower bound was enforced, and the new scheme for generating the S sets was used.

As expected removing unused columns after every Branch-and-Bound node lead to slower running times. Columns are often reused and as the algorithm is shifting from one branch of the Branch-and-Bound tree to another removing columns after every Branch-and-Bound nodes mean that columns have to be regenerated very often. It also leads to bad running times as the starting values of the dual variables are relatively bad resulting in generation of poor quality routes before better routes can be generated.

In none of the 27 instances does column deletion after every Branch-and-Bound node result in the best running time. In fact worse performance is only attained at instances 7 and 26. For the remaining 25 instances column deletion after every Branch-and-Bound node yields the worst running time consistently. In instance 7 and 26 running without column deletion performs worse (for instance 7 column deletion after every Branch-and-Bound node needs to check only a third of the Branch-and-Bound nodes as running without column deletion and therefore performs better).

Column deletion after every 5 and 10 Branch-and-Bound nodes is in all 27 instances outperformed by column deletion after every 20 Branch-and-Bound nodes. Column deletion after every 20 Branch-and-Bound nodes is typically around a factor 2 better then column deletion after every 5 and 10 Branch-and-Bound nodes.

All the figures clearly show that running column deletion after every 20 Branch-and-Bound nodes does lead to an algorithm with better performance. Comparing column deletion after 20 Branch-and-Bound nodes and the algorithm without column deletion reveals that only for instance 22 is column deletion after every 20 Branch-and-Bound nodes outperformed. Running without column deletion results in a running time 15% lower, but the real reason for the lower running time is that only 53 Branch-and-Bound nodes have to be checked, whereas the same number for column deletion

No.	Time (s)	BB	Calls
1	786.64	263	679
2	10114.34	2129	3759
3	3554.49	1057	2126
4	4906.45	1217	2453
5	240.55	95	307
6	363.02	101	325
7	605.27	107	345
8	2215.96	933	2030
9	514.46	441	983
10	788.73	465	1155
11	600.25	335	940
12	533.53	445	1069
13	910.79	381	821
14	1644.66	601	1283
15	1965.89	601	1322
16	3607.22	1091	2094
17	6550.68	1409	2688
18	1201.16	418	991
19	1396.38	479	1070
20	661.46	235	616
21	898.20	335	794
22	1525.23	53	297
23	1126.32	327	944
24	231.54	89	301
25	321.79	335	849
26	1863.87	1681	3162
27	148.35	79	339
Total	49259.23	15702	33742

Table 7.15: The reference algorithm not using column deletion.

	Reference		After every node			
No.	Time (s)	BB	Time (s)	BB	Calls	RD
1	786.64	263	1487.97	261	4832	647011
2	10114.34	2129	14685.05	2039	33383	4380848
3	3554.49	1057	5696.65	1033	18734	2492619
4	4906.45	1217	7715.56	1361	24634	3343864
5	240.55	95	772.74	129	2471	329819
6	363.02	101	662.60	101	2176	287514
7	605.27	107	407.30	33	880	112018
8	2215.96	933	2816.52	957	14336	1753939
9	514.46	441	772.64	329	4473	536201
10	788.73	465	2524.39	547	7526	898682
11	600.25	335	1513.64	339	5576	689900
12	533.53	445	1144.51	455	5868	660267
13	910.79	381	1736.54	375	7379	970836
14	1644.66	601	4993.27	1087	19164	2552661
15	1965.89	601	2574.49	411	8020	1101773
16	3607.22	1091	4780.54	839	15520	2155006
17	6550.68	1409	9092.14	1439	25972	3430415
18	1201.16	418	3078.53	631	11040	1449278
19	1396.38	479	2566.00	561	9796	1281868
20	661.46	235	1060.69	201	3908	528762
21	898.20	335	2032.96	415	6914	897783
22	1525.23	53	5906.70	51	1665	260219
23	1126.32	327	3358.26	349	5891	753148
24	231.54	89	529.83	115	2056	265309
25	321.79	335	853.52	359	4670	523920
26	1863.87	1681	1448.77	1673	28008	3116443
27	148.35	79	288.09	75	1183	137762
Total	49259.23	15702	83469.9	16165	276075	35557865

Table 7.16: After every Branch-and-Bound node the columns that has not been part of the basis are removed.

	Reference		After every 5 nodes			
No.	Time (s)	BB	Time (s)	BB	Calls	RD
1	786.64	263	716.66	263	2185	168315
2	10114.34	2129	7109.47	2179	17431	1363013
3	3554.49	1057	2926.65	1063	9015	712590
4	4906.45	1217	3731.19	1383	11651	939202
5	240.55	95	385.95	129	1094	80079
6	363.02	101	394.36	101	1010	79264
7	605.27	107	224.19	33	383	26244
8	2215.96	933	1229.94	951	7965	632561
9	514.46	441	287.30	331	2327	169713
10	788.73	465	922.89	557	4055	287399
11	600.25	335	542.02	339	2709	199472
12	533.53	445	442.63	447	3363	233533
13	910.79	381	897.82	379	3290	254528
14	1644.66	601	2320.79	1087	8845	678793
15	1965.89	601	1257.22	409	3480	282365
16	3607.22	1091	2499.16	907	7429	616438
17	6550.68	1409	4725.49	1551	13454	1062126
18	1201.16	418	1395.77	625	5052	382645
19	1396.38	479	1193.07	555	4377	337801
20	661.46	235	528.38	195	1606	132844
21	898.20	335	1047.50	451	3453	263802
22	1525.23	53	2768.40	53	722	67940
23	1126.32	327	1312.52	345	2967	244506
24	231.54	89	251.42	111	891	68049
25	321.79	335	319.59	339	2426	166064
26	1863.87	1681	1020.04	1615	15050	979063
27	148.35	79	209.08	79	734	48793
Total	49259.23	15702	40659.5	16477	136964	10277670

Table 7.17: After every 5 Branch-and-Bound nodes the columns that has not been part of the basis are removed.

	Reference		After every 10 nodes			
No.	Time (s)	BB	Time (s)	BB	Calls	RD
1	786.64	263	583.57	261	1696	97021
2	10114.34	2129	5582.53	2099	13184	753679
3	3554.49	1057	2271.20	1035	6794	393069
4	4906.45	1217	3476.04	1567	10327	606938
5	240.55	95	320.05	129	820	44099
6	363.02	101	343.81	101	773	43681
7	605.27	107	172.32	33	309	14384
8	2215.96	933	1021.63	961	6388	398196
9	514.46	441	232.84	333	1908	99997
10	788.73	465	728.08	565	3341	176089
11	600.25	335	413.88	337	2156	113968
12	533.53	445	356.31	449	2659	139004
13	910.79	381	724.93	377	2471	145769
14	1644.66	601	1824.17	1091	6776	379232
15	1965.89	601	1019.26	413	2735	161942
16	3607.22	1091	2122.55	899	5658	353346
17	6550.68	1409	3948.59	1553	10410	615854
18	1201.16	418	1191.33	635	3984	218404
19	1396.38	479	959.27	547	3327	185034
20	661.46	235	414.65	201	1248	73611
21	898.20	335	842.28	427	2509	141644
22	1525.23	53	2012.32	49	461	32799
23	1126.32	327	1054.33	313	2129	136102
24	231.54	89	213.86	111	692	39968
25	321.79	335	252.30	337	1956	100328
26	1863.87	1681	944.68	1621	12210	586640
27	148.35	79	162.01	75	562	28474
Total	49259.23	15702	33188.79	16519	107483	6079271

Table 7.18: After every 10 Branch-and-Bound nodes the columns that has not been part of the basis are removed.

	Reference		After every 20 nodes			
No.	Time (s)	BB	Time (s)	BB	Calls	RD
1	786.64	263	311.03	263	1367	54320
2	10114.34	2129	3011.84	2117	10712	443946
3	3554.49	1057	1237.72	1047	5413	224949
4	4906.45	1217	1287.24	1119	5731	231572
5	240.55	95	156.56	129	677	23867
6	363.02	101	167.23	101	589	23874
7	605.27	107	84.24	33	226	7310
8	2215.96	933	540.80	977	5311	241709
9	514.46	441	124.51	333	1523	58233
10	788.73	465	357.83	563	2702	102475
11	600.25	335	233.17	337	1786	69449
12	533.53	445	186.13	453	2193	83094
13	910.79	381	382.03	379	1912	78083
14	1644.66	601	983.68	1085	5403	212610
15	1965.89	601	537.24	411	2094	86644
16	3607.22	1091	1203.92	935	4622	202148
17	6550.68	1409	2212.00	1557	8296	350750
18	1201.16	418	626.24	629	3093	118743
19	1396.38	479	531.26	547	2619	104080
20	661.46	235	231.11	197	933	37129
21	898.20	335	454.83	431	1993	77595
22	1525.23	53	1808.42	163	1271	80373
23	1126.32	327	658.93	315	1776	81028
24	231.54	89	113.05	111	558	21127
25	321.79	335	134.07	333	1535	55965
26	1863.87	1681	519.75	1611	9687	330128
27	148.35	79	87.75	75	453	13330
Total	49259.23	15702	18182.58	16251	84475	3414531

Table 7.19: After every 20 Branch-and-Bound nodes the columns that has not been part of the basis are removed.

after every 20 Branch-and-Bound nodes is 163.

It is worth noting that column deletion after 20 Branch-and-Bound nodes outperforms the code not using column deletion totally by a factor 2.5, but not because fewer Branch-and-Bound nodes have to be checked. Whereas the algorithm without column deletion has to check a total of 15702 Branch-and-Bound nodes column deletion after every 20 Branch-and-Bound nodes needs additionally around 500 Branch-and-Bound nodes (note that approximately twice as many calls to the SPPTWCC subroutine are necessary). Gains are made because the checks on columns are drastically reduced. The effectiveness is underlined by the fact that only for instance 7 does column deletion after every 20 Branch-and-Bound nodes need significantly fewer Branch-and-Bound nodes. For instance 14 column deletion after every 20 Branch-and-Bound nodes outperforms the algorithm without column deletion with more than 50% although almost twice as many Branch-and-Bound nodes is needed.

7.8 Speeding up the column generation.

As described earlier column generation is done by solving the SPPTWCC. Each time the SPPTWCC function is called it either returns a number of columns or none indicating that the solution of the master problem is optimal. The further away from optimum the dual variables are the less they reflect the optimal solution and thereby generally increase the running time of the SPPTWCC usually generating routes of poor quality. We would therefore like to tune the SPPTWCC part of the algorithm to get results faster, but still maintain optimality.

7.8.1 Random selection.

The SPPTWCC code ranks the routes according to the most negative reduced costs using a dynamic programming formulation. Therefore the algorithm terminates when the last label is checked. The upper limit on the number of routes returned is controlled by a constant MAX_COLS_ITER.

The SPPTWCC algorithm returns all the routes with negative accumulated cost although at most MAX_COLS_ITER. But in order to return the best routes all labels have to be checked.

Instead of returning the route with the lowest reduced cost we could generate a certain number of routes (lets call it MAX_COLS_GEN) and select randomly from these a number of routes and return them. Now SPPTWCC can be stopped as MAX_COLS_GEN is reached and the appropriate number of routes can be returned to the master problem. The price is that we no longer return the best routes according to the dual variables, and thereby we risk not being able to lower the objective value as much as possible.

In many cases this is not a high price to pay. If only the route with the lowest reduced cost was returned, this would surely be in the basis as the optimal solution is returned from CPLEX, but as a number of routes are returned synergy might result in the route with the lowest reduced cost not being used in the basis at all. To test this we implemented it in our SPPTWCC code and used our test-bed on the setting shown in table 7.20

Test name	MAX_COLS_GEN	MAX_COLS_ITER
sp200300	300	200
sp200inf	∞	200
sp2030	30	20
sp2040	40	20

Table 7.20: Four test where made on random selection.

Note that problem *sp200inf* is different from running the ordinary code. In both cases the SPPTWCC algorithm checks all labels. The difference is that instead of selecting the 200 best we return 200 randomly selected routes. The last column in each of the following tables is the number of calls of the SPPTWCC subroutine where random selection of routes was used (if fewer routes than the upper limit was generated random selection was of course not used).

Comparing table 7.21 with table 7.22 the first observation is that the running times of the *sp200inf* configuration are consistently (in 21 out of 27

sp200300					
No.	Time	BB	No. of SPPTWCC	Routes generated	Random calls
1	577.40	173	503	9786	28
2	15512.60	2171	3869	19637	43
3	1555.75	493	1104	10701	27
4	6602.69	1345	2717	17617	36
5	302.49	97	331	8566	25
6	397.60	101	329	6866	19
7	206.52	33	175	6176	19
8	2582.86	965	2038	17868	30
9	339.08	337	743	8547	20
10	1263.08	521	1282	14302	27
11	744.30	335	922	12453	24
12	732.79	471	1102	11189	25
13	902.69	379	829	9030	21
14	1877.05	599	1304	11720	26
15	1645.50	451	1063	11626	29
16	3766.89	831	1622	12638	29
17	9172.49	1545	2943	17704	38
18	1262.23	395	901	11097	26
19	955.16	335	789	10577	27
20	338.93	111	331	8541	27
21	1092.16	335	763	10474	30
22	737.77	57	309	17109	64
23	1674.39	185	609	13397	36
24	265.51	99	300	7588	22
25	384.74	289	726	9157	21
26	2283.19	1647	3150	13670	15
27	288.76	79	331	8299	21

Table 7.21: Using random selection generating at most 300 routes, and selecting 200 randomly.

sp200inf					
No.	Time	BB	No. of SPPTWCC	Routes generated	Random calls
1	417.76	147	442	7455	17
2	12568.64	2083	3735	15838	20
3	1518.31	505	1100	10025	21
4	5675.27	1603	3016	13554	17
5	290.16	97	316	6998	15
6	405.93	101	311	6247	16
7	199.20	31	144	5164	14
8	2401.09	945	2036	17639	27
9	340.77	365	782	7641	14
10	1139.85	533	1337	13272	21
11	686.46	335	930	10492	16
12	647.42	469	1149	10400	18
13	1003.55	383	832	8457	20
14	1691.33	599	1306	10804	18
15	1454.35	421	956	9263	18
16	3100.94	763	1527	11437	22
17	6689.06	1469	2825	14257	19
18	1184.08	395	920	9893	19
19	864.81	333	786	8924	18
20	338.12	113	355	7292	19
21	870.34	321	741	8501	16
22	1541.25	55	303	11319	34
23	1151.30	185	577	11443	26
24	270.44	101	322	6915	16
25	341.66	289	725	7889	14
26	2565.32	1839	3479	13745	11
27	286.34	79	346	7379	12

Table 7.22: Using random selection. A full run of the SPPTWCC subroutine is made. Among the routes with negative accumulated cost 200 selected randomly are returned.

instances) better than those of the *sp200300* configuration. The difference is although not big. Beside instance 22, that is atypical of this test, we never reach a factor 2 in improvement on either side.

It is also noteworthy that only a small fraction of calls to the SPPTWCC subroutine use random selection. Returning at most 200 routes seems to yield good performance. This result is confirmed by experiments done by Kohl [Koh95]. But only a few of the calls to the SPPTWCC actually produce 200 or more routes, and therefore it is only for a fraction of the calls that random selection is being used.

Therefore in order to test the idea of random selection we need to have a higher fraction of calls to the SPPTWCC subroutine. We therefore lowered the number of routes being transfered from the SPPTWCC subroutine to the master problem from 200 to 20.

The table 7.23 and 7.24 shows the results of the tests *sp2030* respectively *sp2040*.

Generally the difference in running times and the number of Branch-and-Bound nodes needed to solve the problems is only marginally different. The *sp2040* configuration performed consistently better (24 out of the 27 instances) but in no instances is the difference significant.

The poor impact of the idea when applied to a high fraction of SPPTWCC calls combined with the normally low number of SPPTWCC calls where more than 200 routes with negative reduced cost are generated has lead us to not do further research on this idea. It should be noted that the tests show that when a number of routes are returned from the same call of the route generator, the routes are typically of the same quality.

During some of the preliminary tests of random selection the idea for the technique described in the next section came about. The basic idea is that if all routes are basically of the same quality it does not make sense to find the best routes if they are all of poor quality.

7.8.2 Forced early stop.

Along with the “standard tests”, we also ran a number of “breaking the limit” tests where we tried to solve problems from the Solomon test-cases

sp2030					
No.	Time	BB	No. of SPPTWCC	Routes generated	Random calls
1	551.79	173	786	8476	321
2	13330.06	2157	4293	17834	551
3	1554.96	491	1407	10456	386
4	6118.72	1397	3135	15265	471
5	328.35	97	577	7241	299
6	435.70	103	559	6616	265
7	204.86	33	354	5224	232
8	1792.16	939	2307	13228	404
9	285.26	357	933	6110	199
10	893.44	527	1482	9207	291
11	854.17	335	1147	8864	301
12	703.06	467	1315	8156	258
13	801.45	381	1068	7768	269
14	1729.08	607	1580	10711	355
15	1399.64	415	1231	9637	349
16	3086.46	771	1854	11240	384
17	7709.11	1457	3208	15531	487
18	1186.88	415	1227	9584	340
19	869.80	335	1053	9177	343
20	354.06	113	591	7248	302
21	1163.17	361	1131	9533	365
22	1375.99	55	914	15249	700
23	4713.95	179	809	8394	332
24	301.06	99	574	7456	318
25	436.79	291	914	6870	233
26	1580.12	1697	3372	11141	263
27	424.36	79	564	6750	271

Table 7.23: After generating 30 routes 20 are randomly selected and returned.

sp2040					
No.	Time	BB	No. of SPPTWCC	Routes generated	Random calls
1	524.46	173	746	8018	315
2	12011.30	1961	3989	16984	525
3	1444.13	487	1381	10010	355
4	6761.63	1615	3418	15218	441
5	310.83	97	556	7269	301
6	381.32	99	529	6191	253
7	201.33	33	358	5149	217
8	1616.13	937	2307	12871	396
9	270.47	361	943	6155	199
10	951.46	593	1625	9668	295
11	768.12	335	1139	8581	288
12	577.81	465	1268	7599	234
13	779.86	377	1028	7425	263
14	1610.80	589	1579	10446	340
15	1270.35	413	1219	9318	336
16	2812.46	757	1765	10583	365
17	6802.40	1479	3168	14378	430
18	1143.25	415	1214	9265	325
19	906.42	343	1092	9233	338
20	347.04	113	605	7258	301
21	996.69	335	1073	8943	337
22	1224.50	57	838	14049	646
23	4261.81	173	806	8561	337
24	302.23	99	563	7027	294
25	425.59	289	913	6685	231
26	1536.93	1665	3388	11276	259
27	431.53	79	548	6437	259

Table 7.24: After generating 40 routes 20 are randomly selected and returned.

not solved to optimality previously (these results are show later).

One of these tests highlighted an unpleasant "feature" of the code. The output from our sequential algorithm for instance R203 with 25 customers is shown in figure 7.10. As the figure shows 3 routes are needed to service the 25 customers. The number in the round brackets indicates the column number of the route in the set partitioning formulation, while the number in the square brackets is the length of the route times 10. Hereafter follows a couple of lines of statistics of the execution.

As can be seen, the majority of the running time is used in the root node. Furthermore one of the calls to the SPPTWCC functions uses over 4400 seconds. This is a major drawback considering our efforts to develop an efficient parallel code. It became essential to cut down the time used in the root node. The SPPTWCC function is based on dynamic programming, and in a effort to cut time the execution could be stopped before the underlying search tree is fully investigated. Of course the execution can only be stopped when at least one route has been identified. Otherwise the set partitioning would remain unchanged and consequently the dual variables would remain unchanged, which would result in an identical execution of the SPPTWCC function.

The code for the SPPTWCC function was therefore changed. Two constants LIMIT and MIN_COLS_PER_ITER was introduced. The code has always used a constant called MAX_COLS_ITER which is the maximum number of routes that are returned, now MIN_COLS_ITER is the minimum number of routes that should be generated before we prematurely abort the generation of labels. Note that this is not necessarily the same routes that would be returned if the SPPTWCC function was run optimally and the MIN_COLS_ITER best routes was returned. The constant LIMIT is the number of labels that have to be generated before we think of prematurely stopping the SPPTWCC code. Based on the tracing from the tests mentioned earlier the values LIMIT and MIN_COLS_ITER were chosen manually.

In order to gain more knowledge on the properties of forced early stop 3 instances were selected for initial trials. R101 with 100 customers represented the easy problems as the customers in R101 have relatively small time windows and the demands limit the routes to at most 10 customers.

```
---------- Solution
Problem R203 with 25 customers is solved
The solution is the following routes:

( 6648) [1533] d - 2 - 15 - 23 - 22 - 21 - 4 - 25 - 24 - 3 - 12 - d

( 6946) [1041] d - 6 - 5 - 8 - 17 - 16 - 14 - 13 - d

( 9688) [1365] d - 18 - 7 - 19 - 11 - 20 - 9 - 10 - 1 - d

---------- Statistics
This program ran on serv3 (hp9000s700).
Total execution time                     13483.83 seconds
         (Solving root 13249.63 seconds)
Time used in separation                     0.29 seconds
         Cuts generated           2
Accumulated time used in calls of SPPTWCC 13332.516 seconds
Time used in largest single SPPTWCC call  4447.04 seconds
Branching nodes examined 43 (Veh 1, Arc 20, TW 0)
        (hereof 0 where not feasible)
No of calls to SPPTW 281, Routes generated 21250
Max no of columns selected per SPPTW 200
No of multiple customers deleted explicitly 0
IP value 3914
RP value 3816.250
LP value 3798.818
-------------------------------------------------------
```

Figure 7.10: The result of solving R203 with 25 customers with our algorithm.

R103 with 50 customers represented the problems that are a bit more difficult to solve, while R202 represented the really tough problems (in R202 the time windows are relatively wide and the routes can contain up to around 30 customers). For these problems the number of labels used in each execution of the SPPTWCC code was recorded. Table 7.25 show the results:

Problem	Cust.	SPPTWCC calls	Routes made	No. of labels Min	Max	|Average|
R101	100	142	2098	1365	6005	2189
R103	50	178	2228	1350	20503	2573
R202	25	117	2009	422	275813	9631

Table 7.25: Characteristics for the "normal" trace of the 3 selected problems.

All three problems have the same feature with a number of relatively time-consuming executions in the start. This is because our initial setting of the dual variables is far away from the optimal dual values. Hence a number of "heavy-duty" calls is necessary before the values are good enough to reduce the size of the underlying search tree.

The results are depicted in the tables 7.26 to 7.28.

R101 – 100 customers					
LIMIT	–	4000	4000	2000	1000
MIN_COLS_ITER	–	10	1	1	1
Running time (total)	19.94	16.22	16.37	16.11	72.27
Running time (root)	7.47	5.84	5.60	6.13	62.66
No. of nodes	15	15	15	15	15
SPPTWCC calls	88	89	89	117	2352
No. of routes	3970	3479	3479	3375	2555

Table 7.26: Testing prematurely stop of SPPTWCC on "easy" problems.

These preliminary results were surprisingly positive. Most noteworthy is the phenomenal reduction in running time for R202 with 25 customers –

R103 – 50 customers						
LIMIT	–	10000	5000	5000	2000	2000
MIN_COLS_ITER	–	10	10	1	10	1
Running time (total)	41.27	34.38	26.81	27.30	33.75	61.62
Running time (root)	14.28	4.59	3.21	3.27	9.17	43.08
No. of nodes	39	45	43	43	43	41
SPPTWCC calls	158	168	157	157	292	719
No. of routes	5701	5743	5159	5159	4550	4507

Table 7.27: Testing prematurely stop of SPPTWCC on "medium" problems.

R202 – 25 customers					
LIMIT	–	50000	50000	10000	5000
MIN_COLS_ITER	–	10	1	1	1
Running time (total)	3322.17	256.34	227.46	13.278	7.97
Running time (root)	3316.75	251.44	222.438	9.43	4.34
No. of nodes	5	5	5	5	5
SPPTWCC calls	59	63	63	59	58
No. of routes	6913	6921	6921	6174	5983

Table 7.28: Testing prematurely stop of SPPTWCC on "tough" problems.

from over 3300 seconds to a mere 8 seconds (more than a factor 400!).

The running time for solving the root node is the key to understanding the greatly improved performance. For R202 with 25 customers the difference between the overall running time and the time spent solving the root node is roughly around 5 seconds, which means that the time is gained entirely in solving the root node. Clearly the less reliable the dual variables are the more one can gain from stopping early. The drawback of stopping early is lack in route quality, but as the quality of the dual variables is low there is practically nothing to loose. And even considering the "easy" problem an improvement is made in the running time indicating that stopping early is an advantage for all problems.

Now we can return to our initial example: R203 with 25 customers. As a test we ran it with LIMIT = 5000 and MIN_COLS_ITER = 1. The result can be seen in figure 7.11.

```
---------- Solution
Problem R203 with 25 customers is solved
The solution is the following routes:

( 5120) [1041] d - 6 - 5 - 8 - 17 - 16 - 14 - 13 - d

( 5778) [1533] d - 2 - 15 - 23 - 22 - 21 - 4 - 25 - 24 - 3 - 12 - d

( 9416) [1365] d - 18 - 7 - 19 - 11 - 20 - 9 - 10 - 1 - d

---------- Statistics
This program ran on serv3 (hp9000s700).
Total execution time                       147.55 seconds
         (Solving root 8.56 seconds)
Time used in separation                      0.26 seconds
         Cuts generated           2
Accumulated time used in calls of SPPTWCC   14.65 seconds
Time used in largest single SPPTWCC call     0.25 seconds
Branching nodes examined 41 (Veh 1, Arc 20, TW 0)
        (hereof 0 where not feasible)
No of calls to SPPTW 273, Routes generated 19396
Max no of columns selected per SPPTW 200
Prematurely exiting of SPPTWCC enables.
        LIMIT is set to 5000.
        MIN_COLS_ITER is set to 1.
No of multiple customers deleted explicitly 0
IP value 3914
RP value 3816.250
LP value 3798.818
-------------------------------------------------------
```

Figure 7.11: The result of solving R203 with 25 customers with our algorithm improved the early stopping criteria.

Recall that before the running time was over 13000 seconds, now we are down to around 150 seconds - a reduction by a factor 91. Note how the time spent in the most time consuming SPPTWCC call is reduced from 4447.04 seconds to 0.25 seconds.

To test our algorithm we tried to solve instances from the R2, C2 and RC2 test sets. Their large time windows make even instances with few customers difficult to solve. The large time windows result in many feasible routes thereby slowing down the SPPTWCC subroutine. For the few instances where we know the running time of the original code it is reported in the first column of table 7.29, where the results are reported. Every instance of R2, C2 and RC2 using the 25 and 50 first customers were run for 30 minutes before execution was stopped. An "**R**" in the second column indicates that execution was stopped while working on the root node of the Branch-and-Bound tree, whereas a boldface integer presents the Branch-and-Bound node that the algorithm was working on as the algorithm was stopped.

The solutions of all the instances that were solved are presented in appendix B. Especially for large time windows forced early stop results in a substantial decrease of running time. We were able to solve 16 of the demanding R2, C2 and RC2 problems. Comparing with the running times for the algorithm without forced early stop the improvement in performance is huge.

As a final test we tested forced early stop on the 27 instances of our testbed. The results are presented in table 7.30.

As can be seen from table 7.30 forced early exit is not the answer to all our problems. The running times does not decrease as much as in the cases of R2, C2 and RC2. The are in fact instances (9 out of 27) where the running time increases when we use forced early stop. For some of these (instance 16 and 17) the larger running times can be explained by an increase in the number of Branch-and-Bound nodes that have to be checked. But this can not explain the bad performance in instance 9 where we get a worse running time even though we have to explore fewer Branch-and-Bound nodes. Worse performance is an indication that the construction of good quality routes is stopped before all routes have been explored. Then a new call of the SPPTWCC subroutine has to generate an almost identical set of labels to reach the position where the previous call of the SPPTWCC subroutine was aborted. This suggests that "quality control" should be

Instance	25 customers		50 customers	
	Time (before)	Time (now)	Time (before)	Time (now)
R201	14.82	1.92		10.73
R202	3322.17	7.97		272.92
R203	13249.63	147.55	> 10000	**R**
R204		**3**		**R**
R205	325.99	16.69		**93**
R206		**12**		**R**
R207		**3**		**R**
R208	> 10000	**R**		**R**
R209		**3**		**3**
R210		**18**		**R**
R211		**3**		**R**
C201	48.57	3.12	> 10000	208.74
C202	107.93	12.9		**R**
C203	1411.91	33.17		**R**
C204		**R**	> 10000	**R**
C205	28.55	7.66		**R**
C206		21.60		**R**
C207	> 10000	149.53		**R**
C208		80.28		**R**
RC201	8.52	1.29	70.83	47.30
RC202		**35**		**R**
RC203		**14**		**R**
RC204		**R**		**R**
RC205	116.45	72.16		**R**
RC206		**4**		**R**
RC207		**3**		**R**
RC208		**R**		**R**

Table 7.29: The results of our half-hour test of 25 and 50 customer problems from the test sets R2, C2 and RC2 (using LIMIT = 5000 and MIN_COLS_ITER = 1).

	No Forced early stop		Forced early stop	
No.	Time	BB	Time	BB
1	809.63	263	553.89	215
2	10117.10	2129	10055.56	2065
3	3453.24	1057	3629.94	1059
4	4788.94	1217	4008.44	1383
5	239.27	95	205.57	95
6	360.18	101	342.68	101
7	596.75	107	496.37	105
8	2130.17	933	1811.92	931
9	481.12	441	499.46	425
10	760.55	465	794.83	431
11	582.26	335	564.41	335
12	519.02	445	743.51	441
13	891.43	381	764.93	379
14	1651.66	601	1402.66	591
15	1947.58	601	1783.73	597
16	3549.38	1091	4436.76	1273
17	6476.61	1409	6783.94	1517
18	1197.65	418	1059.65	397
19	1260.22	479	1190.19	493
20	663.42	235	687.86	233
21	904.23	335	827.77	339
22	1547.66	53	1479.62	51
23	1119.12	327	921.89	267
24	233.53	89	176.52	87
25	318.73	335	322.54	333
26	1719.57	1681	1812.13	1651
27	148.57	79	122.61	75

Table 7.30: Testing forced early stop on our standard test-bed (using LIMIT = 5000 and MIN_COLS_ITER = 1).

included in the decision whether to abort the current call of SPPTWCC or continue.

On all the time consuming instances forced early stop does although result in a decrease in running time. The accumulated picture of table 7.29 and 7.30 shows that forced early stop is an effective technique to reduce running time. The efficiency is significantly larger on instances with large time windows, but using forced early stop in instances with small time windows does only in a few instances give worse running times. This suggests that the involved constants *LIMIT* and *MIN_COLS_ITER* should be determined dynamically depending on geography, time windows etc.

Chapter 8

Parallel computational experiments

The experimental tests of the parallel VRPTW algorithm were carried out on the IBM SP2 at UNI•C. Even though parallel jobs using up to 32 processors are possible (with a special permission even up to 64 processors are allowed) the number of CPLEX licenses sets an upper bound of 10 processors on the experiments. On the IBM SP2 the installed version of CPLEX is 6.0.1. This may result in differences from the results obtained for the sequential experiments.

Another difference to the sequential experiments is that only a certain amount of CPU hours have been available. In order to run experiments on the IBM SP2 one has to apply for CPU hours and then it is a question of having to fit your experiments according to the amount granted.

When running many tests it is very difficult to estimate how many CPU hours are needed. We therefore used our time with care. Each configuration of the parallel program has hence only been run once. A larger number is generally preferable, but the minimum number was chosen to ensure that all tests could be made with the amount of CPU time at our disposal.

All programs are written in C and MPI is used for the communication framework of the program.

8.1 The basic parallel program

First the basic parallel program as described in chapter 6 was tested to measure the performance. We have run 4 relatively easy instances in 4 setups: sequentially, and in parallel using 4, 6 and 8 processors. The four instances are: R104 with 50 customers, S1053 with 100 customers, S105-3-4 with 100 customers and S1055-2 with 100 customers.

The running times and the number of Branch-and-Bound nodes that were used to solve the instances are shown in table 8.1.

	No. of processors			
Instance	1	4	6	8
R104	1170.80	780.43	701.07	705.24
	189	278	245	282
S1053	356.36	146.55	143.49	151.07
	113	114	131	189
S1055-2	1167.09	341.17	267.64	224.87
	403	448	483	479
S105-3-4	1087.40	320.62	251.61	224.14
	331	335	390	419

Table 8.1: Performance of the basic version of the parallel program. The first line is the running time in seconds, and the second line is the number of Branch-and-Boundnodes used to solve the instance.

Note that with 2 exceptions the number of Branch-and-Bound nodes are increasing as more processors are used to solve the instance. This is a common feature of parallel best-first Branch-and-Bound. As more processors are assigned to the task they do not always have the same global upper bound, as a new global upper bound is found by one processor it has to be distributed to the others before they can use it to fathom the Branch-and-Bound tree.

In table 8.2 we present the speedup calculated on the basis of table 8.1. Speedup is calculated as $s_n = \frac{t_1}{t_n}$ where t_1 is the running time of the sequential algorithm and t_n is the running time of the parallel algorithm using n processors. Ideally we would like the speedup to be n (the parallel program using n processors is n times faster than the sequential program), but we would be satisfied with less. A linear growth in speedup with a coefficient less than 1 would also be acceptable.

	No. of processors		
Instance	4	6	8
R104	1.50	1.67	1.66
S1053	2.43	2.48	2.36
S1055-2	3.42	4.36	5.19
S105-3-4	3.39	4.32	4.85

Table 8.2: The speedup achieved by the different test instances.

To get an overview of the speedup we have plotted the numbers in figure 8.1.

Instance R104 seems to behave very badly. Taking a closer look at the statistics of the R104-runs reveals the reason. In our parallel program the master processor has to generate n "live" Branch-and-Bound nodes before the parallel phase can be initiated by sending 1 Branch-and-Bound node to each processor. In R104 generating 4 "live" subspaces requires 550.22 seconds of computing. So almost half way through the time used by the sequential program, the parallel program is still running like a sequential program. The problem here is especially solving the root node, where the first calls to the SPPTWCC subroutine are very costly.

After the initial phase is finished the parallel program uses approximately 205 seconds on each processor (a total of 821.83 seconds). For 6 processors a total of 797.32 seconds are used in the parallel phase and for 8 processors the amount is 1071.41 seconds. So apart from the test using 8 processors the amount used in total in the parallel phase is almost identical, which disregarding the bad initial phase suggests good speedup. In table 8.3 we have calculated the totals of all the parallel phases.

The accumulated totals of the parallel phases underline the good results

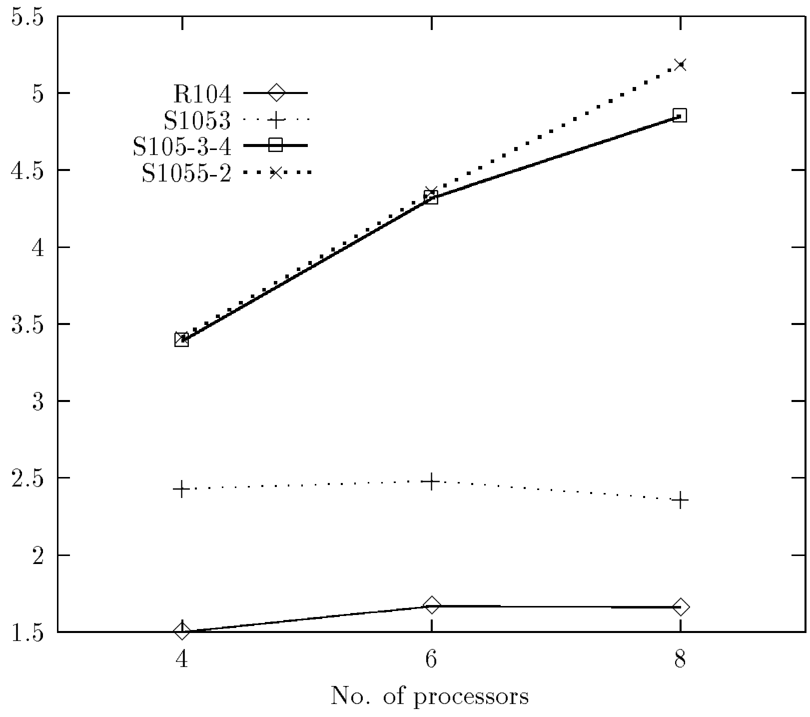


Figure 8.1: Plot of the speedup.

	No. of processors		
Instance	4	6	8
R104	821.83	797.32	1071.41
S1053	458.27	640.82	852.80
S1055-2	1264.72	1432.09	1537.02
S105-3-4	1151.59	1298.59	1471.47

Table 8.3: Accumulated running time in the parallel phase.

obtained for the S1055-2 and S105-3-4 instances. It also shows that for R104 the parallel algorithm also behaves quite well as soon as the parallel phase is started.

Instance S1053 is clearly too small for 6 and 8 processors. Nice results are although obtained for 4 processors.

The number of Branch-and-Bound nodes solved on each processor clearly demonstrates the good performance of our loadbalancing scheme as shown in table 8.4.

	No. of processors	
Instance	4	6
R104	82 61 68 67	56 39 42 35 37 36
S1053	41 26 21 26	34 23 18 19 20 19
S1055-2	126 113 101 108	96 80 82 78 76 71
S105-3-4	90 80 78 87	73 58 62 69 63 65
	8	
R104	49 31 40 31 34 29 38 30	
S1053	36 26 22 20 18 25 25 18	
S1055-2	75 64 54 62 62 64 46 52	
S105-3-4	63 50 55 57 54 49 47 44	

Table 8.4: The number of Branch-and-Bound nodes solved by each of the processors.

Note that the first number in each field in table 8.4 is the number of Branch-and-Bound nodes processed by the master processor. Included in this number is also the number of Branch-and-Bound nodes processed in the "sequential" part of the parallel algorithm, that is, before the parallel phase is started. Generally the numbers in table 8.4 are highly satisfactory. The show that a very balanced load can be maintained by only using estimates on the load of the neighbours and only using local information, that is, information from the two neighbours in the ring topology. For the parallel algorithm for the VRPTW it looks as if a more global load balancing strategy is not needed.

8.2 Strategy for selection of subspaces

In our initial implementation unsolved subspaces are transferred from one processor to another when $n - n' > \frac{n}{2}$, where n is the heap size of the sending processor and n' is the *estimate* of the heap size of the receiving processor. Then the best $\max\{\frac{n-n'}{3}, 1\}$ subspaces are transferred. Transferring the best subspaces is likely to result in a significant loss of quality when solving the next subspace.

Two approaches could resolve the undesired properties. One way would be to send subspaces of different quality with respect to bounding value. We still send $\max\{\frac{n-n'}{3}, 1\}$ subspaces from the sending processor to the receiving processor, but instead of only taking the best subspaces every second subspace from the heap is chosen. The other subspace remain in the heap of the sending processor.

A second approach would be to only transfer one subspace – the top Branch-and-Bound node from the heap. This still leaves subspaces of high quality for the sending processor. Furthermore this approach is faster than the other approaches as only one subspace has to be packed for transmission and the receiving processor only has to unpack one subspace.

Both new approaches are implemented in our parallel algorithm, and tested on the instances previously used. Table 8.5 shows the performance of the two new approaches (HALF is the approach where we take $2 \cdot \max\{\frac{n-n'}{3}, 1\}$ off the heap and transmit every second of the subspaces to the neighbouring processor (the remaining are put back on the heap) and BEST is the approach where only the top element on the heap is transmitted).

Selection strategy	No. of processors 1	4	6	8
R104				
Ref.	1170.80	780.43	701.07	705.24
	189	278	245	282
Best	—	755.77	687.20	691.76
		243	203	287
Half	—	721.46	690.80	675.12
		202	245	254
S1053				
Ref.	356.36	146.55	143.49	151.07
	113	114	131	189
Best	—	146.77	167.55	147.36
		114	161	169
Half	—	226.35	145.13	107.54
		122	160	152
S1055-2				
Ref.	1167.09	341.17	267.64	224.87
	403	448	483	479
Best	—	324.08	266.95	242.21
		413	460	494
Half	—	291.34	221.09	178.14
		406	436	440
S105-3-4				
Ref.	1087.40	320.62	251.61	224.14
	331	335	390	419
Best	—	338.42	269.79	262.38
		364	387	460
Half	—	340.91	260.07	235.61
		368	391	424

Table 8.5: Comparison of the performance of the selection strategy compared with the basic version of the parallel algorithm (Ref).

The running times and the number of subspaces that were necessary are quite similar between the three strategies. All three strategies perform alike and they all have the same problem with the smallest instance (S1053). HALF does however exhibits better scalability performance. HALF seems to be able to cope better with an increasing number of processors, so that all processors for the most of the time are doing good work. Therefore this exchange strategy is used from now on.

8.3 Exchange of "good" routes

In the basic model we distribute unsolved subspaces among the processors at the start of the parallel phase. Each processor still has to generate all the routes needed by itself. It seems obvious that the processors could help each other by exchanging "good" routes.

When the slave processors start working they start with the *depot* - i - *depot* routes as the only routes in the set partitioning problem. Thereby, every slave processor is facing the same problems as the master processor initially does, namely dual variables of poor quality. Solving the first subspace on each processor may therefore take almost as much time as it took to solve the root node, which is always the most time consuming Branch-and-Bound node to solve. To get an effective parallel algorithm we have to ensure that the process of solving the first subspace on each slave processor is made more effective using information already generated. One way would be to use the routes already generated in the initial phase of the parallel algorithm, that is, utilize the routes generated by the master processor.

This can be accomplished in a number of ways:

1. Send routes that have been part of the basis of the optimal relaxed solution.
2. Send routes that have been part of the basis of a solution to a subspace.
3. Send routes that are or have been part of a global upper bound.

Here (2) is really an extension of (1). The idea of sending good routes to all other processors is to exploit the "good" routes already found by the master processor. This will give the slave processors a better start instead of starting with only the generic routes: depot - i - depot.

Generally (2) will result in a significant number of good quality routes. As the problems solved by the slave processors all have constraints imposed by the branching operations there should routes enough to transfer to guarantee that a fair fraction of them are feasible with respect to these constraints. Therefore (1) and (3) are excluded as they only result in a limited number of routes (at most equal to the number of customers in the problem). Indeed (3) might even result in no routes passed along if no global upper bound is found in the initial phase (and that is very likely to happen for large problems).

We have chosen to implement (2). In order to keep track of which routes qualify for transmission we introduce a "basis" bit for each route. Initially the basis bit is 0, but as soon as a route has been part of the basis of a solution to a subspace it is set to 1.

When selecting routes for transmission we start from the beginning of the array of routes. As subspaces are processed according to the bounding-value of their parent we generally select routes from the low-valued subspaces first by running though the array of basis bits from route 1 and upwards.

When all routes with a 1 in the basis bit is transfered to the buffer or as soon as the buffer is full (the buffer is 10000 bytes which leaves room for plenty of routes) the routes are broadcasted to the slave processors. Note that we broadcast the routes as a sequence of customer visits, not as the coefficients from the set partitioning matrix. Therefore each slave processor will have to generate the coefficients for the set partitioning problem and generate the coefficients originating from the cuts.

In table 8.6 we have compared the running time and the number of Branch-and-Bound nodes needed with the parallel algorithm using the Half selection strategy and broadcasting "good" routes initially Initial with the parallel algorithm using the Half selection strategy only.

Unfortunately the results are not very clear. As the routes transfered from the master processors to the slave processors may result in a different

	No. of processors		
	4	6	8
R104			
HALF	721.46	690.80	675.12
189	202	245	254
INITIAL	688.10	644.30	633.97
189	196	199	192
S1053			
HALF	226.35	145.13	107.54
113	122	160	152
INITIAL	144.84	132.63	124.96
113	114	137	162
S1055-2			
HALF	291.34	221.09	178.14
403	406	436	440
INITIAL	357.23	226.06	202.37
403	462	407	459
S105-3-4			
HALF	340.91	260.07	235.61
331	368	391	424
INITIAL	334.49	227.86	191.71
331	359	367	427

Table 8.6: Comparison of the performance of the selection strategy compared with the basic version of the parallel algorithm (REF). The number of Branch-and-Bound nodes used by the sequential algorithm is displayed emphasized in the first column.

Branch-and-Bound tree than in HALF the difference in the total number of Branch-and-Bound checked can be quite large. It is not possible without further tests to draw any conclusions.

The idea of exchanging routes can be taken one step further. Together with a batch of subspaces we can send a set of "good" routes. When routes where broadcasted after the initial phase "good" routes are routes that have been part of the basis of an optimal solution to a subspace.

So after the subspaces have been moved to the transmission buffer we fill the rest of the buffer with good routes. When we insert the new routes in the set partitioning after the initial broadcast of routes they can be inserted straight away without any problems. Now the receiving processor may already have generated some of the routes itself. Either we allow for a route to be in the set partitioning problem more than once, or we check routes received from other processors before inserting them. Checking before insertion of a route results in a significant amount of extra work as thousands of routes has to be compared repeatedly. If we do not check and the sending processor keep track of which routes have been sent to which neighbour and additionally only transfer routes generated by the given processor itself, at most 3 copies of the same route are in the set partitioning problem (the one generated by the processor itself and one copy from each neighbour).

If we use the column reduction idea proposed and implemented in section 7.7 the worse case of 3 columns will only exist for a limited amount of Branch-and-Bound iterations, namely between two column reduction operations. Due to the not very clear results of the scheme for an initial distribution of routes we have decided not to implement this idea.

Chapter 9

Conclusions

This chapter summarizes the contributions of the thesis and the conclusions. A number of interesting ideas for further research are also presented.

9.1 The Road Ahead

A lot of interesting topics regarding the sequential VRPTW problem and related areas are still open for research.

9.1.1 Forced early of stop the the 2-path separation algorithm

In a small study of our new scheme for constructing S sets we printed out the number of each cut. That is, for each candidate set S that was checked for feasibility with respect to capacity and "feasibility TSPTW" was assigned a consecutive number in the order they are generated.

By printing out the numbers of the candidate sets that actually were confirmed as being cuts we got a trace of the execution of the 2-path separation algorithm. For example for one instance the 4 cuts were generated by candidate set number 74, 106, 130 and 286, but additionally 9689 candidate sets needed to be checked before the separation algorithm was finished. In another example the candidate sets 65, 71, 1571, 1580, 2472, 2486 and 5357 were identified as cuts but after generating the last cut, a further 3937 candidate sets had to be checked before the separation algorithm terminated.

The picture illustrated by the two examples above was repeated for every trace of the separation algorithm we made. After the last cut was found a large number of sets still needed to be checked by the separation algorithm.

As we do not know when the last cut is generated a priori the ideas of when to stop will always be based on estimates derived by information gathered during the early part of the execution of the algorithm.

It would be interesting to see if information obtained during the generation of cuts could be used to stop the separation algorithm earlier. One idea would be to maintain an upper limit of how many candidate sets now being identified as cuts are allowed between two cuts. This value could then be adjusted during the run of the separation algorithm.

9.1.2 Describing and implementing new cuts

Work on cuts for the VRPTW has been scarse. In [Koh95] the first cuts specifically for the VRPTW are introduced and implemented. In [Kon97] two new sets of cuts are added to the list, but still work on cuts for the VRPTW are rare. The cuts by Kontoravdis presented in [Kon97] can not be used in a set partitioning formulation. Efforts should be made to try to transfer the cuts of Kontoravdis to the set partitioning formulation.

Using established techniques and a thorough study of past results in related research areas should make it possible to describe and implement new classes of cuts.

A class of "infeasible path elimination" constraints are presented by Ascheuer et al. in a preprint [AFG97] for the ATSP with time windows. No experimental results are reported in the paper but it is stated that the new cuts outperforms alternative formulations on some classes of problem instances. Implementing these cuts for the VRPTW would be interesting.

9.1.3 Redesign of the 2-path cuts

Solving the R2, C2 and RC2 problems our algorithm did often hit the upper limit of the number of customers allowed to be in a candidate set. The limit (presently set to 15) is introduced in order to keep down the running time of the TSPTW algorithm. In order to be able to take advantage of the 2-path cuts even for the more difficult instances (larger time windows and/or larger vehicle capacity) a redesign of the separation algorithm is worth considering.

9.1.4 Heuristics based on the column generation technique

In our implementation of the algorithm for the VRPTW an upper limit on the number of columns in the set partitioning problem is imposed. As Kohl in [Koh95] we allow at most 50000 columns in the set partitioning problem. Presently the algorithm stops as the upper limit is reached. It would be interesting to use the information available (the 50000 columns, obtained global upper bound etc.) in designing a heuristic.

For the set covering problem a number of papers have been published among those [Kwa93, Ca93, Wed95]. The set covering problem is extensively used in a number of airline related problems (for example the algorithm developed by Wedelin in [Wed95] is used in the CARMEN system for airline crew scheduling).

Using the columns in an effective heuristic (previous similar or related approaches have been made in [CJR81, Kri95, Tai96]).

9.1.5 Advanced branching methods

In the research done so far on exact methods the branching-criteria has not been the focus of attention. Present ideas are simple and reflect the fact that the main efforts of researchers so far have been on other aspects of solving the VRPTW. More intelligent ways of branching include exploiting information on geography, time windows etc. These areas of research are open for new ideas. Initial promising attempts were made in collaboration with professor David M. Ryan from the University of Auckland.

Our present arc branching strategy is based on the accumulated arc flow. One way to strengthen the arc selection might be to compute an estimate of the additional costs of using a given branch by using more information than only the accumulated arc flow. We now consider the arc (i, j) and assume that the flow on the arc is fractional i.e. $0 < f_{ij} < 1$. In the method we propose here we will try to estimate the additional cost by repairing mass-imbalance locally. Now if we where to raise the flow of (i, j) from its present value of f_{ij} to 1 the customers i and j would have a mass-imbalance. Too much flow $(1 - f_{ij})$ is flowing out of i and to much flow $(1 - f_{ij})$ is flowing into j.

The excess flow at customer i can be removed by deleting customer i from the remaining routes currently servicing customer i, that is, let these routes go directly from the predecessor of i to the successor of i. In the same way we manage the excess of in-flow to customer j. The flow added to the (i, j) to get to flow 1 is supplied by the route *depot* - i - j - *depot*. The changes in flow is depicted in figure 9.1. So our estimate of the cost of choosing to raise the flow of arc (i, j) to 1 is:

$$\begin{aligned}(1 - f_{ij}) \cdot (c_{0i} + c_{ij} + c_{j0}) \quad &+ \quad \textstyle\sum_k (c_{k'k} - c_{k'i} - c_{ik}) \cdot f_{ik} \\ &+ \quad \textstyle\sum_l (c_{l'l} - c_{l'j} - c_{jl}) \cdot f_{jl}\end{aligned}$$

Now if we want to estimate the cost of lowering the flow on (i, j) to 0 we estimate the cost of "redirecting" the vehicle driving the present route. Instead of going from i to j we send it from customer i to customers j', the successor of customer j on the route. Thereby we save $f_{ij} \cdot (c_{ij} + c_{jj'})$ but adds $f_{ij} c_{ij'}$ to the cost. Now customer j is serviced by $(1 - f_{ij})$ vehicle and in order to restore the mass balance we use the route *depot* - j - *depot* to supply the lack in flow. This results in the following estimate:

$$f_{ij} \cdot (2 \cdot c_{0j} - c_{ij} - c_{jj'} + c_{ij'}).$$

Instead of dropping customer j from the route a symmetric situation occurs if we leave out customer i from the route. As an estimate we choose the smallest one:

$$\min\{f_{ij} \cdot (2 \cdot c_{0j} - c_{ij} - c_{jj'} + c_{ij'}), f_{ij} \cdot (2 \cdot c_{0i} - c_{ij} - c_{ii'} + c_{i'j})\}$$

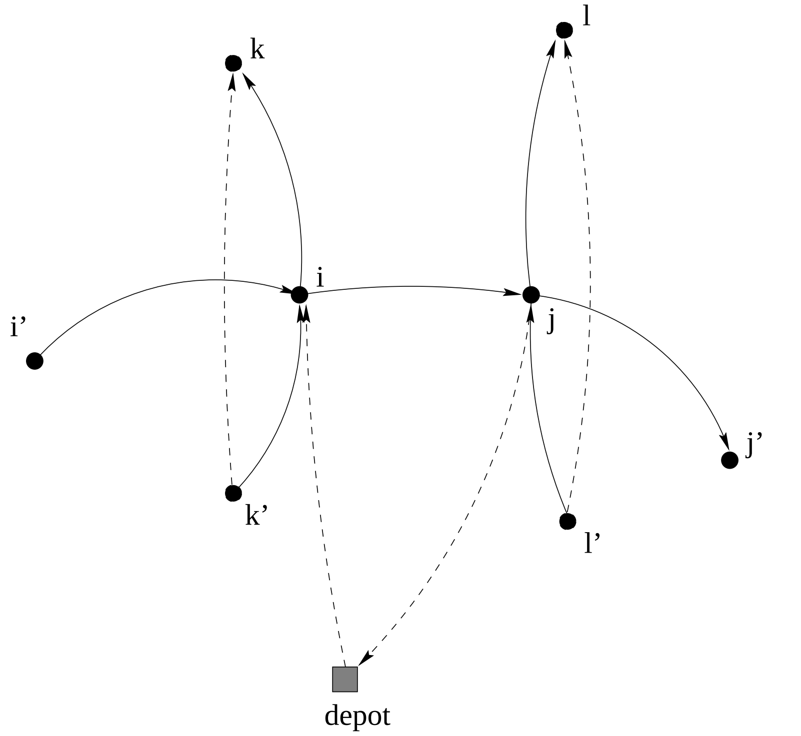


Figure 9.1: The estimate on setting the flow between i and j is established by redirecting the other routes with flow pass i and j and supply the additional flow $(1 - f_{ij})$ from the depot.

The situation for calculating the estimate when customer j is removed from the route is depicted in figure 9.2.

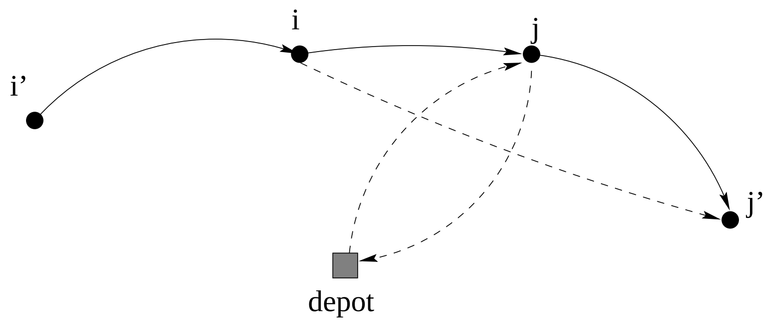


Figure 9.2: The estimate of the cost when setting the flow between i and j is calculated by removing the cost of the flow on the arcs (i, j) and (j, j') and adding the flow-cost on the dashed arcs.

This new rule for selecting the arc to branch on requires more computing than the present arc branching method. Instead the more thorough estimate hopefully results in better quality branching. Our preliminary tests was ambiguous. On some instances the desired effect was produced, while we did not receive better running times on a number of other instances. Further and more thorough investigations are necessary and could result in a promising a branching rule. This method and similar methods where the aim is to produce better estimates deserves more attention in order to construct better branching rules.

A more thorough investigation of the time window branching rule might reveal under which conditions it is superior to the other branching rules. Preliminary efforts in this directions were made. In an effort to utilize both the concept of branching on time windows and on arcs, we designed a branching scheme where arc branching was used on flows around 0.5 for the chosen arc. Therefore as the flow on the chosen arc deviates from 0.5 it must be regarded as a less attractive candidate. In the interval $[0.5 - \epsilon_{TW}, 0.5]$ arc-branching is regarded as attractive and branching on time windows is not considered, while if the flow of the chosen arc-branch is in the interval

$[0, 0.5 - \epsilon_{TW}[$ branching on time windows is also considered (see figure 9.3).

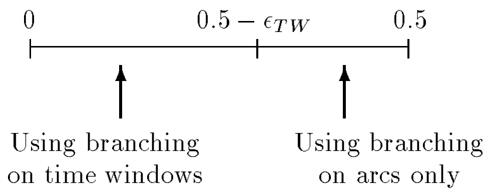


Figure 9.3: Using branching on time windows only when the candidate for branching on arcs does not seem attractive enough.

We did some preliminary tests of this scheme on our test-bed problems with different values of ϵ_{TW} but did not get any conclusive results. Further research in this direction is necessary.

Another interesting idea would be to branch in order to exploit properties of the resulting matrices. In the literature 3 types of matrices are know to have *integral property*, that is, the matrix represents a feasible region in which all extreme points are integer. The 3 types are: total unimodular matrices, balanced matrices and perfect matrices (a thorough description is given in [RF88]). This would require a branching strategy that was guided by how "close" the matrix was to one of the 3 types of matrices with integral property.

9.1.6 Stabilized column generation

Adaptation and implementation of stabilized column generation should be considered for the VRPTW. In a paper [dVDH99] by du Merle et al. a method to stabilize and accelerate convergence of column generation is described.

Two ways of overcoming (or at least reducing) degeneracy are to perturb the problem slightly by adding bounded surplus and slack variables. So instead of

$$(P) \qquad \min_{x\geq 0}\{c_T x : Ax = b\}$$

we try to solve

$$(P_\epsilon) \qquad \min_{(x,y_-,y_+)\geq 0}\{c_T x : Ax - y_- + y_+ = b, y_- \geq \epsilon_-, y_+ \geq \epsilon_+\}.$$

Another method is adding penalties to the objective function. Here, we get

$$\begin{aligned}(P_\delta) \qquad & \min_{x\geq 0} c_T x + \delta\|Ax - b\|_1 = \\ & \min_{(x,y_-,y_+)\geq 0}\{c_T x + \delta y_- + \delta y_+ : Ax - y_- + y_+ = b\}.\end{aligned}$$

By combining the methods we get:

$$\begin{aligned}(P_{\epsilon,\delta}) \qquad \min_{(x,y_-,y_+)\geq 0}\{c_T x + \delta y_- + \delta y_+ \quad : \quad & Ax - y_- + y_+ = b, \\ & y_- \geq \epsilon_-, y_+ \geq \epsilon_+\}.\end{aligned}$$

Here y_- and y_+ are vectors of surplus and slack variables with upper bounds ϵ_- and ϵ_+. In the objective function y_- and y_+ are penalized by δ_- and δ_+, respectively.

Now consider the dual of $(P_{(\epsilon,\delta)})$ (denoted $(D_{(\epsilon,\delta)})$).

$$(D_{(\epsilon,\delta)}) \qquad \begin{array}{ll} \max & b^T - w_-\epsilon_- - w_+\epsilon_+ \\ s.t. & A^T\tilde{\phi} \leq c \\ & -\tilde{\phi} - w_- \leq -\delta_- \\ & \tilde{\phi} - w_+ \leq \delta_+ \\ & w_-, w_+ \geq 0 \end{array}$$

The penalisation of y_- and y_+ in $(P_{(\epsilon,\delta)})$ amounts to penalize the dual variables $\tilde{\phi}$ when they lie outside the interval $[\delta_-, \delta_+]$.

Let x^*, ϕ^* be optimal solutions of (P) and the dual of (P), respectively. Furthermore let $(\tilde{x}^*, y_+^*, y_-^*)$ and $(\tilde{\phi}^*, w_+^*, w_-^*)$ be the optimal solutions of $(P_{(\epsilon,\delta)})$ and $(D_{(\epsilon,\delta)})$, respectively. Then $P \equiv P_{(\epsilon,\delta)}$ if one of two conditions is met:

1. $\epsilon_- = \epsilon_+ = 0$.
2. $\delta_- < \tilde{\phi}^* < \delta_+$.

These provide the stopping criteria for stabilized column generation. Du Merle et al. presents in [dVDH99] strategies for updating $\delta_+, \delta_-, \epsilon_-$ and ϵ_+. So the generic column generation method is extended with a new stopping criteria and an updating scheme for $\delta_+, \delta_-, \epsilon_-$ and ϵ_+.

Sometimes, the column generation phase of the VRPTW algorithm exhibits slow convergence (especially when solving the root node). Using stabilized column generation, du Merle et al. achieve a speedup of 7.41 and a reduction of column generation iterations by a factor 3.33 solving an Airline Crew Pairing problem. It would be interesting to adapt the method to the VRPTW.

9.1.7 Limited subsequence

A technique frequently used in a number of heuristics is to only investigate the candidates, that seen from a local perspective look most promising. This way the number of possible combinations that have to be checked can be drastically reduced.

The huge number of possible routes that sometimes have to be generated in the SPPTWCC subroutine can maybe be reduced by using limited subsequence. Instead of trying to extend a given label to all other labels where time windows and capacity constraints are observed, we try to extend only to the k_l closest customers. By choosing k_l appropriately we can cut the number of extensions significantly but at the same time still be able to deliver routes that are as good as the ones generated by the original SPPTWCC subroutine. This is similar to how it is used by professor David M. Ryan in solving rostering and scheduling problems for airline crews (see [RF88, Rya92]), and the idea for using it on the VRPTW grew out of numerous discussions with professor David M. Ryan. Note that in order to ensure optimality we still have to run the original SPPTWCC subroutine at least once to confirm optimality.

Instance Customers		No lim. subseq.	$k_l = 3$	$k_l = 5$	$k_l = 10$
R101	100	17.88	16.10	15.43	13.71
R202	50	272.92	46.56	63.37	105.39
R203	25	147.55	80.92	116.65	382.85

Table 9.1: Running times of the sequential algorithm for VRPTW not using limited subsequence and 3 different levels of limited subsequence.

We did some preliminary investigations on 3 problems. The results of extending only to the 3, 5, and 10 closest customers are shown in table 9.1.

As can be seen from table 9.1 limited subsequence can lead to significant savings with respect to running time. On the other hand it seems to be more sensitive to how we choose the involved constant (k_l) than forced early stop from section 7.8.2. The idea is definitely promising as we generally obtain faster running times, but on the other hand as the table shows sometimes the running times gets worse (this will be the case if the variant of the SPPTWCC subroutine uses almost as much time as the "exact" version, thereby forcing us to make two time consuming subroutine calls instead of only one).

In a more dynamic setting where the geography, vehicle capacity and time windows are used in setting k_l limited subsequence can be an effective way to reduce computing time.

9.1.8 Speeding up the parallel algorithm

One of the present problems with the parallel algorithm is the dependency on the time used to solve the root node. The root node contains the first calls of the SPPTWCC routine that are often very time consuming relative to the remaining calls of the subroutine. A Branch-and-Bound tree is often called *slim* if only a few Branch-and-Bound nodes are alive on each level of the tree depth. If the Branch-and-Bound tree is very slim, a considerable amount of time may be used before the parallel phase can be started. An idea would be to apply a heuristic instead of SPPTWCC in the first sequential part of the parallel algorithm. Then instead of only branching once we branch enough times to give every processor at least one Branch-and-Bound node to work on. As the parallel phase is started again we use the SPPTWCC subroutine.

9.2 Main conclusions

The investigation of several characteristics of the execution of a column-generation-based VRPTW algorithm have led to new insight in the practical performance of these algorithms. We have successfully implemented a series of new techniques to overcome some of the challenges of the VRPTW. The running time has been reduced with column deletion and forced early stop from the SPPTWCC subroutine. New ways of generating 2-path cuts have made that part of the code more effective. The new code made most impact on the most time-consuming problems.

The techniques have made it possible to solve problems to optimality that have not been solved before. Some of the R2, C2 and RC2 problems of Solomon have been solved, and a number of previously unsolved problems from the R1, C1 and RC1 sets have been solved.

A number of different techniques have been applied in order to speed up the VRPTW algorithm based on column-generation. Our first experiments tested the idea of branching on resource constraints developed by Gélinas et al. ([GDDS95]). At the same time we tested whether lazy evaluation and best first selection is the best setup of the Branch-and-Bound scheme for the VRPTW. To my knowledge this has not previously been tested, and therefore it has previously seemed to be a foregone conclusion to use lazy evaluation and best first selection. It is worth noting that there exists problems and techniques where the lazy evaluation/best first selection is not the best choice (for example in [CP] side-effects make depth-first selection better for both the Job-Shop Problem and the Quadratic Assignment Problem).

The idea of branching on resource constraints developed by Gélinas et al. in [GDDS95] performs quite well. They work on the Vehicle Routing Problem with Backhauling and Time Windows (VRPBTW, see section 5.6).

Obviously there are structural differences between VRPBTW and VRPTW as we were not able to achieve the same positive results for the VRPTW. Therefore further development of advanced branching strategies based on branching on resource constraints were not carried out. It would be interesting to analyze what characteristics of the backhauling scheme made branching on resource windows behave well.

Experiments with the Ryan-Foster branching rule showed that due to the large number of possible extensions of the partial routes in the SPPTWCC this idea does not work for the VRPTW.

The new scheme for detecting S sets generates good results. By using geographical information we are able to generate cuts faster. The implementation of a heuristic for the "feasibility TSPTW" does only have minor effects on the performance. For the R1, C1 and RC1 instances the sets that need to be checked are seldomly so large that the algorithm used by Kohl has problems. For the R2, C2 and RC2 instances a few large sets are generated but not checked due to an upper limit on the set size. If this upper limit was removed the algorithm used by Kohl would have trouble solving some cases. Here the heuristic could be used instead.

The trivial lower bound was tested. It does not seem to have any significant effect. On the other hand it generally does not slow down the algorithm either, and it is fast to perform. Generating cuts in other Branch-and-Bound nodes beside the root node seems generally not to be worthwhile. Only a few Branch-and-Bound nodes can be removed and the cuts generated can not be inserted as they are not globally valid.

The column reduction scheme performs really well. The time saved removing columns not likely to be used again outweighs the administration costs of the scheme. The performance results are very encouraging. In order to obtain even better performance the number of Branch-and-Bound nodes between calls of the reduction subroutine must be made dependent of the breath of the Branch-and-Bound tree. For each "live" branch the "good" routes are worth keeping so ideally we should keep track of which branches are alive. In this way a "good" route belonging to a branch that is no longer alive can be deleted with a good chance of not affecting performance. Instead it is simpler to have an estimate B on the number of "live" branches and then call the reduction subroutine every time B Branch-and-Bound nodes have been processed.

Random selection did not contribute to faster execution time, but forced early stop reduced the running time, sometimes drastically. Further investigations should determine under which condition the best performance is obtained.

The experiments undertaken have added valuable knowledge on structure and properties of solving VRPTW instances. It has identified bottlenecks in computation that needs to be investigated in the future. Furthermore instances not previously solved to optimality has been solved, including a number of R2, C2 and RC2 instances (as reported in section 7.8) but also R103 with 100 customers, R106 with 100 customers and R107 with 100 customers (the solutions can be found in appendix A). In an effort to solve the R109, R110 and RC102 with 100 customers the limit on the number of Branch-and-Bound nodes was reached before the optimal solution was found, and for R108 with 50 customers we reached the limit of 50000 columns before the optimal solution was found.

A number of tests were run on the SP2 using the parallel algorithm. For the RC102 with 50 customers which is one of the most time consuming instances solved in [Koh95] the running time for 5 processors was 882.42 seconds and for 10 processors the running time was cut to 350.19 seconds, that is, we experience a speed-up anomaly. Furthermore the parallel algorithm solved R112 with 50 customers in 4501.50 seconds using 10 processors. This instance has never been solved to optimality before. Again the load-balancing was excellent and the parallel phase was initiated after approximately 500 seconds.

We have developed a parallel algorithm that exhibits good speedup performance. The load-balancing strategy leads to good results as we get the work quite well balanced both on 4 processors and on 8 processors. This parallel algorithm together with the newly designed strategies from chapter 7 could solve several of the Solomon problems not yet solved. Unfortunately access to the SP2 is a sparse resource, and the job-queues for jobs that are estimated to run for more than 1 hour are permanently congested.

Still, the main weakness of the parallel algorithm is the time it takes to generate $k \cdot p$ unexplored subspaces. In particular the time it takes to solve the root node can be a serious problem. Further improvements of the parallel algorithm should focus on using parallelism in initial sequential phase. Here a parallel version of the SPPTWCC subroutine might be fruitful, when the time windows are too wide to allow the sequential SPPTWCC to explore the dynamic programming-tree fast.

Bibliography

[ABMS94] R. Alasdair, A. Bruce, James G. Mills, and A. Gordon Smith. Chimp/mpi user guide. Technical report, Edinburgh Parallel Computing Centre, University of Edinburgh, 1994. Available via WWW at `www.epcc.ed.ac.uk/epcc-tec/documents`.

[ACD+96] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, pages 18 – 28, February 1996.

[AD95] Jürgen Antes and Ulrich Derigs. A new parallel tour construction algorithm for the vehicle routing problem with time windows. Technical report, Lehrstuhl für Wirtschaftsinformatik und Operations Research, Universität zu Köln, March 1995.

[AFG97] Norbert Ascheuer, Matteo Fischetti, and Martin Grötschel. A polyhedral study of the asymmetric travelling salesman problem with time windows. Available via WWW at `www.zib.de`, February 1997. Preprint.

[AMS89] Yogesh Agarwal, Kamlesh Mathur, and Harvey M. Salkin. A set-partitioning-based exact algorithm for the vehicle routing problem. *Networks*, 19:731 – 749, 1989.

[Ant96] Mario Antonioletti. Scalable computing – from workstations to mpps. Available via WWW at `www.epcc.ed.ac.uk/-epcc-tec/documents`, July 1996.

[Bal93] Nagraj Balakrishnan. Simple heuristics for the vehicle routing problem with soft time windows. *Journal of the Operational Research Society*, 44(3):279 – 287, 1993.

[BGAB83] Lawrence Bodin, Bruce Golden, Arjang Assad, and Michael Ball. Routing and scheduling of vehicles and crews - the state of art. *Computers & Operations Research*, 10(2):62 – 212, 1983.

[BGG+95] Philippe Badeau, Michel Gendreau, François Guertin, Jean-Yves Potvin, and Éric Taillard. A parallel tabu search heuristic for the vehicle routing problems with time windows. Technical Report CRT-95-84, Centre de recherche sur les transports, December 1995.

[BJN+94] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-price: Column generation for solving huge integer problems. In J. R. Birge and K. G. Murty, editors, *Mathematical Programming: State of the art 1994*. Braun-Brumfield, 1994.

[BJN+98] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46(3):316 – 329, 1998.

[Bod90] Lawrence D. Bodin. Twenty years of routing and scheduling. *Operations Research*, 38(4):571 – 579, July-August 1990.

[Bre95] Alex Van Breedam. Vehicle routing: Bridging the gap between theory and practice. *Belgian Journal of Operations Research, Statistics and Computer Science*, 35(1):63 – 80, 1995.

[BS86] Edward K. Baker and Joanne R. Schaffer. Solution improvement heuristics for the vehicle routing and scheduling problem with time window constraints. *American Journal of Mathematical and Management Science*, 6(3, 4):261 – 300, 1986.

[Ca93] N. Christofides and J. Paix ao. Algorithms for large scale set covering problems. *Annals of Operations Research*, 43:261 – 277, 1993.

[CCPS98] William J. Cook, William H. Cunningham, William R. Pulleybank, and Alexander Schrijver. *Combinatorial Optimization*. Integer and Combinatorial Optimization. Wiley Interscience, 1998.

[CGR93] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest path algorithms: Theory and experimental evaluation. Draft, July 1993.

[CJR81] Frank H. Cullen, John J. Jarvis, and H. Donald Ratliff. Set partitioning based heuristics for interactive routing. *Networks*, 11(2):125 – 143, 1981.

[CKP+96] David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken. Logp : A practical model of parallel computation. *Communications of the ACM*, 39(11):78 – 85, 1996.

[CL98] Teodor Gabriel Crainic and Gilbert Laporte. *Fleet Management and Logistics*. Kluwer, 1998.

[Cla90] Jens Clausen. Solving difficult combinatorical optimization-problems – can parallel computers help? [In Danish], August 1990.

[Cla96] Jens Clausen. Parallel search-based methods in optimization. draft, 1996.

[Cla97] Jens Clausen. Parallel branch and bound – principles and personal experiences. In Athanasios Migdalas, Panos M. Pardalos, and Sverre Storøy, editors, *Parallel Computing in Optimization*, Applied Optimization. Kluwer Academic Publishers, 1997.

[CMT79] Nicos Christofides, Aristide Mingozzi, and Paolo Toth. The vehicle routing problem. In Nicos Christofides, Aristide Mingozzi, Paolo Toth, and Claudio Sandi, editors, *Combinatorial*

Optimization, chapter 11, pages 315 – 338. John Wiley & Sons, 1979.

[CMT81] Nicos Christofides, Aristide Mingozzi, and Paolo Toth. Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxation. *Mathematical Programming*, 20(3):255 – 282, 1981.

[CP] Jens Clausen and Michael Perregaard. On the best search strategy in parallel branch-and-bound - best-first-search vs. lazy depth-first-search. to appear in *Annals of Operations Research*.

[CR96] Wen-Chyuan Chiang and Robert A. Russell. Simulated annealing metaheuristics for the vehicle routing problem with time windows. *Annals of Operations Research*, 63:3 – 27, 1996.

[CR97] Wen-Chyuan Chiang and Robert A. Russell. A reactive tabu search metaheuristic for the vehicle routing problem with time windows. *INFORMS Journal of Computing*, 9(4):417 – 430, 1997.

[CT96] Alan Chalmers and Jonathan Tidmus. *Practical Parallel Processing*. International Thomson Computer Press, 1996.

[CW64] G. Clarke and W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12:568 – 581, 1964.

[DDS92] Martin Desrochers, Jacques Desrosiers, and Marius Solomon. A new optimization algorithm for the vehicle routing problem with time windows. *Operations Research*, 40(2):342 – 354, March-April 1992.

[DDSS93] Jacques Desrosiers, Yvan Dumas, Marius M. Solomon, and F. Soumis. Time constrained routing and scheduling. Technical report, GERAD, September 1993.

[Des88] Martin Desrochers. An algortihm for the shortest path problem with resource constraints. Technical Report G-88-27, GERAD, École des Hautes Études Commerciales, Université de Montréal, September 1988.

[DFvG83] Edsger W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16:217 – 219, June 1983.

[DLSS88] Martin Desrochers, Jan K. Lenstra, Martin W. P. Savelsbergh, and François Soumis. Vehicle routing with time windows: Optimization and approximation. *Vehicle Routing: Methods and Studies*, pages 65 – 84, 1988. Edited by Golden and Assad.

[dRT88] A. de Bruin, A. H. G. Rinnooy Kan, and H. Trienekens. A simulation tool for the performance of parallel branch and bound algorithms. *Mathematical Programming*, 42:245 – 271, 1988.

[DS88a] Martin Desrochers and François Soumis. A generalized permanent labelling algorithm for the shortest path problem with time windows. *INFOR*, 26(3):191 – 212, 1988.

[DS88b] Martin Desrochers and François Soumis. A reoptimization algorithm for the shortest path problem with time windows. *European Journal of Operational Research*, 35:242 – 254, 1988.

[DS96] Jack J. Dongarra and Horst D. Simon. High performance computing in the u.s. in 1995 – an analysis on the basis of the top 500 list. Technical Report uk-cs-96-318, Department of Computer Science, University of Tennessee, 1996. Available via WWW at `www.cs.utk.edu/~library/TechReports.html`.

[DSD84] Jacques Desrosiers, François Soumis, and Martin Desrochers. Routing with time windows by column generation. *Networks*, 14(4):545 – 565, 1984.

[DSDS85] Jacques Desrosiers, François Soumis, Martin Desrochers, and Michel Sauvé. Routing and scheduling with time windows solved by network relaxation and branch-and-bound on time variables. *Computer Scheduling of Public Transport 2*, pages 451 – 469, 1985.

[Dun90] Ralph Duncan. A survey of parallel computer architectures. *IEEE Computer*, pages 5 – 16, February 1990.

[dVDH99] Olivier du Merle, Daniel Villeneuve, Jacques Desrosiers, and Pierre Hansen. Stabilized column generation. *Discrete Mathematics*, 194:229 – 237, 1999.

[FD96] Graham E. Fagg and Jack J. Dongarra. Pvmpi: An integration of the pvm and mpi systems. Technical report, Department of Computer Science, University of Tennessee, April 1996. Available via WWW at `www.cs.utk.edu/~library/-TechReports.html`.

[Fis94a] Marshall L. Fisher. Optimal solution of vehicle routing problems using minimum k-trees. *Operations Research*, 42(4):626 – 642, July-August 1994.

[Fis94b] Marshall L. Fisher. A polynomial algorithm for the degree-constrained minimum k-tree problem. *Operations Research*, 42(4):775 – 779, July-August 1994.

[Fis97] Marshall Fisher. Vehicle routing. In M. O. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser, editors, *Network Routing*, volume 8 of *Handbooks in Operations Research and Management Science*, chapter 1, pages 1 – 79. North-Holland, 1997.

[FJM94] Marshall L. Fisher, Kurt O. Jörnsten, and Oli B. G. Madsen. Vehicle routing with time windows - two optimization algorithms. Technical Report IMM-REP-1994-28, Department of Mathematical Modelling, Technical University of Denmark, 1994.

[FJM97] Marshall L. Fisher, Kurt O. Jörnsteen, and Oli B. G. Madsen. Vehicle routing with time windows: Two optimization algorithms. *Operations Research*, 45(3):488 – 492, May-June 1997.

[Fos94] Ian Foster. *Designing and Building Parallel Programs.* Addison Wesley, 1994.

[FP93] Christian Foisy and Jean-Yves Potvin. Implementing an insertion heuristic for vehicle routing on parallel hardware. *Computers & Operations Research*, 20(7):737 – 745, 1993.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine – A Users' Guide and Turorial for Networked Parallel Computing*. MIT Press, 1994.

[GC94] Bernard Gendron and Teodor Gabriel Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, 42(6):1042 – 1066, November-December 1994.

[GDDS95] Sylvie Gélinas, Martin Desrochers, Jacques Desrosiers, and Marius M. Solomon. A new branching strategy for time constrained routing problems with application to backhauling. *Annals of Operations Research*, 61:91 – 109, 1995.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[GKP] Ananth Grama, Vipin Kumar, and Panos Pardalos. Parallel processing of discrete optimization problems. Available via WWW at `www.cs.umn.edu/~ananth/work.html`.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI*. MIT Press, 1994.

[Gol84] Bruce L. Golden. Introduction to and recent advances in vehicle routing methods. In M. Florian, editor, *Transportation Planning Models*, pages 383 – 418. Elsevier Science Publishers B.V., 1984.

[GP88] Giorgio Gallo and Stefano Pallottino. Shortest path algorithms. *Annals of Operations Research*, 13:3 – 79, 1988.

[GPR94] Bruno-Laurent Garcia, Jean-Yves Potvin, and Jean-Marc Rousseau. A parallel implementation of the tabu search heuristic for vehicle routing problems with time window constraints. *Computers & Operations Research*, 21(9):1025 – 1033, 1994.

[GTA99] Luca Maria Gambardella, Éric Taillard, and Giovanni Agazzi. Macs-vrptw: A multiple ant colony system for vehicle routing problems with time windows. Technical Report IDSIA-06-99, IDSIA, 1999.

[Hal92] Karsten Halse. *Modeling and Solving Complex Vehicle Routing Problems*. PhD thesis, Department for Mathematical Modeling, Technical University of Denmark, 1992.

[HTd95] Alain Hertz, Eric Taillard, and Dominique de Werra. A tutorial on tabu search. Technical report, EPFL, Département de Mathématique, MA-Ecublens, 1995. Available via WWW on `www.idsia.ch/~eric/`.

[JáJá92] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, 1992.

[JMS86] Kurt O. Jörnsten, Oli B. G. Madsen, and Bo Sørensen. Exact solution of the vehicle routing and scheduling problem with time windows by variable splitting. Technical Report 5, Department of Mathematical Modelling, Technical University of Denmark, 1986.

[KB95] George Kontoravdis and Jonathan F. Bard. A grasp for the vehicle routing problem with time windows. *ORSA Journal on Computing*, 7(1):10 – 23, 1995.

[KDM+99] Niklas Kohl, Jacques Desrosiers, Oli B. G. Madsen, Marius M. Solomon, and François Soumis. 2-path cuts for the vehicle routing problem with time windows. *Transportation Science*, 33(1):101 – 116, 1999.

[KL86] G. A. P. Kindervater and J. K. Lenstra. An introduction to parallelism in combinatorial optimization. *Discrete Applied Mathematics*, 14:135 – 156, 1986.

[KLK89] G. A. P. Kindervater, J. K. Lenstra, and A. H. G. Rinnooy Kan. Perspectives on parallel computing. *Operations Research*, 37(6):985 – 989, November-December 1989.

[KM97] Niklas Kohl and Oli B. G. Madsen. An optimization algorithm for the vehicle routing problem with time windows based on lagrangean relaxation. *Operations Research*, 45(3):395 – 406, May-June 1997.

[Koh95] Niklas Kohl. *Exact methods for Time Constained Routing and Related Scheduling Problems.* PhD thesis, Department of Mathematical Modelling, Technical University of Denmark, 1995.

[Kon97] Georgios Athanassio Kontoravdis. *The Vehicle Routing Problem with Time Windows.* PhD thesis, The University of Texas at Austin, August 1997.

[KPS97] Philip Kilby, Patrick Prosser, and Paul Shaw. Guided local search for the vehicle routing problem. In *MIC97, 2nd International Conference on Metaheuristics.* INRIA & PRiSM, 1997.

[Kri95] Jens Kanstrup Kristensen. Route planning and set partitioning. Master's thesis, Department of Mathematical Modelling, Technical University of Denmark, 1995. [in danish].

[KRT87] A. W. J. Kolen, A. H. G. Rinnooy Kaan, and H. W. J. M. Trienekens. Vehicle routing with time windows. *Operations Research*, 35(2):266 – 273, March-April 1987.

[Kwa93] Renata Krystyna Kwatera. Clustering heuristics for set covering. *Annals of Operations Research*, 43:295 – 308, 1993.

[Lap97] Gilbert Laporte. Recent advances in routing algorithms. Technical Report G-97-38, GERAD and École des Hautes Études Commerciales, May 1997.

[Lau94a] Per S. Laursen. General optimizationheuristics – an introduction. Technical report, Department of Computer Science, University of Copenhagen, 1994. [in danish].

[Lau94b] Per S. Laursen. *Parallel Optimization Algorithms - Efficiency vs. Simplicity.* PhD thesis, Department of Computer Science, University of Copenhagen, 1994.

[LK81] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of vehicle routing and scheduling problems. *Networks*, 11:221 – 227, 1981.

[LO95] Gilbert Laporte and Ibrahim H. Osman. Routing problems: A bibliography. *Annals of Operations Researcg*, 61:227 – 262, 1995.

[LRT95] B. Le Cun, C. Roucairol, and The PNN Team. Bob: a unified platform for implementing branch-and-bound like algorithms. Technical report, Laboratoire PRiSM, Université de Versaille, 1995.

[Mad88] Oli B. G. Madsen. Variable splitting and vehicle routing problems with time windows. Technical Report 1A/1988, Department of Mathematical Modelling, Technical University of Denmark, 1988.

[Mad90] Oli B. G. Madsen. Lagrangean relaxation and vehicle routing. Technical report, Department of Mathematical Modelling, Technical University of Denmark, 1990.

[Mal96] Joël Malard. Mpi: A message-passing interface standard. Technical report, Edinburgh Parallel Computing Center, The University of Edinburgh, 1996. Available via WWW at `www.epcc.ed.ac.uk/`.

[MDF+95] Burkhard Monien, Ralf Dieckmann, Rainer Feldmann, Ralf Klasing, Reinhard Lüling, Knut Menzel, Thomas Römke, and Ulf-Peter Schroeder. Efficient use of parallel & distributed systems: From theory to practice. Available via WWW at `www.uni-paderborn.de/pc2/services/public/`, 1995.

[MJ] Oli B. G. Madsen and Søren Kruse Jacobsen. Notes to mathmatical programming 2 – large systems. Department of Mathematical Modelling, Technical University of Denmark. [in danish].

[MMHB] Neil MacDonald, Elspeth Minty, Tim Harding, and Simon Brown. Writing message-passing parallel programs with mpi

– a two day course. Available via WWW at `http://-www.epcc.ed.ac.uk/`.

[NW88] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. Discrete Mathematics and Optimization. Wiley Interscience, 1988.

[Ols88] Brian Olsen. Route-planning with time restrictions. Technical Report 2/88, Department of Mathematical Modelling, Technical University of Denmark, 1988. [in danish].

[Pac95] Peter S. Pacheco. A user's guide to mpi. Available via WWW at `www.usfca.edu/mpi/`, March 1995.

[Pan96] Cherri M. Pancake. Is parallelism for you? *IEEE Computational Science & Engineering*, 3(2):18 – 37, 1996.

[PB96] Jean-Yves Potvin and Sam Bengio. The vehicle routing problem with time windows – part ii : Genetic search. *INFORMS Journal of Computing*, 8(2):165 – 172, Spring 1996.

[PC98] Michael Perregaard and Jens Clausen. Parallel branch-and-bound methods for the job-shop scheduling problem. *Annals of Operations Research*, 83:137 – 160, 1998.

[PDR96] Jean-Yves Potvin, Danny Dubé, and Christian Robillard. A hybrid approach to vehicle routin using neural networks and genetic algorithms. *Applied Intelligence*, 6:241 – 252, 1996.

[PKGR96] Jean-Yves Potvin, Tanguy Kervahut, Bruno-Laurent Garcia, and Jean-Marc Rousseau. The vehicle routing problem with time windows – part i : Tabu search. *INFORMS Journal of Computing*, 8(2):158 – 164, Spring 1996.

[PL90] Panos Pardalos and Xiaoye Li. Parallel branch-and-bound algorithms for combinatorial optimization. *Supercomputer*, 39(VII-5):23 – 30, September 1990.

[PR93] Jean-Yves Potvin and Jean-Marc Rousseau. A parallel route building algorithm for the vehicle routing and scheduling problem with time windows. *European Journal of Operational Research*, 66:331 – 340, 1993.

[PR95] Jean-Yves Potvin and Jean-Marc Rousseau. An exchange heuristic for routering problems with time windows. *Journal of the Operational Research Society*, 46(12):1433 – 1446, 1995.

[PR99] Jean-Yves Potvin and Christian Robillard. Clustering for vehicle routing with a competitive neural network. *Neurocomputing*, 8:125 – 139, 1999.

[RF81] D. M. Ryan and B. A. Foster. An integer programming approach to scheduling. In A. Wren, editor, *Computer Scheduling of Public Transport*, pages 269 – 280. North-Holland Publishing Company, 1981.

[RF88] D. M. Ryan and J. C. Falkner. On the integer properties of scheduling set partitioning models. *European Journal of Operational Research*, 35:442 – 456, 1988.

[Roc] Yves Rochat. Solomon's vrptw instances. Available on the WWW at `http://dmawww.epfl.ch/~rochat/rochat_data/-solomon.html`.

[Rou96] Catherine Roucairol. Parallel processing for difficult combinatorial optimization problems. *European Journal of Operational Research*, 92:573 – 590, 1996.

[RT95] Yves Rochar and Éric D. Taillard. Probalistic diversification and intensification in local search for vehicle routing. *Journal of Heuristics*, 1:147 – 167, 1995.

[Rus95] Robert A. Russell. Hybrid heuristics for the vehicle routing problem with time windows. *Transportation Science*, 29(2):156 – 166, May 1995.

[Rya92] David M. Ryan. The solution of massive generalized set partitioning problems in aircrew rostering. *Journal of the Operational Research Society*, 43(5):459 – 467, 1992.

[Sal75] Harvey M. Salkin. *Integer Programming*. Addison-Wesley Publishing Company, 1975.

[Sav90] Martin W. P. Savelsbergh. An efficient implementation of local search algorithms for constrained routing. *European Journal of Operational Research*, 47:75 – 85, 1990.

[Sav56] Martin W. P. Savelsbergh. Local search in routing problems with time windows. *Annals of Operations Research*, 4:285 – 305, 1985/6.

[SD88] Marius M. Solomon and Jacques Desrosiers. Time window constrained routing and scheduling problems. *Transportation Science*, 22(1):1 – 13, February 1988.

[SF96] Jürgen Schulze and Torsten Fahle. A parallel algorithm for the vehicle routing problem with time window constraints. Working paper, March 1996.

[SF99] Jürgen Schulze and Torsten Fahle. A parallel algorithm for the vehicle routing problem with time window constraints. *Annals of Operations Research*, 86:585 – 607, 1999.

[Sha97] Paul Shaw. A new local search algorithm providing high quality solutions to vehicle routing problems. Available on the web, July 1997.

[Sol86] Marius M. Solomon. On the worst-case performance of some heuristics for the vehicle routing and scheduling problem with time window constraints. *Networks*, 16:161 – 174, 1986.

[Sol87] Marius M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254 – 265, March-April 1987.

[Tai96] Éric Taillard. A heuristic column generation method for the heterogeneous fleet vrp. Technical report, CRT, 1996.

[TBG+95] Éric Taillard, Phillipe Badeau, Michel Gendreau, François Guertin, and Jean-Yves Potvin. A tabu search heristic for the vehicle routing problem with soft time windows. Technical Report CRT-95-66, CRT, 1995.

[Tel94] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

[Tha] Sam R. Thangiah. Vehicle routing with time windows using genetic algorithms. Available from the authors homepage.

[The97] The GreenTrip Consortium. Efficient logistics via intelligent vehicle routing systems. Presented at the European Conference on Integration in Manufacturing (IiM), Dresden, Germany, September 24-26, 1997, 1997.

[The98] The Danish Ministry of Transport. Trafikredegørelse 1997, January 1998. [in danish].

[TO89] J. M. Troya and M. Ortega. A study of parallel branch-and-bound algorithms with best-bound-first search. *Parallel Computing*, 11:121 – 126, 1989.

[TOS] Sam R. Thangiah, Ibrahim H. Osman, and Tong Sun. Metaheuristics for vehicle routing problems with time windows. Available via WWW at `www.aiai.ed.ac.uk/~timd/-vehicle/`.

[TOS94] Sam R. Thangiah, Ibrahim H. Osman, and Tong Sun. Hybrid genetic algorithm, simulated annealing and tabu search methods for vehicle routing problems with time windows. Technical Report SRU-CpSc-TR-94-27, Computer Science Department, Slippery Rock University, 1994.

[TP96] Stefan Tschöke and Thomas Polzer. Portabel parallel branch-and-bound library. Technical report, Department of Computer Science, University of Paderborn, June 1996.

[vD96] Aad J. van der Steen and Jack J. Dongarra. Overview of recent supercomputers. Technical report, Department of Computer Science, University of Tennessee, 1996. 6. edition. Available via WWW at `www.cs.utk.edu/~library/TechReports.html`.

[vL88] H. R. G. van Landeghem. A bi-criteria heuristic for the vehicle routing problem with time windows. *European Journal of Operational Research*, 36:217 – 226, 1988.

[Wed95] Dag Wedelin. An algorithm for large scale 0-1 integer programming with application to airline crew scheduling. *Annals of Operations Research*, 57:283 – 301, 1995.

Appendix A

The solutions to the R1, C1 and RC1 problems

Usually the LP and IP values are reported for the solutions to the Solomon test-sets in the exact papers (in the papers on heuristics it is often worse as the numbers reported are accumulated for each test-set) and nothing more. This appendix contains the LP and IP values of all R1, C1 and RC1 problems that have been solved together with the routes of the optimal solution.

R101	25	**IP_opt**	617.0
		LP_opt	617.0
		Routes	
		5 16 6	
		7 8 17	
		11 19 10	
		18	
		23 22 4 25	
		14 15 13	
		12 9 20 1	
R101	50	**IP_opt**	1044.0
		LP_opt	1043.367
		Routes	
		2 21 40 50 1	
		33 29 9 34 35	
		42 15 41 26	
		36 47 7 10	
		14 44 38 43 13	
		39 23 22 4	
		31 30 20 32	
		5 16 37	
		11 19 49 48	
		45 8 46 17	
		28 12 3 24 25	
		27 18 6	
R101	100	**IP_opt**	1637.7
		LP_opt	1631.15
		Routes	
		39 23 67 55 25	
		31 88 7	
		63 64 49 48	
		62 11 90 10	
		40 53 26	
		52 6	
		36 47 19 8 46 17	
		27 69 30 51 20 32 70	
		92 42 15 87 57 97	
		65 71 9 66 1	
		14 44 38 43 13	
		33 81 50 68	
		2 21 73 41 56 4	
		45 82 18 84 60 89	
		72 75 22 74 58	
		12 76 79 3 54 24 80	
		28 29 78 34 35 77	
		59 98 16 86 91 100	
		5 83 61 85 37 93	
		95 99 94 96	

R102	25	**IP_opt**	547.1
		LP_opt	546.333
		Routes	
		14 16 6 13	
		3 9 20 1	
		18	
		8 17 5	
		2 15 22 4 25	
		7 11 19 10	
R102	50	**IP_opt**	909.0
		LP_opt	909.0
		Routes	
		45 16 6	
		40	
		18 10 31	
		37 42 15 41 2	
		14 44 38 43 13	
		28 29 24 12	
		27 1 30 20 32	
		50 33 9 35 34 3	
		36 47 8 46 17 5	
		11 19 49 48 7	
		21 39 23 22 4 25 26	
R102	100	**IP_opt**	1466.6
		LP_opt	1466.6
		Routes	
		40 53	
		50 33 29 78 34 35 77	
		18	
		28 76 79 54 24 80 12	
		39 23 67 55 25 26	
		73 22 74 72 21	
		27 69 88 10 31	
		36 47 19 8 46 17 93 59	
		65 71 81 3 68	
		87 57 2 58	
		94 96 99 6	
		42 15 41 75 56 4	
		95 14 44 38 43 100 37	
		30 51 9 66 1	
		83 45 61 84 5 60 89	
		62 11 90 20 32 70	
		92 98 85 16 86 91 97 13	
		63 64 49 48 82 7 52	

R103	25	**IP_opt**	454.6
		LP_opt	454.6
		Routes	
		6 13	
		7 19 11 8 18	
		21 23 22 4 25	
		2 15 14 16 17 5	
		12 24 3 9 10 20 1	
R103	50	**IP_opt**	772.9
		LP_opt	765.95
		Routes	
		27 6	
		40	
		36 11 19 49 47 48 7	
		42 43 15 41 2 13	
		45 46 8 18	
		50 33 20 30 32 10 31 1	
		21 39 23 22 4 25 26	
		37 14 44 38 16 17 5	
		28 12 24 29 9 35 34 3	
R103	100	**IP_opt**	1208.7
		LP_opt	1206.38
		Routes	
		40 53	
		51 65 71 9 66 20 32 70	
		94 95 97 87 13	
		96 99 6	
		21 39 23 67 55 25 54	
		52 7 62 11 63 90 10 31	
		2 22 74 72 73 58	
		36 64 49 19 47 48 82 18	
		42 43 15 57 41 75 56 4 26	
		50 33 3 76 79 29 24 68 80 12	
		83 45 84 61 85 93 59	
		27 69 88 8 46 17 5 60 89	
		92 98 14 44 38 86 16 91 100 37	
		1 30 78 34 35 81 77 28	

R104	25	**IP_opt**	416.9
		LP_opt	416.9
		Routes	
		7 19 11 8 18 6 13	
		2 15 14 16 17 5	
		12 24 3 9 20 10 1	
		21 22 23 4 25	
R104	50	**IP_opt**	625.4
		LP_opt	616.5
		Routes	
		21 22 41 23 39 4 25 26	
		40 2 15 43 42 13	
		6 16 44 38 14 37	
		7 48 19 11 10 32 20 30 31 27	
		28 1 50 3 33 9 35 34 29 24 12	
		18 5 17 45 8 46 36 49 47	

R105	25	**IP_opt**	530.5
		LP_opt	530.5
		Routes	
		7 18 8 17	
		2 15 13	
		21 23 22 4	
		19 11 10 20 1	
		12 9 3 24 25	
		5 14 16 6	
R105	50	**IP_opt**	899.3
		LP_opt	892.12
		Routes	
		28 12 29 3 50 1	
		33 30 9 34 35 24	
		21 40 26	
		39 23 41 22 4 25	
		47 36 11 10 20 32	
		42 14 44 16 6	
		2 15 38 43 37 13	
		27 31 7 19 49 48	
		5 45 18 8 46 17	
R105	100	**IP_opt**	1355.3
		LP_opt	1346.142
		Routes	
		28 12 29 79 78 34 35 77	
		63 64 11 90 10	
		62 88 7 18	
		31 30 51 9 81 3 68 24 80	
		72 39 23 67 55 54 4 25	
		27 69 76 50 1	
		52 82 8 84 17 60 89	
		21 73 75 22 41 56 74 58	
		53 40 26	
		33 65 71 66 20 32 70	
		47 36 19 49 46 48	
		2 15 57 87 97 13	
		42 14 44 38 86 43 100 91 93	
		59 95 92 98 16 61 85 37 96	
		5 45 83 99 94 6	

R106	25	**IP_opt**	465.4
		LP_opt	457.3
		Routes	
		18 8 17 5	
		2 15 23 22 4 25 21	
		7 19 11 10	
		14 16 6 13	
		1 9 20 3 24 12	
		21 23 24 12	
R106	50	**IP_opt**	793.0
		LP_opt	791.367
		Routes	
		50 33 29 24 12	
		45 8 18	
		48 47 36 49 46 17 5	
		2 15 40 6	
		27 28 1 30 9 35 34 3	
		21 39 23 41 22 4 25 26	
		7 19 11 10 20 32 31	
		42 14 44 16 38 43 37 13	
R106	100	**IP_opt**	1234.6
		LP_opt	1226.44
		Routes	
		50 33 65 71 66 20 32 70 1	
		28 76 40 53	
		63 64 11 90 10 31	
		69 30 51 81 9 35 34 3 77	
		96 85 91 16 61 99 6	
		73 41 22 75 56 74 2 58	
		48 47 36 19 49 46 82 7 52	
		94 92 42 15 57 87 97 95 13	
		83 45 8 84 17 5 60	
		27 62 88 18 89	
		59 37 14 44 38 86 43 100 98 93	
		12 29 78 79 68 54 24 80	
		21 72 39 23 67 55 4 25 26	

R107	25	**IP_opt**	424.3
		LP_opt	422.925
		Routes	
		2 15 14 6 13	
		21 23 22 4 25 24	
		12 3 9 20 10 1	
		18 7 11 19 8 17 16 5	
R107	50	**IP_opt**	711.1
		LP_opt	704.438
		Routes	
		26 21 39 23 41 22 4 25 24	
		18 45 8 6 13	
		42 43 15 2 40	
		46 36 11 19 49 47 48 7	
		37 14 44 38 16 17 5	
		50 3 33 9 35 34 29 12	
R107	100	**IP_opt**	1064.6
		LP_opt	1051.844
		Routes	
		40 53	
		33 81 65 71 9 35 34 3 77	
		21 72 39 23 67 55 25 54 26	
		2 57 15 41 22 75 56 4 74 73 58	
		42 43 14 44 38 86 16 91 100 37 98	
		28 76 79 78 29 24 68 80 12	
		94 96 92 59 99 6 87 97 95 13	
		52 7 62 11 63 90 32 66 20 51 50	
		60 83 45 46 8 84 5 17 61 85 93	
		48 47 36 64 49 19 82 18 89	
		27 69 30 88 31 10 70 1	
R108	25	**IP_opt**	397.3
		LP_opt	396.139
		Routes	
		2 15 14 16 17 5 6 13	
		21 22 23 4 25 24	
		1 20 9 3 12	
		7 10 11 19 8 18	

R109	25	**IP_opt**	465.4
		LP_opt	457.3
		Routes	
		5 8 18 6	
		7 19 11 10	
		12 3 9 20 1	
		21 22 23 4 25 24	
		2 15 14 16 17 13	
R109	50	**IP_opt**	786.8
		LP_opt	775.096
		Routes	
		27 30 33 9 35 34 24	
		5 45 8 18 6	
		28 12 29 3 50	
		21 23 39 25 4	
		2 15 41 22 40 26	
		7 11 10 32 20 1	
		42 16 44 38 14 43 37 13	
		31 19 47 49 36 46 48 17	
R110	25	**IP_opt**	465.4
		LP_opt	457.3
		Routes	
		2 15 14 16 17 8	
		12 3 9	
		7 19 11 10 20 1	
		21 22 23 4 25 24	
		18 5 6 13	
R110	50	**IP_opt**	697.0
		LP_opt	692.577
		Routes	
		31 11 19 47 49 36 46 48	
		5 16 44 38 14 43 15 42 13	
		6 18 8 45 17 37	
		28 12 29 24 26	
		33 9 35 34 3 50	
		27 7 10 30 20 32 1	
		2 40 21 22 41 23 39 25 4	

R111	25	**IP_opt**	465.4
		LP_opt	457.3
		Routes	
		12 3 9 20 10 1	
		21 23 22 4 25 24	
		2 15 14 16 6 13	
		7 11 19 8 18 17 5	
R111	50	**IP_opt**	707.2
		LP_opt	691.812
		Routes	
		42 15 23 39 4 25 24	
		37 16 44 38 14 43 13	
		27 1 30 20 32 10 31	
		40 2 41 22 21 26	
		28 12 29 3 33 9 35 34 50	
		7 45 8 18 6	
		48 19 11 49 36 47 46 17 5	
R112	25	**IP_opt**	465.4
		LP_opt	457.3
		Routes	
		21 22 23 4 25 24	
		18 8 7 19 11 10	
		2 15 14 16 17 5 6 13	
		12 3 9 20 1	
R112	50	**IP_opt**	630.2
		LP_opt	607.219
		Routes	
		6 5 17 16 44 38 14 37	
		18 8 45 46 36 49 47 48	
		12 29 24 34 35 9 33 3 50	
		28 21 22 41 23 39 25 4 26	
		27 31 7 19 11 10 30 32 20 1	
		40 2 15 43 42 13	

C101	25	**IP_opt**	191.3
		LP_opt	191.3
		Routes	
		5 3 7 8 10 11 9 4 6 2 1	
		13 17 18 19 15 16 14 12	
		20 24 25 23 22 21	
C101	50	**IP_opt**	362.4
		LP_opt	362.4
		Routes	
		20 24 25 27 29 30 28 26 23 22 21	
		5 3 7 8 10 11 9 6 4 2 1	
		43 42 41 40 44 46 45 48 50 49 47	
		32 33 31 35 37 38 39 36 34	
C101	100	**IP_opt**	827.3
		LP_opt	827.3
		Routes	
		43 42 41 40 44 46 45 48 51 50 52 49 47	
		5 3 7 8 10 11 9 6 4 2 1 75	
		20 24 25 27 29 30 28 26 23 22 21	
		67 65 63 62 74 72 61 64 68 66 69	
		90 87 86 83 82 84 85 88 89 91	
		81 78 76 71 70 73 77 79 80	
		98 96 95 94 92 93 97 100 99	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12	
		57 55 54 53 56 58 60 59	

C102	25	**IP_opt**	190.3
		LP_opt	190.3
		Routes	
		5 3 7 8 10 11 9 6 4 2 1	
		13 17 18 19 15 16 14 12	
		20 24 25 23 22 21	
C102	50	**IP_opt**	361.4
		LP_opt	361.4
		Routes	
		20 24 25 27 29 30 28 26 23 22 21	
		32 33 31 35 37 38 39 36 34	
		43 42 41 40 44 46 45 48 50 49 47	
		13 17 18 19 15 16 14 12	
		7 8 10 11 9 6 4 2 1 3 5	
C102	100	**IP_opt**	827.3
		LP_opt	827.3
		Routes	
		43 42 41 40 44 46 45 48 51 50 52 49 47	
		5 3 7 8 10 11 9 6 4 2 1 75	
		20 24 25 27 29 30 28 26 23 22 21	
		67 65 63 62 74 72 61 64 68 66 69	
		90 87 86 83 82 84 85 88 89 91	
		81 78 76 71 70 73 77 79 80	
		98 96 95 94 92 93 97 100 99	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12	
		57 55 54 53 56 58 60 59	

C103	25	**IP_opt**	190.3
		LP_opt	190.3
		Routes	
		20 24 25 23 22 21	
		13 17 18 19 15 16 14 12	
		7 8 10 11 9 6 4 2 1 3 5	
C103	50	**IP_opt**	361.4
		LP_opt	361.4
		Routes	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12	
		7 8 10 11 9 6 4 2 1 3 5	
		20 24 25 27 29 30 28 26 23 22 21	
		43 42 41 40 44 46 45 48 50 49 47	
C103	100	**IP_opt**	826.3
		LP_opt	826.3
		Routes	
		43 42 41 40 44 46 45 48 51 50 52 49 47	
		5 3 7 8 10 11 9 6 4 2 1 75	
		20 24 25 27 29 30 28 26 23 22 21	
		67 65 62 74 72 61 64 68 66 69	
		90 87 86 83 82 84 85 88 89 91	
		81 78 76 71 70 73 77 79 80 63	
		98 96 95 94 92 93 97 100 99	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12	
		57 55 54 53 56 58 60 59	

C104	25	**IP_opt**	186.9
		LP_opt	186.9
		Routes	
		20 24 25 23 22 21	
		7 8 11 9 6 4 2 1 3 5	
		13 17 18 19 15 16 14 12 10	
C104	50	**IP_opt**	358.0
		LP_opt	357.25
		Routes	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12	
		5 3 7 8 10 11 9 6 4 2 1	
		20 24 25 27 29 30 28 26 23 22 21	
		43 42 41 40 44 46 45 48 50 49 47	
C104	100	**IP_opt**	822.9
		LP_opt	822.9
		Routes	
		43 42 41 40 44 45 46 48 51 50 52 49 47	
		5 3 7 8 11 9 6 4 2 1 75	
		20 24 25 27 29 30 28 26 23 22 21	
		67 65 62 74 72 61 64 68 66 69	
		90 87 86 83 82 84 85 88 89 91	
		81 78 76 71 70 73 77 79 80 63	
		98 96 95 94 92 93 97 100 99	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12 10	
		57 55 54 53 56 58 60 59	

C105	25	**IP_opt**	191.3
		LP_opt	191.3
		Routes	
		20 24 25 23 22 21	
		5 3 7 8 10 11 9 6 4 2 1	
		13 17 18 19 15 16 14 12	
C105	50	**IP_opt**	362.4
		LP_opt	362.4
		Routes	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12	
		5 3 7 8 10 11 9 6 4 2 1	
		20 24 25 27 29 30 28 26 23 22 21	
		43 42 41 40 44 46 45 48 50 49 47	
C105	100	**IP_opt**	827.3
		LP_opt	827.3
		Routes	
		43 42 41 40 44 46 45 48 51 50 52 49 47	
		5 3 7 8 10 11 9 6 4 2 1 75	
		20 24 25 27 29 30 28 26 23 22 21	
		67 65 63 62 74 72 61 64 68 66 69	
		90 87 86 83 82 84 85 88 89 91	
		81 78 76 71 70 73 77 79 80	
		98 96 95 94 92 93 97 100 99	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12 10	
		57 55 54 53 56 58 60 59	

C106	25	**IP_opt**	191.3
		LP_opt	191.3
		Routes	
		20 24 25 23 22 21	
		5 3 7 8 10 11 9 6 4 2 1	
		13 17 18 19 15 16 14 12	
C106	50	**IP_opt**	362.4
		LP_opt	362.4
		Routes	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12	
		5 3 7 8 10 11 9 6 4 2 1	
		20 24 25 27 29 30 28 26 23 22 21	
		43 42 41 40 44 46 45 48 50 49 47	
C106	100	**IP_opt**	827.3
		LP_opt	827.3
		Routes	
		43 42 41 40 44 46 45 48 51 50 52 49 47	
		5 3 7 8 10 11 9 6 4 2 1 75	
		20 24 25 27 29 30 28 26 23 22 21	
		67 65 63 62 74 72 61 64 68 66 69	
		90 87 86 83 82 84 85 88 89 91	
		81 78 76 71 70 73 77 79 80	
		98 96 95 94 92 93 97 100 99	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12 10	
		57 55 54 53 56 58 60 59	

C107	25	**IP_opt**	191.3
		LP_opt	191.3
		Routes	
		20 24 25 23 22 21	
		5 3 7 8 10 11 9 6 4 2 1	
		13 17 18 19 15 16 14 12	
C107	50	**IP_opt**	362.4
		LP_opt	362.4
		Routes	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12	
		5 3 7 8 10 11 9 6 4 2 1	
		20 24 25 27 29 30 28 26 23 22 21	
		43 42 41 40 44 46 45 48 50 49 47	
C107	100	**IP_opt**	827.3
		LP_opt	827.3
		Routes	
		43 42 41 40 44 46 45 48 51 50 52 49 47	
		5 3 7 8 10 11 9 6 4 2 1 75	
		20 24 25 27 29 30 28 26 23 22 21	
		67 65 63 62 74 72 61 64 68 66 69	
		90 87 86 83 82 84 85 88 89 91	
		81 78 76 71 70 73 77 79 80	
		98 96 95 94 92 93 97 100 99	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12 10	
		57 55 54 53 56 58 60 59	

C108	25	**IP_opt**	191.3
		LP_opt	191.3
		Routes	
		20 24 25 23 22 21	
		5 3 7 8 10 11 9 6 4 2 1	
		13 17 18 19 15 16 14 12	
C108	50	**IP_opt**	362.4
		LP_opt	362.4
		Routes	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12	
		5 3 7 8 10 11 9 6 4 2 1	
		20 24 25 27 29 30 28 26 23 22 21	
		43 42 41 40 44 46 45 48 50 49 47	
C108	100	**IP_opt**	827.3
		LP_opt	827.3
		Routes	
		43 42 41 40 44 46 45 48 51 50 52 49 47	
		5 3 7 8 10 11 9 6 4 2 1 75	
		20 24 25 27 29 30 28 26 23 22 21	
		67 65 63 62 74 72 61 64 68 66 69	
		90 87 86 83 82 84 85 88 89 91	
		81 78 76 71 70 73 77 79 80	
		98 96 95 94 92 93 97 100 99	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12 10	
		57 55 54 53 56 58 60 59	

C109	25	**IP_opt**	191.3
		LP_opt	189.333
		Routes	
		20 24 25 23 22 21	
		7 8 10 11 9 6 4 2 1 5 3	
		13 17 18 19 15 16 14 12	
C109	50	**IP_opt**	362.4
		LP_opt	361.61
		Routes	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12	
		5 3 7 8 10 11 9 6 4 2 1	
		20 24 25 27 29 30 28 26 23 22 21	
		43 42 41 40 44 46 45 48 50 49 47	
C109	100	**IP_opt**	827.3
		LP_opt	822.861
		Routes	
		43 42 41 40 44 46 45 48 51 50 52 49 47	
		5 3 7 8 10 11 9 6 4 2 1 75	
		20 24 25 27 29 30 28 26 23 22 21	
		67 65 63 62 74 72 61 64 68 66 69	
		90 87 86 83 82 84 85 88 89 91	
		81 78 76 71 70 73 77 79 80	
		98 96 95 94 92 93 97 100 99	
		32 33 31 35 37 38 39 36 34	
		13 17 18 19 15 16 14 12 10	
		57 55 54 53 56 58 60 59	

RC101	25	**IP_opt**	461.1
		LP_opt	406.625
		Routes	
		2 5 7 6 8 3 1 4	
		14 12 15 16 9 10 13 17	
		23 21 19 18 25 24	
		11 22 20	
RC101	50	**IP_opt**	944.0
		LP_opt	850.021
		Routes	
		14 47 12 15 16 9 10 13 17	
		5 45 2 7 6 8 46 4	
		27 29 31 34 50	
		23 21 19 18 48 25	
		33 30 28 26 32	
		39 36 38 41 40 43 37 35	
		11 22 49 20 24	
		42 44 3 1	
RC101	100	**IP_opt**	1619.8
		LP_opt	1584.094
		Routes	
		82 11 15 16 9 10 13 17	
		90	
		64 51 85 84 56 66	
		69 98 88 53 78 60	
		63 33 28 30 34 50 91 80	
		95 62 67 71 94 96 54	
		23 21 18 49 22 20 25 24	
		5 45 2 7 6 8 3 1 70	
		92 31 29 27 26 32 93	
		59 75 87 97 58 77	
		65 52 99 57 86 74	
		72 36 38 41 40 43 37 35	
		83 19 76 89 48	
		39 42 44 61 81 68 55	
		47 14 12 73 79 46 4 100	

RC102	25	**IP_opt**	351.8
		LP_opt	351.8
		Routes	
		21 23 19 18 22 20 25 24	
		12 14 11 15 16 9 10 13 17	
		7 6 8 5 3 1 4 2	
RC102	50	**IP_opt**	822.5
		LP_opt	719.902
		Routes	
		14 47 11 15 16 9 10 13 17 12	
		42 44 43 35 37	
		39 36 40 38 41	
		34 31 29 27 26 32 50	
		23 19 18 22 49 48 21 25	
		1 3 45 5 8 7 6 46 4 2	
		33 28 30 20 24	
RC103	25	**IP_opt**	332.8
		LP_opt	332.05
		Routes	
		20 19 18 21 23 22 25 24	
		7 6 8 5 3 1 4 2	
		12 15 11 9 10 13 16 17 14	
RC103	50	**IP_opt**	710.9
		LP_opt	643.133
		Routes	
		12 14 15 11 9 10 13 16 17 47	
		33 27 30 32 28 26 29 31 34	
		20 18 48 21 23 22 49 19 25 24	
		42 43 44 40 38 41 50	
		39 36 35 37	
		2 45 46 8 7 6 4 5 3 1	
RC104	25	**IP_opt**	306.6
		LP_opt	305.825
		Routes	
		20 19 18 21 23 25 24 22	
		2 6 7 8 4 5 3 1	
		10 11 15 16 9 13 17 14 12	
RC104	50	**IP_opt**	545.8
		LP_opt	541.8
		Routes	
		12 14 15 11 10 9 13 16 17 47	
		2 6 7 8 46 4 45 5 3 1	
		42 44 43 38 37 35 36 40 39 41	
		20 49 19 23 48 18 21 25 24 22	
		34 31 29 27 26 28 30 32 33 50	

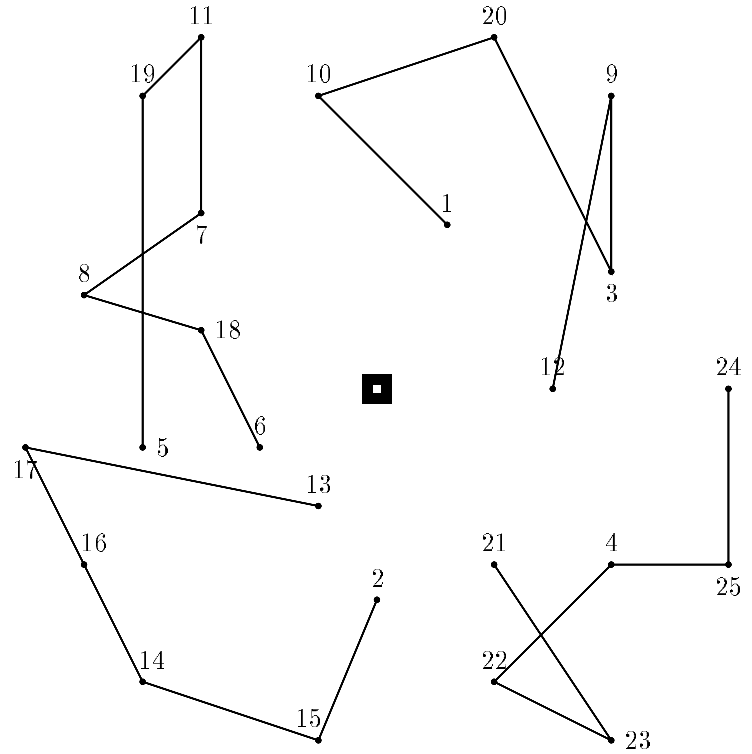
RC105	25	**IP_opt**	411.3
		LP_opt	410.95
		Routes	
		11 9 10	
		19 23 18 22 20 21 25 24	
		12 14 15 16 13 17	
		2 5 3 1 8 6 7 4	
RC105	50	**IP_opt**	855.3
		LP_opt	754.443
		Routes	
		2 45 5 8 6 7 46 4 3 1	
		42 44 40 35 43	
		39 36 37 38 41	
		33 22 49 20	
		11 9 10	
		12 14 47 15 16 13 17	
		31 29 27 30 28 26 32 34 50	
		19 23 21 48 18 25 24	
RC105	100	**IP_opt**	1513.7
		LP_opt	1495.28
		Routes	
		64 86 87 59 97 75 58	
		81 61 68	
		51 76 89 48 21 25 24	
		92 95 62 67 71 93 96	
		2 45 5 3 1 8 6 55	
		12 14 47 15 16 9 10 13 17	
		83 19 23 18 22 49 20 77	
		39 36 37 38 41 72 54 94 80	
		42 44 40 35 43	
		31 29 27 30 28 26 32 34 50 91	
		69 88 79 7 46 4 70 100	
		33 63 85 84 56 66	
		90 53 98	
		65 99 52 57 74	
		82 11 73 78 60	

RC106	25	**IP_opt**	345.5
		LP_opt	339.242
		Routes	
		11 15 16 14 12 10 9 13 17	
		23 21 18 19 20 22 25 24	
		2 5 8 7 6 4 3 1	
RC106	50	**IP_opt**	732.2
		LP_opt	664.433
		Routes	
		11 12 14 47 15 16 9 10 13 17	
		42 44 39 40 36 38 41 43 37 35	
		31 29 27 26 28 34	
		2 45 5 8 7 6 46 4 3 1	
		33 30 32 50	
		23 21 18 19 49 20 22 48 25 24	
RC107	25	**IP_opt**	298.3
		LP_opt	293.550
		Routes	
		12 14 17 16 15 13 9 11 10	
		25 23 21 18 19 20 22 24	
		2 6 7 8 5 3 1 4	
RC107	50	**IP_opt**	642.7
		LP_opt	591.476
		Routes	
		2 6 7 8 5 3 1 45 46 4	
		50	
		11 12 14 47 17 16 15 13 9 10	
		23 25 21 49 19 18 48 22 20 24	
		31 29 27 28 26 34 32 30 33	
		41 38 39 42 44 43 40 37 35 36	
RC108	25	**IP_opt**	294.5
		LP_opt	280.385
		Routes	
		12 14 17 16 15 13 9 11 10	
		22 20 19 18 21 23 25 24	
		2 6 7 8 4 5 3 1	
RC108	50	**IP_opt**	598.1
		LP_opt	538.957
		Routes	
		12 14 47 17 16 15 13 9 11 10	
		25 23 21 48 18 19 49 20 22 24	
		2 6 7 8 46 4 45 5 3 1	
		33 32 30 28 26 27 29 31 34	
		50	
		41 42 44 43 40 38 37 35 36 39	

Appendix B

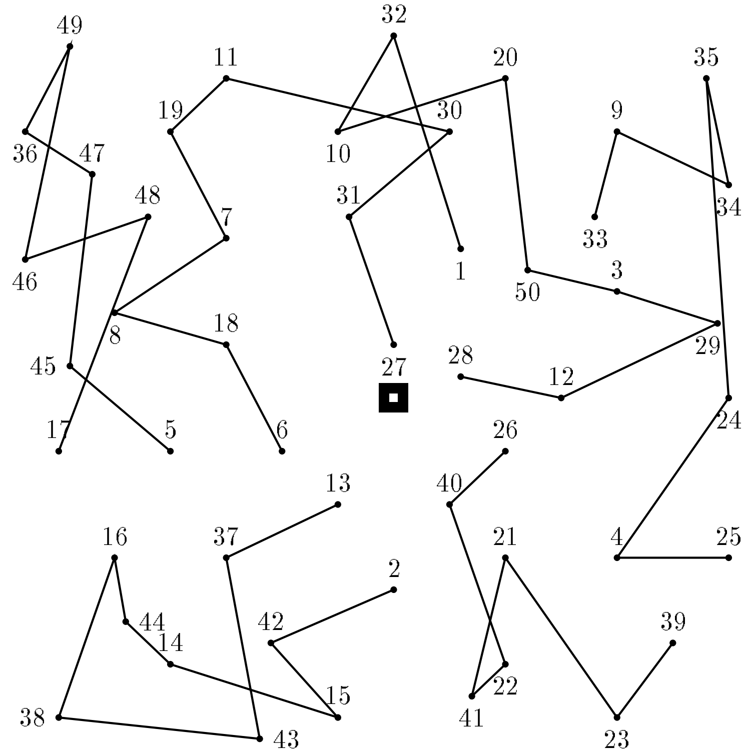
The solutions to the R2, C2 and RC2 problems

This appendix lists the solutions to the problems in the Solomon test sets R2, C2 and RC2, that the algorithm was able to find.



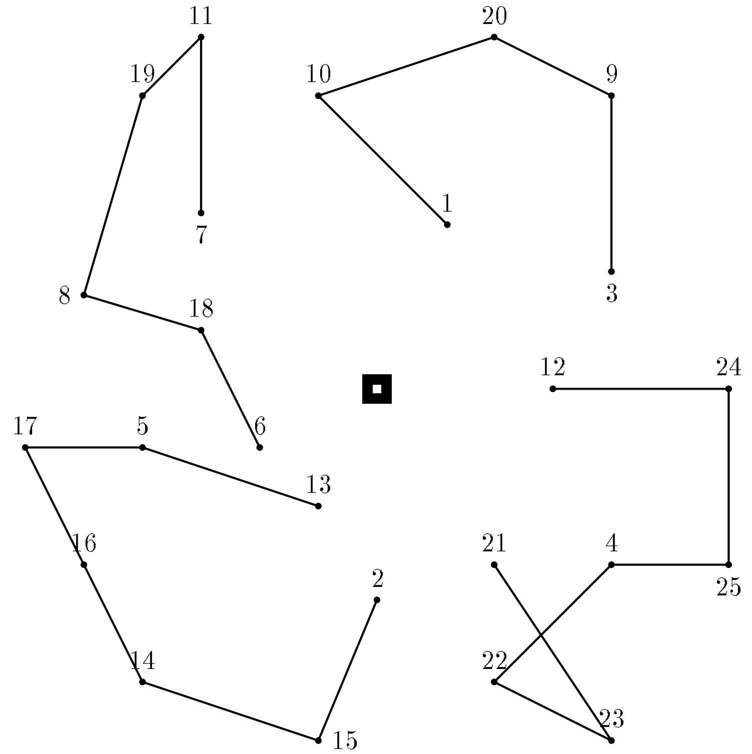
IP_opt	463.3
LP_opt	460.1
Routes	Length
12 9 3 20 10 1	124.2
2 15 14 16 17 13	105.5
5 19 11 7 8 18 6	117.4
21 23 22 4 25 24	116.2

Figure B.1: Optimal solution of R201 with 25 customers.



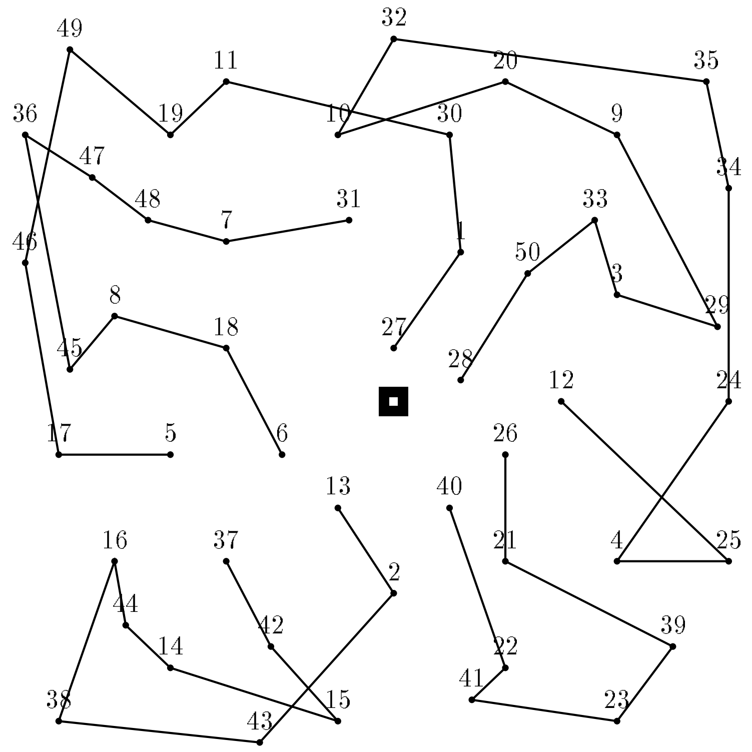
IP_opt	791.9
LP_opt	788.425
Routes	Length
27 31 30 11 19 7 8 18 6	113.1
33 9 34 35 24 4 25	145.6
5 45 47 36 49 46 48 17	152.6
28 12 29 3 50 20 10 32 1	128.8
39 23 21 41 22 40 26	111.9
2 42 15 14 44 16 38 43 37 13	139.9

Figure B.2: Optimal solution of R201 with 50 customers.



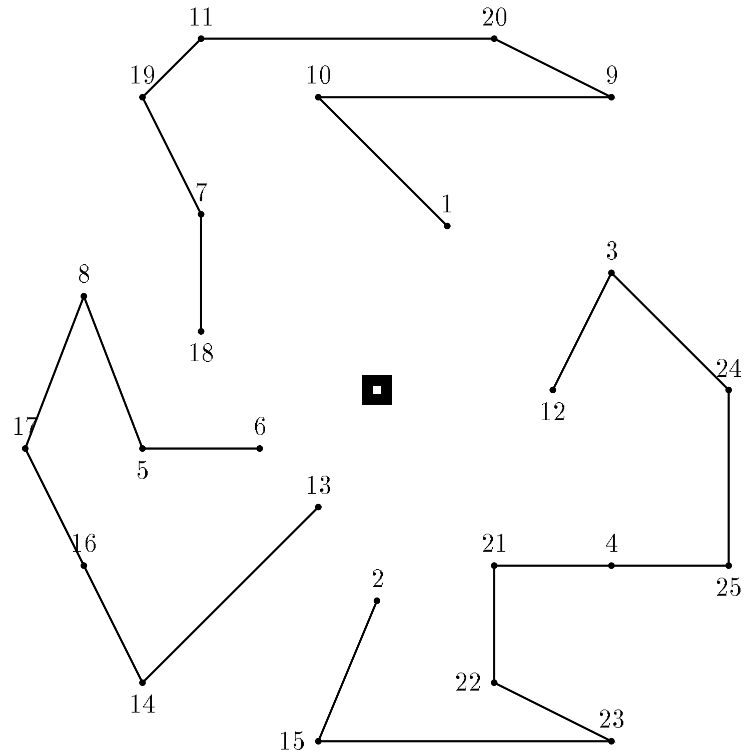
IP_opt	410.5
LP_opt	406.35
Routes	Length
3 9 20 10 1	949
21 23 22 4 25 24 12	116.2
2 15 14 16 17 5 13	105.9
7 11 19 8 18 6	93.5

Figure B.3: Optimal solution of R202 with 25 customers.



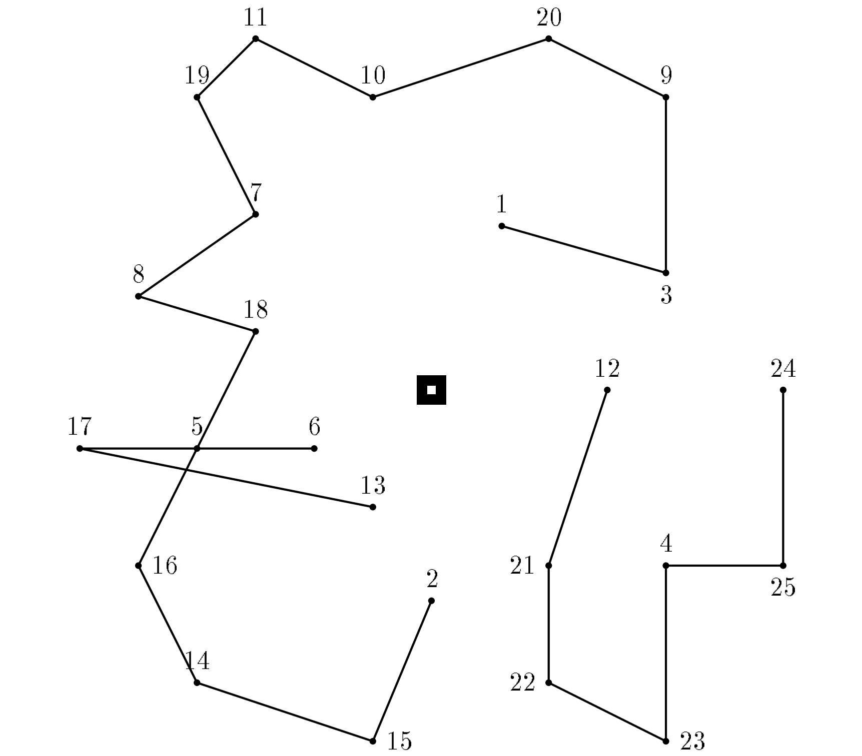
IP_opt	698.5
LP_opt	692.738
Routes	Length
26 21 39 23 41 22 40	90.9
31 7 48 47 36 45 8 18 6	110.4
27 1 30 11 19 49 46 17 5	135.5
28 50 33 3 29 9 20 10 32 35 34 24 4 25 12	222.0
37 42 15 14 44 16 38 43 2 13	139.5

Figure B.4: Optimal solution of R202 with 50 customers.



IP_opt	391.4
LP_opt	379.882
Routes	Length
6 5 8 17 16 14 13	104.1
2 15 23 22 21 4 25 24 3 12	153.3
18 7 19 11 20 9 10 1	136.5

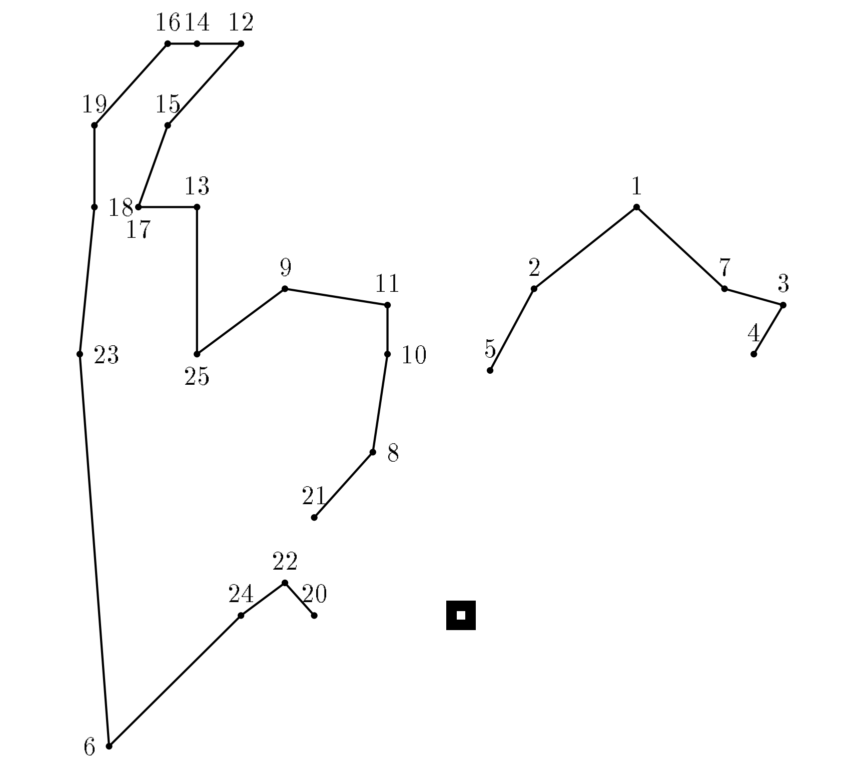
Figure B.5: Optimal solution of R203 with 25 customers.



IP_opt	393.0
LP_opt	381.283

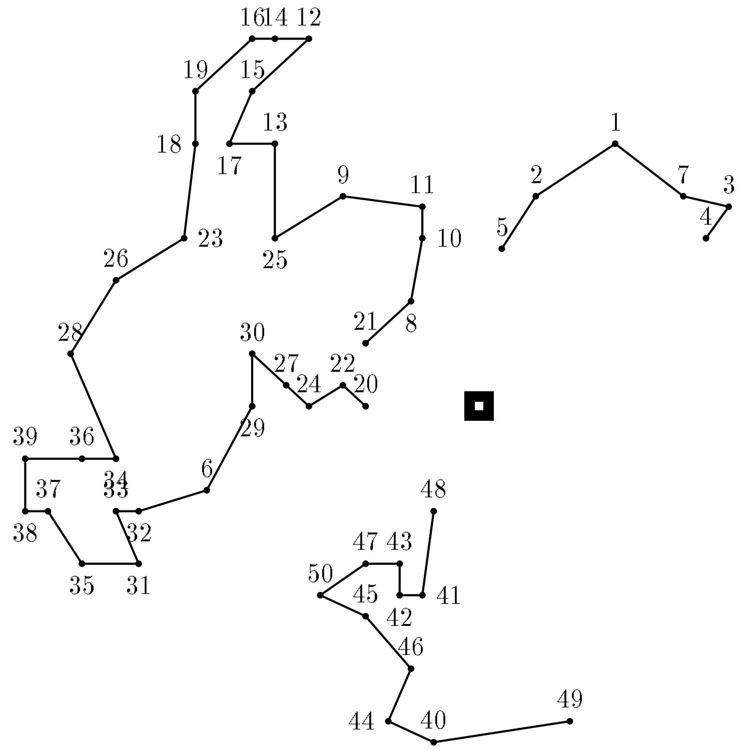
Routes	Length
6 17 13	67.9
12 21 22 23 4 25 24	122.6
2 15 14 16 5 18 8 7 19 11 10 2 0 9 3 1	205.0

Figure B.6: Optimal solution of R205 with 25 customers.



IP_opt	214.7
LP_opt	214.7
Routes	Length
5 2 1 7 3 4	71.2
20 22 24 6 23 18 19 16 14 12 15 17 13 25 9 11 10 8 21	146.0

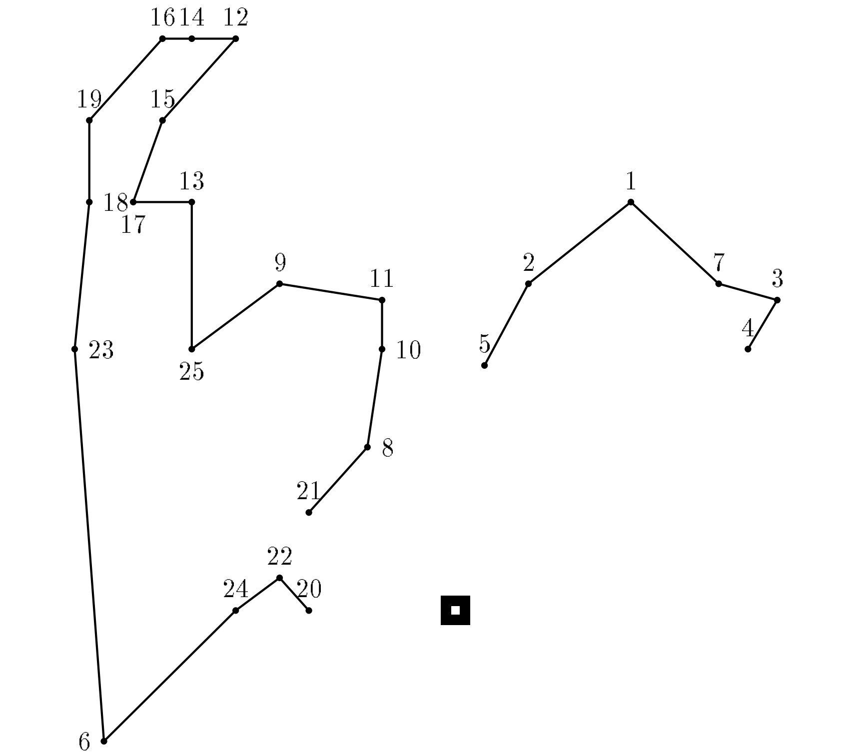
Figure B.7: Optimal solution of C201 with 25 customers.



IP_opt	360.2
LP_opt	360.2

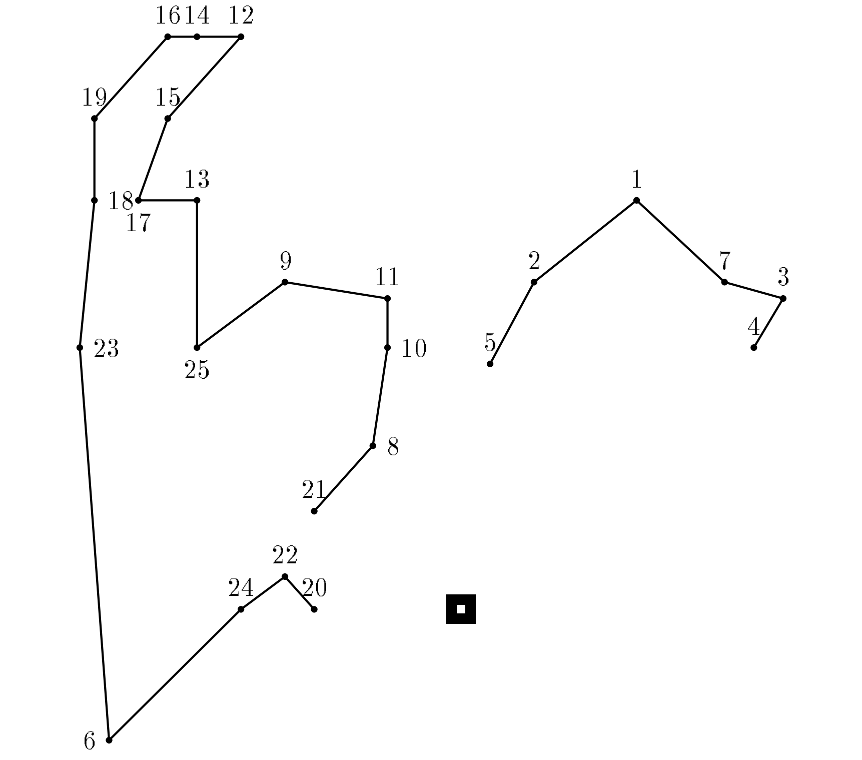
Routes	Length
5 2 1 7 3 4	71.2
49 40 44 46 45 50 47 43 42 41 48	96.4
20 22 24 27 30 29 6 32 33 31 35 37 38 39 36 34 28 26 23 18 19 16 14 12 15 17 13 25 9 11 10 8 21	197.6

Figure B.8: Optimal solution of C201 with 50 customers.



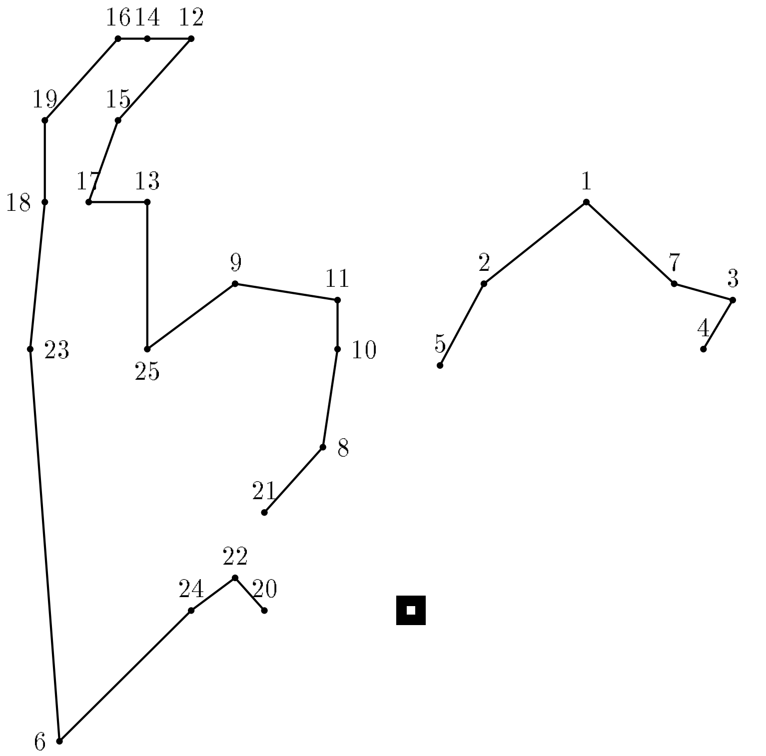
IP_opt	214.7
LP_opt	214.7
Routes	Length
5 2 1 7 3 4	71.2
20 22 24 6 23 18 19 16 14 12 15 17 13 25 9 11 10 8 21	146.0

Figure B.9: Optimal solution of C202 with 25 customers.



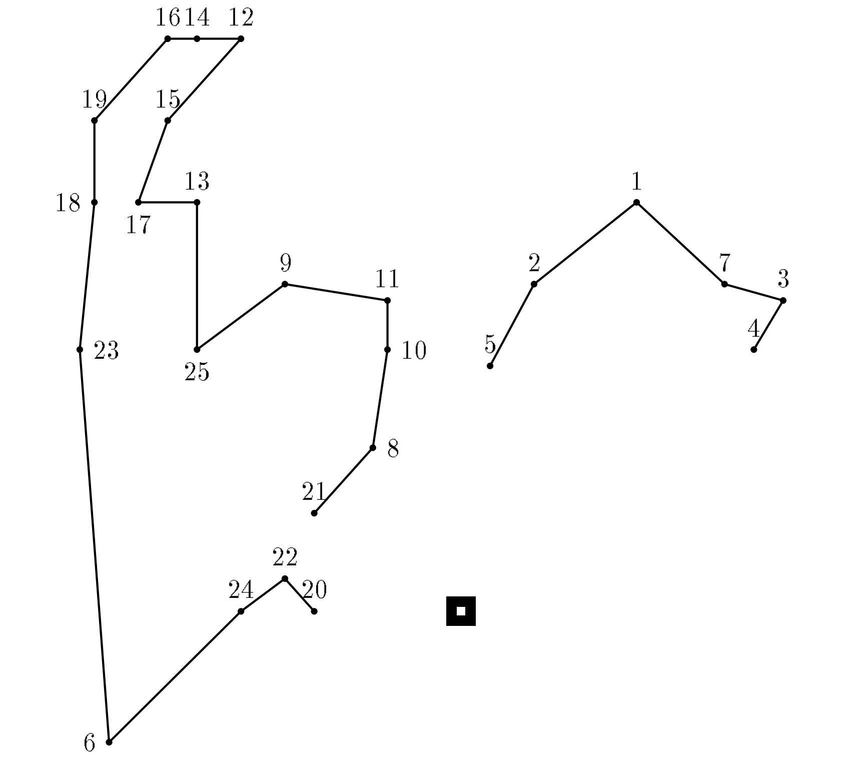
IP_opt	214.7
LP_opt	214.7
Routes	Length
5 2 1 7 3 4	71.2
20 22 24 6 23 18 19 16 14 12 15 17 13 25 9 11 10 8 21	146.0

Figure B.10: Optimal solution of C203 with 25 customers.



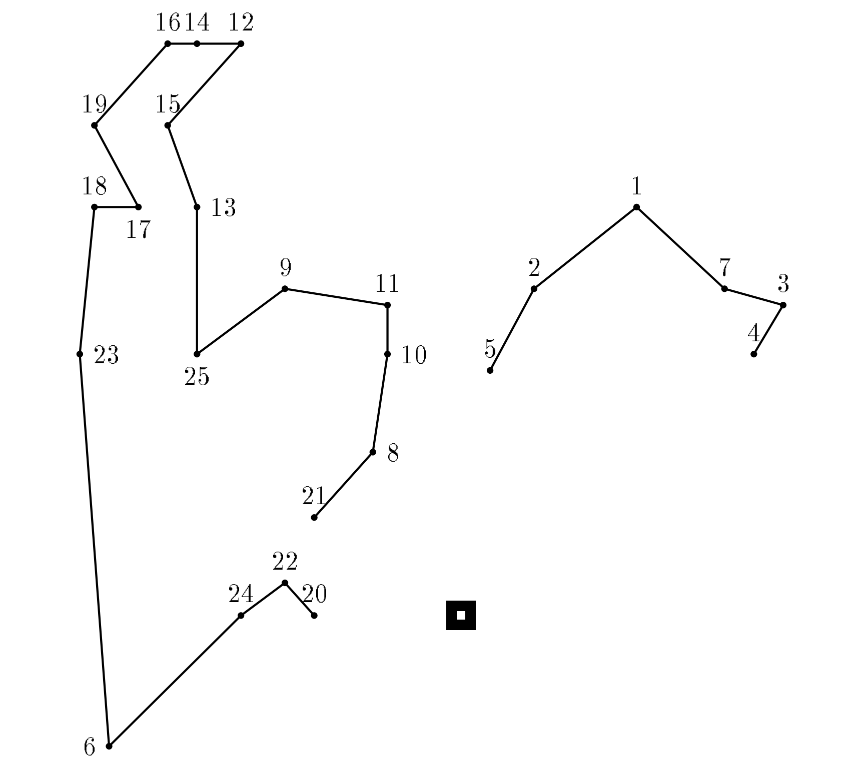
IP_opt	214.7
LP_opt	214.7
Routes	Length
5 2 1 7 3 4	71.2
20 22 24 6 23 18 19 16 14 12 15 17 13 25 9 11 10 8 21	146.0

Figure B.11: Optimal solution of C205 with 25 customers.



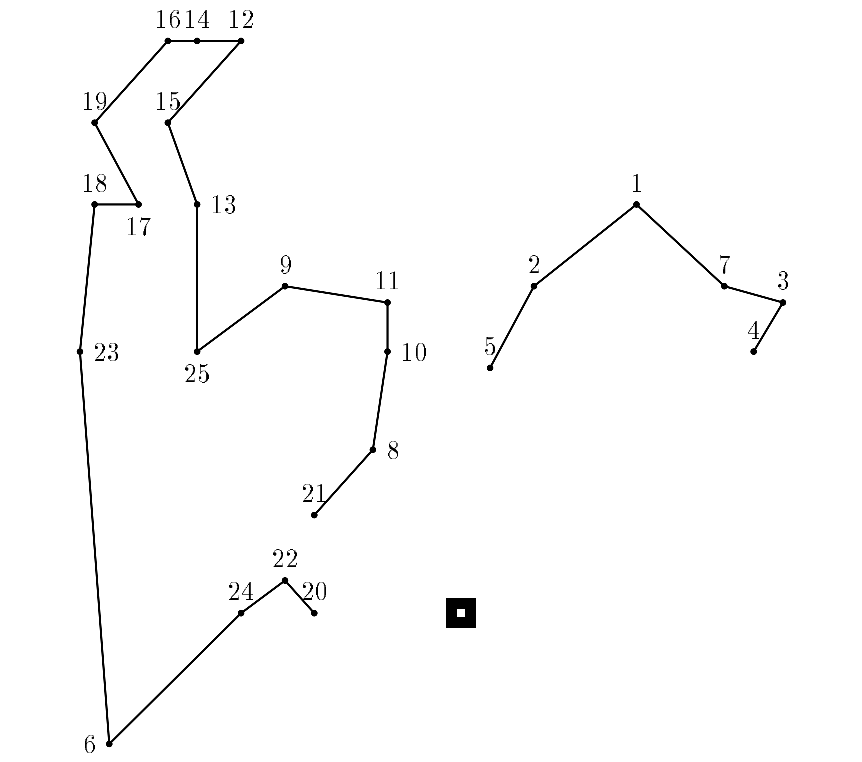
IP_opt	214.7
LP_opt	197.7
Routes	Length
5 2 1 7 3 4	71.2
20 22 24 6 23 18 19 16 14 12 15 17 13 25 9 11 10 8 21	146.0

Figure B.12: Optimal solution of C206 with 25 customers.



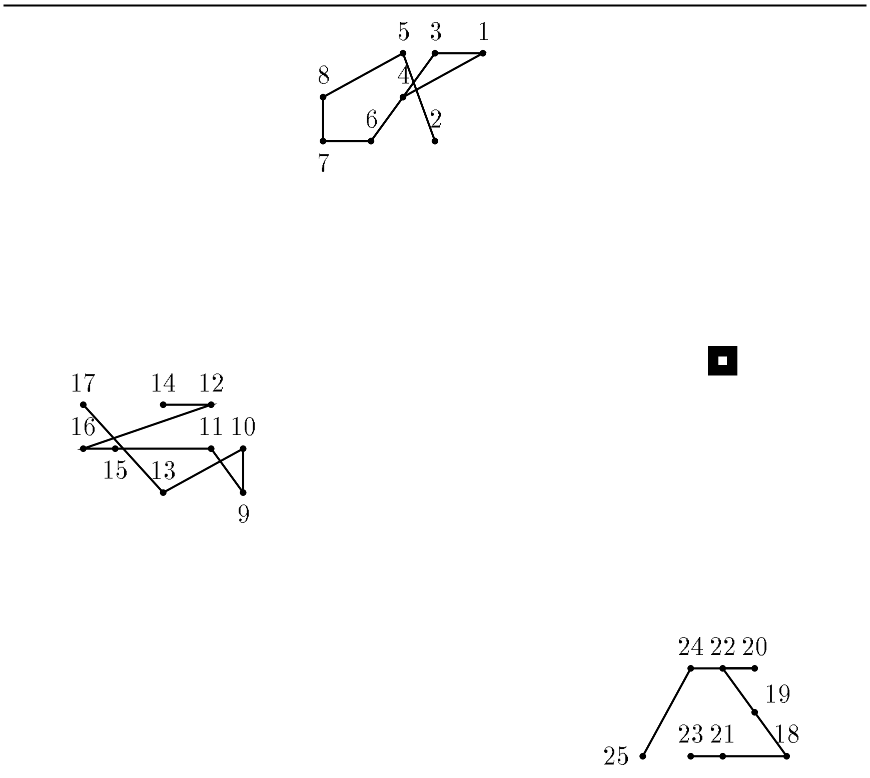
IP_opt	214.5
LP_opt	207.981
Routes	Length
5 2 1 7 3 4	71.2
20 22 24 6 23 18 17 19 16 14 12 15 13 25 9 11 10 8 21	145.8

Figure B.13: Optimal solution of C207 with 25 customers.



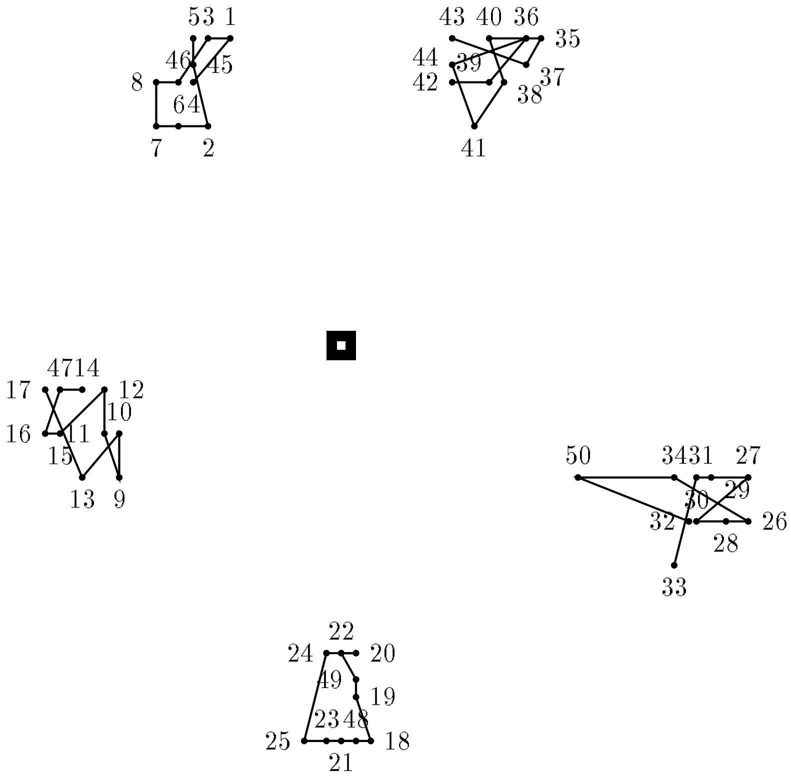
IP_opt	214.5
LP_opt	193.28
Routes	Length
5 2 1 7 3 4	71.2
20 22 24 6 23 18 17 19 16 14 12 15 13 25 9 11 10 8 21	145.8

Figure B.14: Optimal solution of C208 with 25 customers.



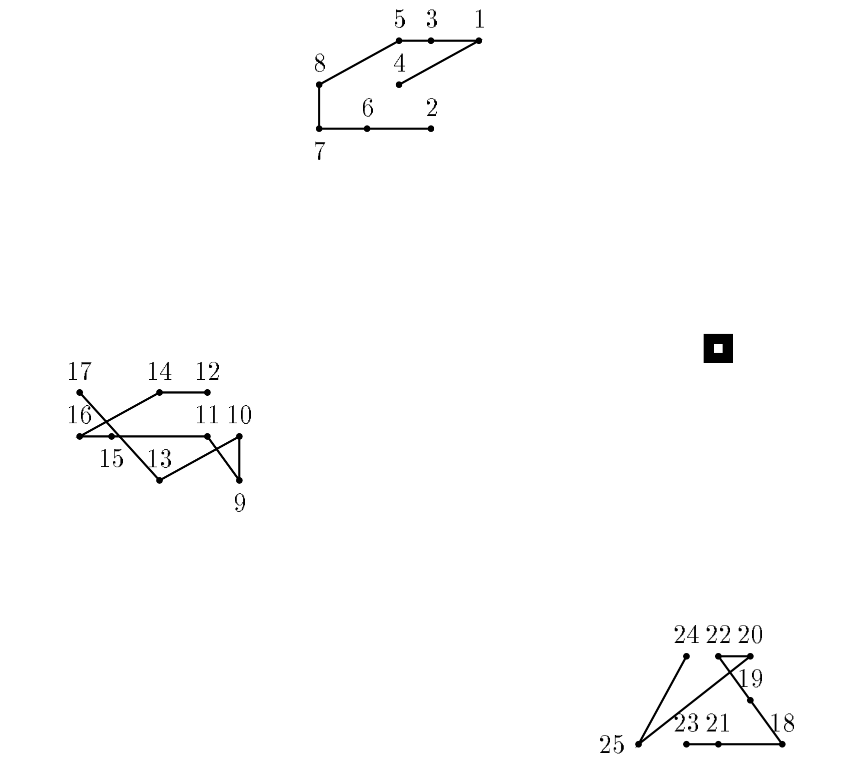
IP_opt	360.2
LP_opt	356.65
Routes	Length
23 21 18 19 22 20 24 25	124.0
2 5 8 7 6 3 1 4	113.4
14 12 16 15 11 9 10 13 17	125.3

Figure B.15: Optimal solution of RC201 with 25 customers.



IP_opt	684.4
LP_opt	670.15
Routes	Length
14 47 16 15 12 11 9 10 13 17	128.1
5 45 2 6 7 8 46 3 1 4	118.9
23 21 18 19 49 22 20 24 25 48	131.3
42 39 36 44 41 38 40 35 37 43	135.2
33 31 29 27 30 28 26 34 50 32	176.3

Figure B.16: Optimal solution of RC201 with 50 customers.



IP_opt	338.0
LP_opt	307.6
Routes	Length
12 14 16 15 11 9 10 13 17	338.0
23 21 18 19 22 20 25 24	
2 6 7 8 5 3 1 4	307.6

Figure B.17: Optimal solution of RC205 with 25 customers.

Appendix C

Ph. D. theses from IMM

1. **Larsen, Rasmus.** (1994). *Estimation of visual motion in image sequences. xiv* + 143 pp.

2. **Rygard, Jens Moberg.** (1994). *Design and optimization of flexible manufacturing systems. xiii* + 232 pp.

3. **Lassen, Niels Christian Krieger.** (1994). *Automated determination of crystal orientations from electron backscattering patterns. xv* + 136 pp.

4. **Melgaard, Henrik.** (1994). *Identification of physical models. xvii* + 246 pp.

5. **Wang, Chunyan.** (1994). *Stochastic differential equations and a biological system. xxii* + 153 pp.

6. **Nielsen, Allan Aasbjerg.** (1994). *Analysis of regularly and irregularly sampled spatial, multivariate, and multi-temporal data. xxiv* + 213 pp.

7. **Ersbøll, Annette Kjær.** (1994). *On the spatial and temporal correlations in experimentation with agricultural applications. xviii* + 345 pp.

8. **Møller, Dorte.** (1994). *Methods for analysis and design of heterogeneous telecommunication networks.* Volume 1-2, *xxxviii* + 282 pp., 283-569 pp.

9. **Jensen, Jens Christian.** (1995). *Teoretiske og eksperimentelle dynamiske undersøgelser af jernbanekøretøjer.* ATV Erhvervsforskerprojekt EF 435. *viii* + 174 pp.

10. **Kuhlmann, Lionel.** (1995). *On automatic visual inspection of reflective surfaces.* ATV Erhvervsforskerprojekt EF 385. Volume 1, *xviii* + 220 pp., (Volume 2, *vi* + 54 pp., fortrolig).

11. **Lazarides, Nikolaos.** (1995). *Nonlinearity in superconductivity and Josephson Junctions.* *iv* + 154 pp.

12. **Rostgaard, Morten.** (1995). *Modelling, estimation and control of fast sampled dynamical systems.* *xiv* + 348 pp.

13. **Schultz, Nette.** (1995). *Segmentation and classification of biological objects.* *xiv* + 194 pp.

14. **Jørgensen, Michael Finn.** (1995). *Nonlinear Hamiltonian systems.* *xiv* + 120 pp.

15. **Balle, Susanne M.** (1995). *Distributed-memory matrix computations.* *iii* + 101 pp.

16. **Kohl, Niklas.** (1995). *Exact methods for time constrained routing and related scheduling problems.* *xviii* + 234 pp.

17. **Rogon, Thomas.** (1995). *Porous media: Analysis, reconstruction and percolation.* *xiv* + 165 pp.

18. **Andersen, Allan Theodor.** (1995). *Modelling of packet traffic with matrix analytic methods.* *xvi* + 242 pp.

19. **Hesthaven, Jan.** (1995). *Numerical studies of unsteady coherent structures and transport in two-dimensional flows.* Risø-R-835(EN) 203 pp.

20. **Slivsgaard, Eva Charlotte.** (1995). *On the interaction between wheels and rails in railway dynamics.* *viii* + 196 pp.

21. **Hartelius, Karsten.** (1996). *Analysis of irregularly distributed points.* *xvi* + 260 pp.

22. **Hansen, Anca Daniela.** (1996). *Predictive control and identification - Applications to steering dynamics.* *xviii* + 307 pp.

23. **Sadegh, Payman.** (1996). *Experiment design and optimization in complex systems.* *xiv* + 162 pp.

24. **Skands, Ulrik.** (1996). *Quantitative methods for the analysis of electron microscope images.* *xvi* + 198 pp.

25. **Bro-Nielsen, Morten.** (1996). *Medical image registration and surgery simulation.* *xxvii* + 274 pp.

26. **Bendtsen, Claus.** (1996). *Parallel numerical algorithms for the solution of systems of ordinary differential equations.* *viii* + 79 pp.

27. **Lauritsen, Morten Bach.** (1997). *Delta-domain predictive control and identification for control.* *xxii* + 292 pp.

28. **Bischoff, Svend.** (1997). *Modelling colliding-pulse mode-locked semiconductor lasers.* *xxii* + 217 pp.

29. **Arnbjerg-Nielsen, Karsten.** (1997). *Statistical analysis of urban hydrology with special emphasis on rainfall modelling.* Institut for Miljøteknik, DTU. *xiv* + 161 pp.

30. **Jacobsen, Judith L.** (1997). *Dynamic modelling of processes in rivers affected by precipitation runoff.* *xix* + 213 pp.

31. **Sommer, Helle Mølgaard.** (1997). *Variability in microbiological degradation experiments - Analysis and case study.* *xiv* + 211 pp.

32. **Ma, Xin.** (1997). *Adaptive extremum control and wind turbine control.* *xix* + 293 pp.

33. **Rasmussen, Kim Ørskov.** (1997). *Nonlinear and stochastic dynamics of coherent structures.* *x* + 215 pp.

34. **Hansen, Lars Henrik.** (1997). *Stochastic modelling of central heating systems. xxii* + 301 pp.

35. **Jørgensen, Claus.** (1997). *Driftsoptimering på kraftvarmesystemer.* 290 pp.

36. **Stauning, Ole.** (1997). *Automatic validation of numerical solutions. viii* + 116 pp.

37. **Pedersen, Morten With.** (1997). *Optimization of recurrent neural networks for time series modeling.* x + 322 pp.

38. **Thorsen, Rune.** (1997). *Restoration of hand function in tetraplegics using myoelectrically controlled functional electrical stimulation of the controlling muscle.* x + 154 pp. + Appendix.

39. **Rosholm, Anders.** (1997). *Statistical methods for segmentation and classification of images. xvi* + 183 pp.

40. **Petersen, Kim Tilgaard.** (1997). *Estimation of speech quality in telecommunication systems.* x + 259 pp.

41. **Jensen, Carsten Nordstrøm.** (1997). *Nonlinear systems with discrete and continuous elements.* 195 pp.

42. **Hansen, Peter S.K.** (1997). *Signal subspace methods for speech enhancement.* x + 214 pp.

43. **Nielsen, Ole Møller.** (1998). *Wavelets in scientific computing. xiv* + 232 pp.

44. **Kjems, Ulrik.** (1998). *Bayesian signal processing and interpretation of brain scans. iv* + 129 pp.

45. **Hansen, Michael Pilegaard.** (1998). *Metaheuristics for multiple objective combinatorial optimization.* x + 163 pp.

46. **Riis, Søren Kamaric.** (1998). *Hidden markov models and neural networks for speech recognition.* x + 223 pp.

47. **Mørch, Niels Jacob Sand.** (1998). *A multivariate approach to functional neuro modeling. xvi* + 147 pp.

48. **Frydendal, Ib.** (1998.) *Quality inspection of sugar beets using vision. iv* + 97 pp. + app.

49. **Lundin, Lars Kristian.** (1998). *Parallel computation of rotating flows. viii* + 106 pp.

50. **Borges, Pedro.** (1998). *Multicriteria planning and optimization. - Heuristic approaches. xiv* + 219 pp.

51. **Nielsen, Jakob Birkedal.** (1998). *New developments in the theory of wheel/rail contact mechanics. xviii* + 223 pp.

52. **Fog, Torben.** (1998). *Condition monitoring and fault diagnosis in marine diesel engines. xii* + 178 pp.

53. **Knudsen, Ole.** (1998). *Industrial vision. xii* + 129 pp.

54. **Andersen, Jens Strodl.** (1998). *Statistical analysis of biotests. - Applied to complex polluted samples. xx* + 207 pp.

55. **Philipsen, Peter Alshede.** (1998). *Reconstruction and restoration of PET images. vi* + 134 pp.

56. **Thygesen, Uffe Høgsbro.** (1998). *Robust performance and dissipation of stochastic control systems.* 185 pp.

57. **Hintz-Madsen, Mads.** (1998). *A probabilistic framework for classification of dermatoscopic images. xi* + 153 pp.

58. **Schramm-Nielsen, Karina.** (1998). *Environmental reference materials methods and case studies. xxvi* + 261 pp.

59. **Skyggebjerg, Ole.** (1999). *Acquisition and analysis of complex dynamic intra- and intercellular signaling events.* 83 pp.

60. **Jensen, Kaare Jean.** (1990). *Signal processing for distribution network monitoring. x* + 140 pp.

61. **Folm-Hansen, Jørgen.** (1999). *On chromatic and geometrical calibration. xiv* + 241 pp.

62. **Larsen, Jesper.** (1999). *Parallelization of the vehicle routing problem with time windows. viii* + 241 pp.

63. **Clausen, Carl Balslev.** (1999). *Spatial solitons in quasi-phase matched struktures. vi* + (flere pag.)

64. **Kvist, Trine.** (1999). *Statistical modelling of fish stocks.* *xiv* + 175 pp.

65. **Andresen, Per Rønsholt.** (1999). *Surface-bounded growth modeling applied to human mandibles.* *xxii* + 116 pp.