Technische Universiteit Delft
Faculteit Elektrotechniek, Wiskunde en Informatica
Delft Institute of Applied Mathematics

Solving the mTSP for fresh food delivery

Report on behalf of the
Delft Institute of Applied Mathematics
as part of obtaining

the title of

BACHELOR OF SCIENCE
in
APPLIED MATHEMATICS

by

Dylan Huizing

Delft, Netherlands
July 2015

BSc verslag TECHNISCHE WISKUNDE

"De mTSP oplossen om vers voedsel te bezorgen"

(Engelse titel: "Solving the mTSP for fresh food delivery")

DYLAN HUIZING

Technische Universiteit Delft

Begeleider

Dr. ir. J.T. van Essen

Overige commissieleden

Dr. D.C. Gijswijt

Dr. J.L.A. Dubbeldam

Dr. ir. M. Keijzer

Juli, 2015

Delft

Abstract

A starting company intends to deliver fresh baby food at home addresses in the area of Zwolle, the Netherlands. The limited time windows of delivery encourage the notion of dividing the addresses over several shifts. Dividing optimal routes over these shifts is shown to be an instance of the multiple traveling salesman problem (mTSP). Three problems are posed, all relating in one way or another to solving the mTSP:

1. Solve the mTSP for the specified delivery area;
2. Find an efficient method for solving an mTSP in which certain nodes may only be visited by certain salesmen;
3. Estimate the largest amount of customers that may be serviced per four hour shift.

A number of methods to solve the mTSP are presented: namely branch and bound, greedy search, simulated annealing and neural networks. The first three of these methods are also modified to handle customer availabilities on specific shifts. The methods are tested and compared within the context of the three different problems. The results of these tests are discussed and solutions to the three problems are suggested.

Contents

Chapter 1

Introduction

1.1 Problem overview

Near the Dutch city of Zwolle, a young company wants to start delivering fresh baby food to home addresses in the approximate area shown in Figure 1.1.

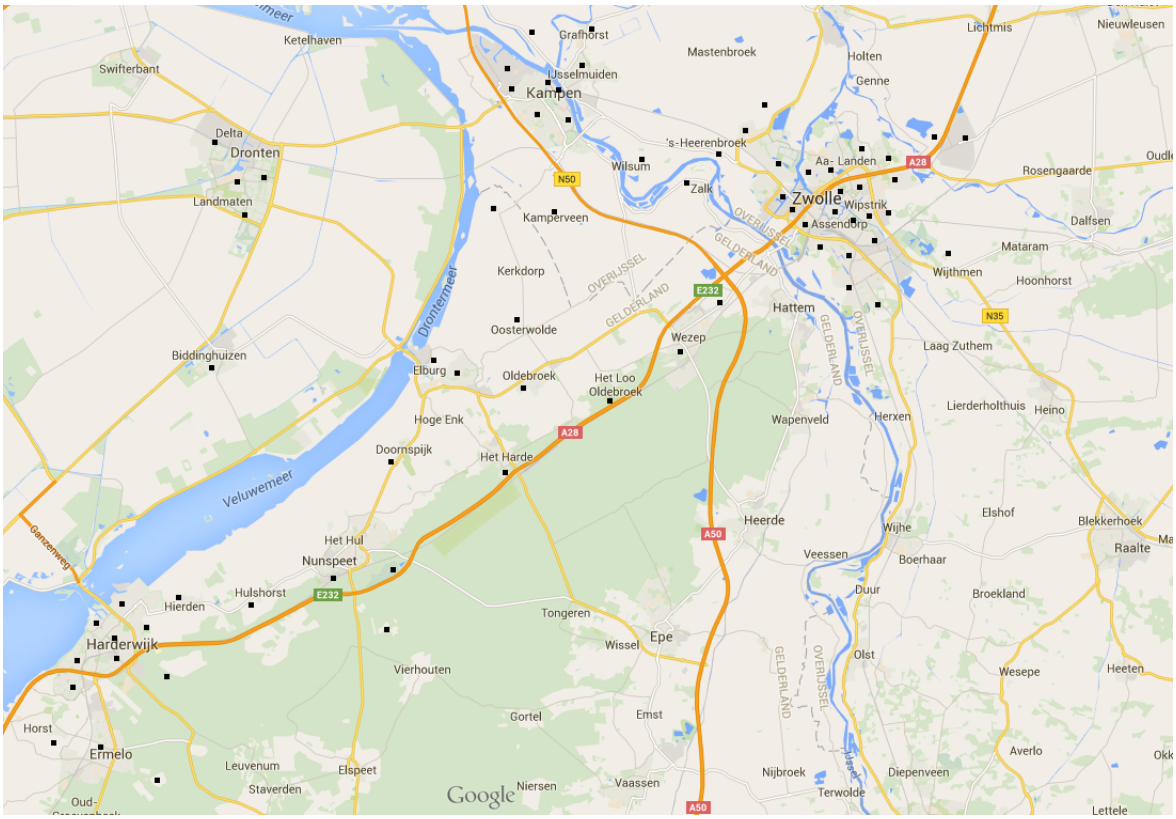

Figure 1.1: Approximate delivery area: Zwolle, Kampen, Dronten, Harderwijk and settlements in between. Four-digit postal codes are marked. `Source: Google Maps`

Their customers are families with young children. Deliveries will be made during evening hours, though not every family may be available to receive a delivery at every day of the week. Additionally, the company intends to have delivery shifts of four hours long (excluding departure from and return to depot) and expects a mere four hours not to be sufficient to serve all of their

customers. As such, they would like to divide the deliveries over several shifts of four hours spread over the week, preferably two shifts. The company has asked three questions that this report seeks to answer:

1. Without prior knowledge of actual customers and their addresses, what is the optimal way to cover the delivery area with a given amount of delivery routes, with optimal meaning fastest?

2. Given some set of home addresses of customers who may or may not be available at certain days, what are the optimal delivery routes? That's to say, what is the way to divide customers over the delivery shifts and determine delivery routes for each shift with the lowest total amount of time spent traveling between addresses?

3. How many deliveries can the company expect to fit within a four hour shift? More specifically, how many customers can the company serve if they have two delivery shifts per week?

The main focus of this research is to find and implement an efficient tool for computing (near) optimal delivery routes and shift divisions. Using this, the answer to the first problem will be an optimal set of routes over the delivery area, the answer to the second problem will be presented in the form of a computation tool and the third problem is answered with an estimated amount of deliveries per shift. For the tool, it is assumed that the company does not have access to large amounts of computational power or time; they want to be able to get a decent set of delivery routes on the fly.

In order to build an efficient tool, several well known methods are tested and compared. These can be used to answer the first question and the best can be implemented in a tool for the second question. The third question can be answered by generating simulations of delivery requests with real world data and testing how large the problems may generally become while still fitting within a given amount of four hour shifts.

A note on notation: the first problem is referred to as 'the simple problem', for it does not yet concern itself with actual addresses or limited availabilities of customers. The second problem is referred to as 'the modified problem' and the third as 'the fitting problem'.

1.2 mTSP as a corresponding mathematical model

This problem of finding an optimal route over two or more shifts can be seen as an instance of the multiple traveling salesman problem (mTSP). The mTSP is a generalisation of the well known traveling salesman problem (TSP) of finding the shortest route that starts in some node of a coherent graph, visits all other nodes exactly once and returns to the starting node. The way that TSP generalises to mTSP is that in mTSP, the nodes must be 'visited once by one of the m salesmen', with all of the m salesmen starting from and ending in the same depot node. Figure 1.2 may serve to give the reader intuition on what 'solving the mTSP' means. Note that by 'shortest route', it is meant that the arcs of such a graph are weighted and the shortest route is the one with the lowest sum of weights of used arcs.

'Visiting all n nodes with m salesmen' may be translated to this problem as 'visiting all n delivery addresses over the course of m shifts'. Finding the optimal way to do so can therefore easily be seen as an instance of the mTSP. Note that no details of the nature of any graph for this problem have yet been given: these details are discussed in Section 2.1. Note also that

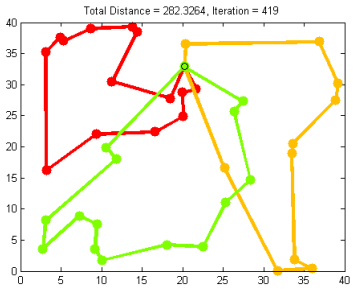

Figure 1.2: The solution to a very general mTSP: all 38 nodes are divided over $m = 3$ tours. `Source: mathworks.com`

this problem differs from the related Vehicle Routing Problem (VRP) as this problem assumes that each delivery vehicle has a boundless capacity. In our context, it is assumed that with a relatively small number of addresses to which relatively small items must be delivered, the total weight or volume of the goods will not pose a problem. If this does present itself as a problem in practice, it is easily solved by increasing the number of shifts. For this and similar purposes, implementations are designed that regard m as a variable, though in practice the reader may assume that $m = 2$.

As the TSP is an NP-complete problem [5] and therefore NP-hard, mTSP must also be NP-hard. Heuristic algorithms are most likely necessary to find a solution within the requested time span.

1.3 Structure of this report

In this report, several methods to solve the mTSP in this specific context are explored. Chapter 2 begins by more thoroughly defining the graph(s) on which the mTSP is to be solved and by justifying the implementation of more than just one solution method. Afterwards, a number of existing methods are presented and their specific implementation for answering both the simple and the modified problem are demonstrated. Chapter 3 presents the setup and results of six experiments with these methods, suggesting an answer to the simple problem and the fitting problem. After this, Chapter 4 discusses these results and explains why simulated annealing is recommended for developing a tool for the modified problem.

Chapter 2

Methods

In this chapter, we first explain how the delivery area and a given set of delivery addresses can be used to form a (near) symmetrically weighted graph for which the mTSP can be solved. Some well known methods are introduced and their exact implementation within the context of these problems is explained. For each method, its implementation for the simple problem is first explained, after which the implementation in the more complicated modified problem is discussed. Details on how well these implementations work are provided in Chapter 3.

2.1 Graph representation of delivery area

In order to answer the real world problems by solving an mTSP, one must first model the real world situation with a weighted, coherent graph of which all nodes must be visited.

In order to answer the simple problem, it is chosen to represent each of the 69 'relevant' four-digit postal codes with a node and observe the completely interconnected graph these nodes form. A 'completely interconnected graph' will henceforth be referred to as a 'complete graph'. Note that the 'relevant' postal codes are the ones that together form the intended delivery area between Zwolle, Kampen, Dronten and Harderwijk and may be more closely inspected in Appendix A. Refer to Figure 2.1 to obtain an intuition for a complete graph of postal code areas.

For the modified problem and the fitting problem, the nodes should represent actual home addresses. These consist of a unique combination of a six-character postal code and a house number. Given these addresses and the depot address, the complete graph between them is observed. Refer to Figure 2.2 to obtain an intuition for a complete graph of customer addresses.

The approximate driving time between any pair of four-digit postal codes or any pair of home addresses can be easily and freely obtained from Google Maps. These driving times can be used as weights for both graphs. Note though that these driving times are generally only approximately symmetric. The small difference that may exist between the obtained time to get from i to j and the time to get from j to i may be caused by such real world factors as one-way traffic (and measurement noise). The obtained times are used as the costs for the (nearly) symmetrical cost matrices describing the graphs.

If the travel times between any given pair of home addresses are not available, an alternative model can be used to form a graph for the modified problem. This model is described in more detail in Appendix B.

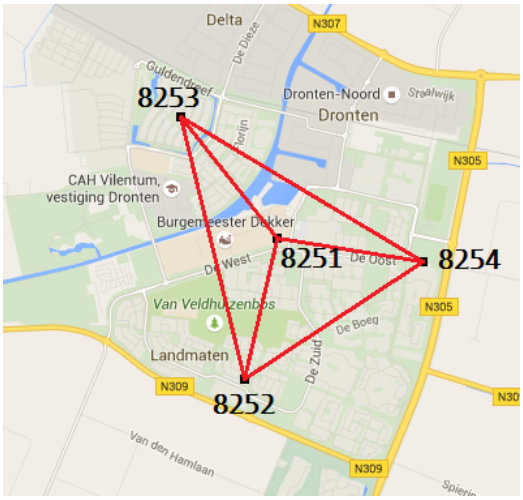

Figure 2.1: A complete graph of four of the 69 postal codes for the simple problem. Note that all addresses in the city of Dronten are divided over these four postal code regions. `Source: Google Maps`

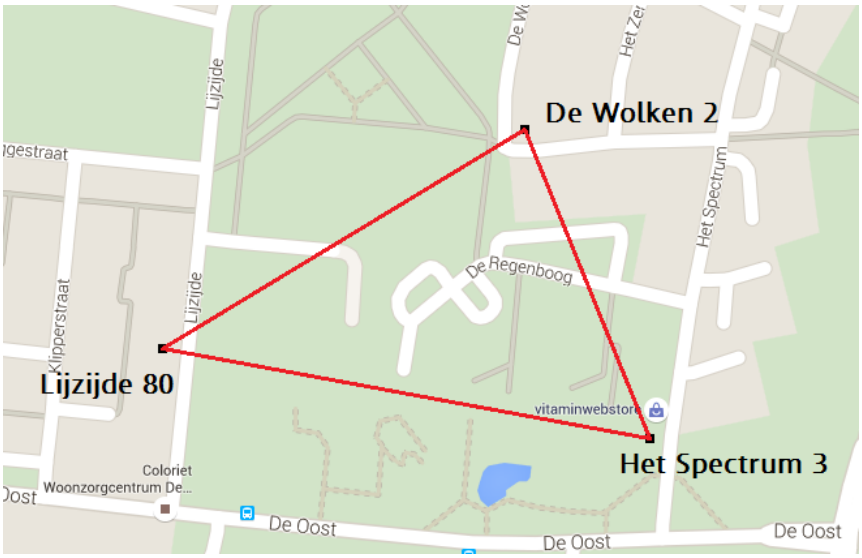

Figure 2.2: A complete graph of three of the n delivery addresses for the modified problem. Note that the address 'Lijzijde 80, Dronten' may also be referred to as '8251CZ 80'. `Source: Google Maps`

2.2 Overview of available methods

With the proper graphs in place, the problem is sufficiently modeled to search for a solution method. Many solution methods have already been studied and tested, but no consensus seems to exist on which method is universally most effective. Sources seem to indicate [1] that the advisable way to go about it is testing whatever methods are readily available and simply judging from experimental results which implementation is most suited to solve this very specific problem.

Solution methods are generally categorised as being either exact or heuristic. Exact meth-

ods are guaranteed to produce the best possible routes, but may take a long time to do so. Heuristic methods can quickly find a solution, but only with the knowledge that they are probably within some percentage of optimality. Researchers seem quick to discard the exact methods whenever the mTSP concerns a graph with more than $n = 100$ nodes [1]. Given that a shift in this problem consists of a single vehicle deployed for no more than four hours, it is quite imaginable that the size of a given problem may be well under $n = 100$. Not only does trying an exact method therefore seem realistic in this context, having the implementation of an exact method is also invaluable in assessing the accuracy of heuristic implementations.

Thus, we first start with implementing an exact method. Branch and bound seems the most prevalent exact method and is therefore selected [1]. Due to the scope of this project, only a limited number of heuristic methods are explored. Simulated annealing is an often used method that comes with an additional benefit: creating a simulated annealing implementation also almost automatically supplies an implementation of greedy search, as will be further explained in Section 2.5.3. A neural network method is chosen for being arguably very different from simulated annealing and thus offering a different perspective on the problem.

2.2.1 A note on the cost functions

For the problem of keeping each shift under four hours, it may seem more appropriate to use the Min-Max cost function rather than the 'classical' cost function. Where the classical cost function seeks to minimise the sum of the costs of each tour, the Min-Max cost function seeks to minimise the cost of the most expensive tour. If this cost is under four hours, automatically all implied shifts are feasible. The classical cost function is still chosen for seeming far more prevalent in literature ([1], [4], [3], [6] and [2]), combined with the fact that the control over the to be introduced parameter p implies much control on how small the difference in costs between tours will be. Note that the cost function has been modified to disregard the time it takes to depart from and return to the depot, as this time is unimportant for the four hour time window of customer availability. Comparing the results found in this report with the results found with the other cost functions may be an interesting direction for future research.

2.3 Branch and bound

As mentioned earlier, it is invaluable to have access to an exact algorithm. Branch and bound is a well known exact method for solving the mTSP. An explanation on how exactly it works, is postponed to the next subsection, where some necessary notation is introduced. It can already be mentioned, however, that branch and bound relies on being able to formulate the mTSP as an integer linear program (ILP). Though the method itself is no different between the simple problem and the modified problem, the ILP formulations will differ significantly.

2.3.1 Simple problem

In order to speak of branch and bound as a solution method for the mTSP, it is necessary to formulate the mTSP as an ILP. Achieving this is attributed to Bektas [1] and his method is described and used here. First, the nodes are numbered, setting the depot node as node 1. Let C_{ij} be the shortest average time to travel from node i to node j. Let the indicator variable x_{ij} be 1 if and only if the arc (i, j) is used in the current solution, i.e., if and only if one of the salesmen traverses arc (i, j). Let x_{ij} be 0 otherwise. For example, the route described in Figure

2.3 corresponds to the following solution:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} & x_{17} & x_{18} & x_{19} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} & x_{27} & x_{28} & x_{29} \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} & x_{37} & x_{38} & x_{39} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} & x_{47} & x_{48} & x_{49} \\ x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} & x_{57} & x_{58} & x_{59} \\ x_{61} & x_{62} & x_{63} & x_{64} & x_{65} & x_{66} & x_{67} & x_{68} & x_{69} \\ x_{71} & x_{72} & x_{73} & x_{74} & x_{75} & x_{76} & x_{77} & x_{78} & x_{79} \\ x_{81} & x_{82} & x_{83} & x_{84} & x_{85} & x_{86} & x_{87} & x_{88} & x_{89} \\ x_{91} & x_{92} & x_{93} & x_{94} & x_{95} & x_{96} & x_{97} & x_{98} & x_{99} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

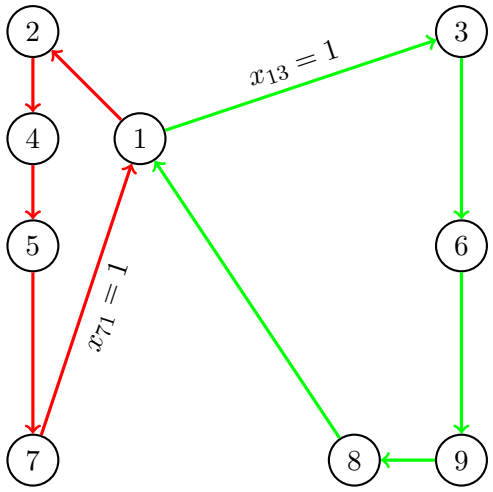

Figure 2.3: Example of a feasible solution for $n = 9$, $m = 2$ and depot at node 1.

Let the potential u_i be the amount of nodes already visited by a salesman when he arrives at node i. Observing the example in Figure 2.3, we see that

$$u_2 = 1,\ u_4 = 2,\ u_5 = 3,\ u_7 = 4; \qquad u_3 = 1,\ u_6 = 2,\ u_9 = 3,\ u_8 = 4$$

Let p be the maximum number of nodes a salesman is allowed to visit, including the return to the home depot. The value of p can manually be set to meet the needs of the context: if 10 addresses must be split in two shifts of 5, this is ensured by setting $p = 6$. Setting p higher makes the balance of the split more lenient. The need for x_{ij} to be either 0 or 1 makes this an

integer programming problem and the basic ILP can be written as follows:

$$\begin{aligned}
\min \quad & \sum_{i=2}^{n}\sum_{j=2}^{n} C_{ij}x_{ij} \\
s.t. \quad & \sum_{j=2}^{n} x_{1j} = m && (2.1)\\
& \sum_{i=2}^{n} x_{i1} = m && (2.2)\\
& \sum_{i=1}^{n} x_{ij} = 1, \qquad j = 2,\ldots,n && (2.3)\\
& \sum_{j=1}^{n} x_{ij} = 1, \qquad i = 2,\ldots,n && (2.4)\\
& u_i - u_j + px_{ij} \leq p-1, \qquad i = 2,\ldots,n,\ j = 2,\ldots,n && (2.5)\\
& x_{ij} \in \{0,1\}, \qquad i,j = 1,\ldots,n && (2.6)\\
& u_i \in \{1,2,\ldots,p-1\}, \qquad i = 2,\ldots,n
\end{aligned}$$

Condition (2.1) ensures that m routes start at the depot node ($i = 1$). Condition (2.2) ensures that m routes also end there. Conditions (2.3) and (2.4) together ensure that all other nodes are visited exactly once. These four conditions alone are not sufficient: they would allow a solution to contain subtours, that's to say, cycles not connected to the depot. Refer to Figure 2.4 for an example of a solution with a subtour. Condition (2.5), due to Miller [4], eliminates the possibility of such subtours.

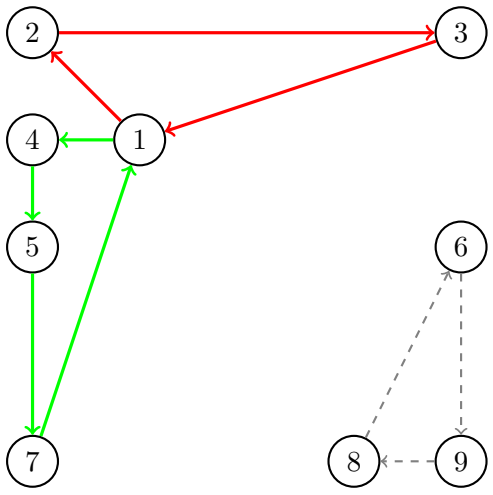

Figure 2.4: Example of a solution with a subtour (dashed).

This ILP can be solved by means of branch and bound. This algorithm examines LP relaxations of the problem: that's to say, copies of the problem in which the condition is dropped that x_{ij} and u_i should be integer. Instead, we allow them to be real values between 0 and 1 for x_{ij} and between 1 and $(p-1)$ for u_i. LP relaxations can be solved using the powerful (dual) simplex algorithm and if this solution contains some $0 < x_{ij} < 1$, it can be branched into the cases $x_{ij} = 0$ and $x_{ij} = 1$. Such a case is represented by taking the parent ILP and

adding the condition $x_{ij} = 0$ or $x_{ij} = 1$. These branches or cases are stored in an active list, from which the first in line is continuously taken and its LP relaxation solved. If the currently investigated branch has an infeasible LP relaxation, the branch is pruned by infeasability. If the LP relaxation yields a solution with a cost higher than the currently best known cost for an integer solution, its lineage is not worth investigating, as the cost of a child branch is by definition higher than that of the less restricted parent branch. In this case, the branch is pruned by bound. When the LP relaxation actually yields an integer solution, the branch is pruned by optimality, because there is no reason to search its lineage for another integer solution with lower cost (for the aforementioned reason). The entire tree is searched if and only if the active list has become empty. At this point, the best found integer solution can be presented.

A remaining question is how to search this tree. Which branch in the active list is 'the first in line'? It is possible to search the tree width-first, hoping to close off the biggest possible chunks of tree quickly and avoiding the meticulous investigation of numerous deep branches. On the other hand, it is also possible to search the tree depth-first, hoping to find a good integer solution quickly so that many of the initial branches can be directly pruned by bound. Combining both merits, one could attempt a 'switch over' method: searching depth-first until an integer solution is found, then switching over to width-first search. Width-first search is attained by, when creating new branches, adding them to the bottom of the active list. Depth-first search is attained by simply adding them to the top of the list instead. Due to the implementational ease and irreputability, all three search methods are tested and compared in Section 3.1.

2.3.2 Modified problem

As announced earlier, solving the modified problem requires a different ILP formulation. Suppose for example that the customers at node q, r and s are unavailable during the first of the m shifts. These customers must be visited on other shifts. But though one can construct m shifts from the solution given by the previous method, it is ambiguous on *which* of the shifts some node i is visited and thus impossible to ensure that node q, r and s are placed on specific shifts. The fact that some x_{ij} equals 1 holds no information over 'which' salesman traverses the arc (i, j).

We therefore instead introduce the indicator variable x_{ijk}, which is equal to 1 if and only if the arc (i, j) is traversed by salesman k and 0 otherwise. In other words: if node q cannot be visited during shift $k = 1$, we must ensure that $x_{iq1} = 0$ for $i = 1, 2, \ldots, n$. This leads to the following ILP formulation:

$$\begin{aligned}
\min \quad & \sum_{i=2}^{n}\sum_{j=2}^{n}\sum_{k=1}^{m} C_{ij}x_{ijk} \\
s.t. \quad & \sum_{j=2}^{n} x_{1jk} = 1, \qquad k = 1, \ldots, m && (2.7)\\
& \sum_{i=2}^{n} x_{i1k} = 1, \qquad k = 1, \ldots, m && (2.8)\\
& \sum_{i=1}^{n}\sum_{k=1}^{m} x_{ijk} = 1, \qquad j = 2, \ldots, n && (2.9)\\
& \sum_{j=1}^{n}\sum_{k=1}^{m} x_{ijk} = 1, \qquad i = 2, \ldots, n && (2.10)
\end{aligned}$$

$$\sum_{i=1}^{n} x_{ijk} - \sum_{l=1}^{n} x_{jlk} = 0, \qquad j = 2, \ldots, n,\ k = 1, \ldots, m \tag{2.11}$$

$$u_i - u_j + p x_{ijk} \leq p - 1, \qquad i = 2, \ldots, n,\ j = 2, \ldots, n,\ k = 1, \ldots, m \tag{2.12}$$

$$x_{ijk} \in \{0, 1\}, \qquad i, j = 1, \ldots, n,\ k = 1, \ldots, m \tag{2.13}$$

$$u_i \in \{1, 2, \ldots, p-1\}, \qquad i = 2, \ldots, n$$

Condition (2.7) demands that the depot is left exactly once by each of the m salesmen, similar to how condition (2.1) demanded that the depot had m outgoing arcs (leaving it to the interpretation that each of these arcs belongs to a different salesman). Analogously, condition (2.8) demands each of the m salesmen returns to the depot. Condition (2.9) and (2.10) demand that each other node is visited and left by exactly one salesman. Condition (2.11) states that some two sums must be equal. This is to be read as follows: if node j is visited by salesman k (so the first sum equals 1), then node j must also be left by salesman k (so the second sum must equal 1). It also implies that if salesman k does *not* visit j, he may not leave from node j either, thus both sums must equal 0. Condition (2.12) is again Miller's subtour elimination constraint, applied to each of the m tours.

Note that no form of customer availability appears represented in this ILP. Algorithms which solve LPs and ILPs necessarily include the option to specify for each variable what its upper and lower bounds are. If node h is not to be visited on shift s, it must simply be insured that $x_{ihs} = 0$ for $i = 1, \ldots, n$ by setting both the upper bounds and lower bounds of all x_{ihs} to 0. Similarly, all x_{hjs} can be set to 0: though it is not necessary to demand that salesman s cannot leave node h if he cannot even reach node h in the first place, it may increase the speed of computation in reducing the amount of variables over which optimisation is to take place.

2.3.3 Additional constraints

When speaking of increasing the speed of computation, adding additional constraints to the ILPs may be of interest. While the conditions given in Section 2.3.1 and Section 2.3.2 are sufficient to describe the mTSP for the simple problem and the modified problem respectively, solving an LP relaxation using only these constraints often produces a theoretical lower bound on a branch far lower than the actual solution found on that branch. Adding additional constraints may theoretically seem superfluous, but in practice helps find far more 'realistic' solutions to LP relaxations. This in turn allows us to prune branches by bound earlier, which may drastically reduce computation time. Two of these additional constraints for the simple problem are explored here. Afterwards, an additional constraint for the modified problem is introduced. The question of which of these additional constraints actually reduce computation time, is addressed in Section 3.2.

An upper bound on the sum of potentials

During testing, LP relaxations at the start of the branch and bound process were observed. Examining the solutions of these LP relaxations in the simple problem showed values of the potentials u_i that were slightly too high: that's to say, they were indeed smaller than or equal to $(p-1)$, but often only by a small margin, causing their sum to exceed any realistic scenario. One could therefore restrict not only the value of all u_i, but also their sum.

An upper bound can be based on the 'worst case scenario', that's to say, the route with the highest sum of potentials. For example, if $(n, m, p) = (10, 4, 5)$, it seems the highest possible sum

$\sum_{i=2}^{n} u_i$ is attained when two salesmen fully use their potential and the remaining node is visited by the third salesman, i.e., $\sum_{i=2}^{n} u_i = 1+2+3+4+1+2+3+4+1 \geq 1+2+1+2+1+2+1+2+3$.

More generally speaking: for a given (n, m, p), to attain the highest sum of potentials, P salesmen must contribute terms $1+2+3+\ldots+(p-1) = \frac{1}{2}p(p-1)$ to the sum of potentials with P the greatest number of times such a sequence fits in the vector $(u_2, u_3, \ldots, u_n)$. This vector is $n-1$ elements long and the rows are $p-1$ elements long, so $P = \lfloor (n-1)/(p-1) \rfloor$. In addition, all the nodes that are not visited by these P salesmen should be visited by one salesman, who thus visits only $0 \leq q < (p-1)$ nodes. Note that q, the number of remaining nodes, is obtained by removing $p-1$ elements at a time from the vector that was $n-1$ elements long until this can no longer be done, so $q = (n-1) \mod (p-1)$. We know for these final q terms that $1+2+3+\ldots+q = \frac{1}{2}(q+1)q$.

In the generalisation, it appears that an upper bound on the sum of potentials could be given by the following additional constraint:

$$\sum_{i=2}^{n} u_i \leq \left\lfloor \frac{n-1}{p-1} \right\rfloor \cdot \frac{1}{2}p(p-1) + \frac{1}{2}(q+1)q, \; q = (n-1) \bmod (p-1) \tag{2.14}$$

To validate this statement, the following theorem is stated and proven:

Theorem 1. *If $u_i \in \{1, 2, \ldots, p-1\}$, $i = 2, \ldots, n$ are potentials of a solution of the mTSP with m the number of salesmen, n the number of nodes and p the maximum number of nodes that can be visited by a salesman (including the return to the home depot), with m and p large enough that solutions are possible, then*

$$\sum_{i=2}^{n} u_i \leq \left\lfloor \frac{n-1}{p-1} \right\rfloor \cdot \frac{1}{2}p(p-1) + \frac{1}{2}(q+1)q, \; q = (n-1) \; mod \; (p-1)$$

Proof. Let $\mathbf{u}^1 = (u_2^1, u_3^1, u_4^1, \ldots u_n^1)$ and $\mathbf{u}^2 = (u_2^2, u_3^2, u_4^2, \ldots u_n^2)$ be the potentials belonging to some feasible solutions 1 and 2. Let $\mathbf{u}^1$ be a permutation of $\mathbf{v} = (1, 2, \ldots, p-1, \; 1, 2, \ldots, p-1, \ldots, p-1, 1, 2, \ldots, q)$. Let $\mathbf{u}^2$ not be a permutation of $\mathbf{v}$. It suffices to show two things:

1. $$\sum_{u_i \in \mathbf{u}^1} u_i = \left\lfloor \frac{n-1}{p-1} \right\rfloor \cdot \frac{1}{2}p(p-1) + \frac{1}{2}(q+1)q, \; q = (n-1) \bmod (p-1)$$

2. $$\sum_{u_i \in \mathbf{u}^1} u_i \geq \sum_{u_i \in \mathbf{u}^2} u_i$$

For the first item, observe that because $\mathbf{u}^1$ is a permutation of $\mathbf{v}$, the sum of their entries are equal:

$$\sum_{u_i \in \mathbf{u}^1} u_i = 1+2+\ldots+(p-1)+1+2+\ldots+(p-1)+1+2+\ldots$$

That this sum equals the right-hand side of the equation, is considered evident from the discussion earlier in this section.

For the second item, let us examine the example $(n, m, p) = (10, 4, 5)$ with $\mathbf{u}^1 = (1, 2, 3, 4, 1, 2, 3, 4, 1)$

and $\mathbf{u}^2 = (1, 2, 1, 2, 1, 2, 1, 2, 3)$. If the elements of $\mathbf{u}^1$ and $\mathbf{u}^2$ are then sorted in ascending order, this results in the vectors

$$\mathbf{s}^1 = (1, 1, 1, 2, 2, 3, 3, 4, 4) \quad \text{and} \quad \mathbf{s}^2 = (1, 1, 1, 1, 2, 2, 2, 2, 3)$$

Note that each element of $\mathbf{s}^1$ is larger than or equal to the element of $\mathbf{s}^2$ at the same index: $(\mathbf{s}^1)_i \geq (\mathbf{s}^2)_i$ for $i = 1, 2, \ldots, n$. So trivially, the sum of the elements is also larger or equal:

$$\sum_{u_i \in \mathbf{u}^1} u_i = \sum_{s_i \in \mathbf{s}^1} s_i \geq \sum_{s_i \in \mathbf{s}^2} s_i = \sum_{u_i \in \mathbf{u}^2} u_i$$

For the more general case, observe three things:

1. A worst case $\mathbf{u}^1$ has a minimal amount of salesmen, thus a minimal amount of entries 1.
2. A worst case $\mathbf{u}^1$ has, by construction, a maximal amount of entries $p - 1$.
3. For each entry of a non-worst case $\mathbf{u}^2$ with value $z < p - 1$ for which $\mathbf{u}^2$ has more entries with value z than $\mathbf{u}^1$ does, there exists a corresponding entry in $\mathbf{u}^1$ with value $z < y \leq p-1$ for which $\mathbf{u}^1$ contains more entries with value y. In more intuitive words: $\mathbf{u}^2$ may only have an additional entry with value z at the cost of having less entries with value $y > z$. In the above example, $\mathbf{u}^2$ can only have a 1, a 2 and a 2 more than $\mathbf{u}^1$ because it has a 3, a 4 and a 4 less.

From these things it follows, again by observing $\mathbf{s}^1$ and $\mathbf{s}^2$ as sorted versions of $\mathbf{u}^1$ and $\mathbf{u}^2$, that

$$\sum_{u_i \in \mathbf{u}^1} u_i = \sum_{s_i \in \mathbf{s}^1} s_i \geq \sum_{s_i \in \mathbf{s}^2} s_i = \sum_{u_i \in \mathbf{u}^2} u_i$$

□

Restriction of symmetry

The next additional constraint for the simple problem concerns not the potentials, but the values of x_{ij} themselves. Suppose that the solution of an LP relaxation in the simple problem has, for a given $i, j \neq 1$, $x_{ij} \approx x_{ji} \approx 1$. This implies that the rounded solution has a subtour between node i and j, which is unwanted. Though condition (2.5) should guard against the appearance of subtours, the first LP relaxation in a branch and bound process surprisingly often had several of such approximate subtours. To eliminate these, one could introduce the additional constraint

$$x_{ij} + x_{ji} \leq 1, \quad i, j = 2, 3, \ldots, n \tag{2.15}$$

This inequality allows x_{ij}, x_{ji} or neither to equal 1, but not both.

Single tour allocation

In defining the modified problem, it may seem intuitive to include a condition that demands that each arc is traversed by no more than one salesman. Such a condition could be formulated as follows:

$$\sum_{k=1}^{m} x_{ijk} \leq 1, \qquad i = 1, \ldots, n, \ j = 1, \ldots, n \tag{2.16}$$

As it turns out, this is already implied by conditions (2.9) and (2.10). If (2.9) states that the node $j \neq 1$ must be visited by one salesman and no more than one salesman, then it is a special case that it may not be visited by more than one salesman coming from some node $i \in \{1, \ldots, n\}$. This implies condition (2.16) for $i = 1, \ldots, n$ $j = 2, \ldots n$. The case $j = 1, i = 1, \ldots, n$ follows similarly from (2.10). This deems (2.16) unnecessary but true and therefore eligible to be included as additional constraint to possibly speed up the computational process.

2.4 Greedy search

Though branch and bound definitely finds the optimal solution, the time it takes to do so is generally large, even with the inclusion of effective additional constraints. Greedy search is a relatively simple way to more quickly search the space of all feasible solutions, albeit without guarantee that the optimal solution will be found. The solution space consists of all valid tours with the given values of n (the number of cities), m (the number of salesmen) and p (the maximum number of nodes a salesman may visit, including the return to depot). Solutions are represented differently from the previous section: if $n = 4$, $m = 2$ and $p = 4$, then the solution

$$\begin{matrix} 2 & 4 & 0 \\ 3 & 0 & 0 \end{matrix}$$

represents the first salesman visiting node 2 and then 4 before returning to the depot and the second salesman only visiting node 3. The depot node is indicated with 1: it is left out of the representation, because it is known that each tour starts at and ends with a 1. The zeros may be ignored when interpreting this representation, but are there to indicate that under the given value of p, the tour of a salesman may have been longer.

2.4.1 Simple problem

A neighbour solution N to a given solution B is defined as any solution found by performing one of the following five operations on B. The first four are due to Király [3], the fifth was added to increase interroute exchange.

1. Intraroute inversion: Pick a section of a tour and invert its order. Example:

$$\begin{matrix} 2 & \mathbf{3} & \mathbf{4} & \mathbf{5} & \mathbf{6} \\ 7 & 0 & 0 & 0 & 0 \end{matrix} \quad \rightarrow \quad \begin{matrix} 2 & 6 & 5 & 4 & 3 \\ 7 & 0 & 0 & 0 & 0 \end{matrix}$$

2. Intraroute switching: Pick two sections within a tour and exchange their locations. Example:

$$\begin{matrix} \mathbf{2} & \mathbf{3} & 4 & 5 & \mathbf{6} \\ 7 & 0 & 0 & 0 & 0 \end{matrix} \quad \rightarrow \quad \begin{matrix} 6 & 4 & 5 & 2 & 3 \\ 7 & 0 & 0 & 0 & 0 \end{matrix}$$

3. Intraroute insertion: Pick a section within a tour and put it in another place within that tour. Example:

$$\begin{matrix} & & & \searrow & \\ 2 & \mathbf{3} & 4 & 5 & 6 \\ 7 & 0 & 0 & 0 & 0 \end{matrix} \quad \rightarrow \quad \begin{matrix} 2 & 4 & 5 & 3 & 6 \\ 7 & 0 & 0 & 0 & 0 \end{matrix}$$

4. Interroute switching: Pick a section within one tour and a section within another and exchange their locations. These sections are selected to ensure that no route ends up with less than 1 or more than $(p-1)$ nodes on it. Example:

$$\begin{matrix} 2 & \mathbf{3} & \mathbf{4} & \mathbf{5} & 6 \\ \mathbf{7} & 0 & 0 & 0 & 0 \end{matrix} \quad \begin{matrix} \rightarrow \\ \rightarrow \end{matrix} \quad \begin{matrix} 2 & 7 & 6 & 0 & 0 \\ 3 & 4 & 5 & 0 & 0 \end{matrix}$$

5. Interroute transfer: Pick a section within one tour and put it at the end of another tour. This section and the receiving route are selected to ensure that no route ends up with less than 1 or more than $(p-1)$ nodes on it. Example:

$$\begin{matrix} 2 & \mathbf{3} & \mathbf{4} & \mathbf{5} & 6 \\ 7 & 0 & 0 & 0 & 0 \end{matrix} \quad \begin{matrix} \rightarrow \\ \rightarrow \end{matrix} \quad \begin{matrix} 2 & 6 & 0 & 0 & 0 \\ 7 & 3 & 4 & 5 & 0 \end{matrix}$$

The cost of a tour is, again, the sum of the weights on the used arcs, excluding the arcs connected to the depot.

The basic idea of greedy search is to pick a random starting solution, compare it with a random neighbour solution, accept it as the current solution if it's cheaper, stop this process when it appears to be stuck in a local optimum, repeat this process r times and return the cheapest solution ever encountered. In more detail, the used algorithm is the following:

1. Select a random starting solution in the solution space. This can be achieved by generating a random permutation of the vector $(2, 3, \ldots, n)$ and cutting it into m pieces, randomly choosing where to cut it, respecting that no piece may become longer than $(p-1)$ elements. Set this to be the current best solution B. If this is the first run, set this to also be the overall best solution O.

2. Pick a random neighbour solution N of B by randomly choosing one of the five operations (with equal probability) and randomly choosing one of the possible ways to perform that operation (with equal probability). If N has a lower cost, set $B = N$. Also check if N is cheaper than O: if so, set $O = N$.

3. Repeat the previous step until B hasn't changed in the last τ attempts. When this happens, it is assumed that a local optimum has been found. The value of the 'patience' parameter τ can be experimented with.

4. Repeat all above steps r times, with r the number of runs, another parameter with which can be experimented.

When all r runs are completed, one can simply return O, the overall best solution ever encountered in the process. Though experimentation with the value of r is possible, we set $r = 1$ because it appears to be conventional.

2.4.2 Modified problem

Greedy search becomes only slightly more complicated when introducing the availability of customers during certain shifts. This is because this formulation of solutions already allocates each tour to its own row k in the representation and therefore to salesman k. The interroute operations need therefore only be updated such that customers are not placed on unwanted tours.

It advisable that the initial random solution also does not violate the customer availabilities. It is arguably not necessary, as the interroute operations may eventually place each of the customers on a wanted tour, but it ensures the convergence to a feasible solution. Especially important to note in this is that initially ignoring the customer availabilities may place the starting solution in a local optimum far better than any more constrained neighbour solution may offer. To randomly initialise a feasible solution, we choose to perform a greedy algorithm:

Initialisation For each of the customers, determine on how many of the m shifts they are available. Find the minimum of these numbers and name it $M \leq m$. If the problem is well posed, initially $M \geq 1$.

Repeat Randomly select a customer who is available on only M shift(s). Add it randomly to one of the tours on which he is available. Set his number of available shifts to ∞ so that he cannot be selected again. Redetermine M for the customers still left. Repeat until either all customers are validly placed on a tour or until one of the tours is 'full' (that's to say, the number of customers placed on that tour equals $(p-1)$).

Case a If all customers are validly placed on a tour, a feasible random starting solution is found. Terminate the algorithm.

Case b If instead a tour k is full, make all remaining customers unavailable for tour k. Update the amount of shifts on which each of the remaining customers is available and redetermine M. If this causes $M = 0$, the current tours are infeasible. In that case, remove all placed customers from their tours, reset the customer availabilities to their initial setting and restart from **Initialisation**. If instead still $M \geq 1$, do not reset and instead continue from **Repeat**.

Stop criterion If **Case b** causes 1000 resets before a feasible solution is found, assume that the problem is not well posed and terminate the algorithm.

Though this algorithm corresponds to an intuitive greedy allocation, we make no guarantees that it will find a feasible starting solution in the general case. Fortunately, for the case $m = 2$, we can guarantee that a feasible starting solution will be found as long as the problem is well posed. To this end, observe that if initially $M = 2$, none of the customers have a preference for any tour and can simply be allocated randomly. If initially $M = 1$, the customers that do have a preference will first be placed on their preferred tour, after which the remaining customers can be allocated randomly to the remainder of the two tours. An example of how this algorithm may theoretically fail for $m \geq 3$ is considered simple and irrelevant enough to be omitted.

2.5 Simulated annealing

Though greedy search works far more quickly than branch and bound (see Section 3), it is not considered a very sophisticated heuristic method. Simulated annealing is considered closely related and more sophisticated, that's to say, less sensitive to getting stuck in local optima. Its concept is based on the notion that molecules in molten metal may move very freely when the metal is still very hot, but become more and more locked in place as the metal begins to cool. In the context of searching a solution space, the parallel lies in allowing the current solution to be replaced with a neighbour solution with very little heed to optimality at first, but forcing it into a local optimum as time progresses. The potential lies in allowing the current solution to move very freely in search of the area of a global optimum, then taking away that freedom to push it towards that global optimum. Note that the probability with which a more expensive solution is accepted depends on how much more expensive it is. Refer to Figure 2.5.1 to obtain an intuition for these varying acceptance probabilities.

2.5.1 Simple problem

Simulated annealing searches the same solution space as greedy search does and we use the same definition of a neighbour solution according to the same five operations presented in Section 2.4.1. However, the method of searching differs slightly. In more detail, the used algorithm for the simple problem is the following:

1. Select a random starting solution B in the solution space, in the same way as with greedy search. Set this to be the current best solution O.

2. Set the *temperature* T to some fixed starting temperature T_0. T_0 is a parameter with which can be experimented.

3. With the current value T of the temperature, pick a random neighbour solution N of B. If N has a lower cost, set $B = N$. If N has a lower cast than O, set $O = N$. If N has a cost greater or equal to that of B, set $B = N$ with a probability

$$p(N, B, T) = \exp(-(cost(N) - cost(B))/T)$$

4. Perform step 3 t times with this temperature, where t is a parameter with which can be experimented. Then set the new temperature to be $T_{new} = \alpha \cdot T_{old}$.

5. When the temperature drops below a value T_{end}, terminate the algorithm and return O.

Simulated annealing comes with the difficulty of having to find good values for T_0, T_{end}, t and α. While α was set to $\alpha = 0.95$ for it being common practice, values for other parameters were experimented with in Section 3.3.

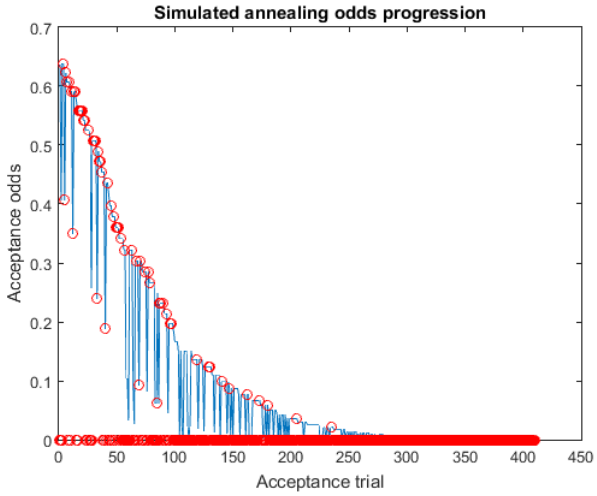

Figure 2.5: Typical probability progression for simulated annealing: the acceptance probabilities fluctuate according to the nearness in costs, but overall start high and end low. A red circle on the blue line at position x and height y indicates that the x-th trial has been accepted with probability y. A circle on the axis means it has not been accepted.

2.5.2 Modified problem

Like with greedy search, a simulated annealing implementation for the modified problem can easily be engineered from the implementation for the simple problem. We only need to update the interroute operations so that they cannot place customers on unwanted tours and change the way a random initial solution is found into the one presented in Section 2.4.2.

2.5.3 Relation to greedy search

An implementation for greedy search is very easily obtained from a simulated annealing implementation, basically by running simulated annealing with the altered probability function $p(N, B, T) = 0$; in other words, by never accepting more expensive solutions. Other changes are small: the termination criterion should be changed to one independent of temperature and the other irrelevant variables should be disregarded.

2.6 Neural networks

A vastly different approach for solving optimisation problems comes from developments in artificial intelligence. Neural network algorithms can be seen as systems that can 'learn' to distinguish between the quality of solutions, when represented as sets of values for declared variables. They do this either by observing a large set of 'good examples' and 'bad examples', in this case cheap feasible solutions and expensive/nonfeasible solutions, or by observing the quality of solutions under a given rule set. A neural networks algorithm for solving the mTSP, based on the latter, is presented by Wacholder et al. [6], whose work builds on Hopfields neural network algorithm for the TSP [2]. Wacholder represents the mTSP as a neural network with random starting voltages on the nodes and a numeric scheme with which this initial state should converge to a feasible solution with a low cost. His method is used and described here.

2.6.1 Simple problem

The mTSP can be represented with a neural network. A state of this network is represented by a $(n+m-1) \times (n+m-1)$ matrix V eventually containing only zeros and ones. An entry V_{ki} equals 1 if and only if node k is the i-th node to be visited. Note that the final $(m-1)$ rows of V represent returns to the depot. Effectively, $(m-1)$ virtual points are introduced with the same distance to the other points as the depot has. As such, the depot is represented by the first row and the final $(m-1)$ rows. The solution

$$\begin{matrix} 2 & 4 & 0 \\ 3 & 0 & 0 \end{matrix}$$

for $(n, m, p) = (4, 2, 4)$ corresponds to the neural network state

	$1st$	$2nd$	$3rd$	$4th$	$5th$
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	0	0	1
4	0	0	1	0	0
1	0	0	0	1	0

because the first 'visit' is to node 1 (the depot), the second visit is to node 2, the third visit is to node 4, the fourth visit is to 'node 5' (the depot, end of first tour) and the fifth visit is to node 3. The final return to node 1 is implied.

A neural network is to be viewed as a collection of electrically charged neurons with a set of rules on the electrical distribution. In a feasible end state of a neural network, the entries of the above matrix are approximately equal to 0 or 1. Each entry represents a neuron of the neural network, with an activation between 0 and 1, where an activated neuron has an activation value near 1. The activation of a neuron is based on the electrical charge on a neuron: if the neuron at location (k, i) is charged with voltage U_{ki}, its activation V_{ki} is determined with the sigmoid function $g : (-\infty, \infty) \to (0, 1)$:

$$0 < V_{ki} = g(U_{ki}) = \frac{1}{2}(1 + \tanh(U_{ki}/u_{00})) < 1$$

with u_{00} a constant dependent on the given parameters. The typical curve for the sigmoid function can be observed in Figure 2.6. If the neuron at location (5,2) is positively charged, so $U_{52} > 0$, this indicates that the second visit is probably made to node 5. The sigmoid function

will set the activation of this neuron closer to 1 than to 0, so $0.5 < V_{52} = g(U_{52}) < 1$. In short: positive voltages correspond to activations close to 1, negative voltages to activations close to 0. This relation is also illustrated by Figure 2.6.

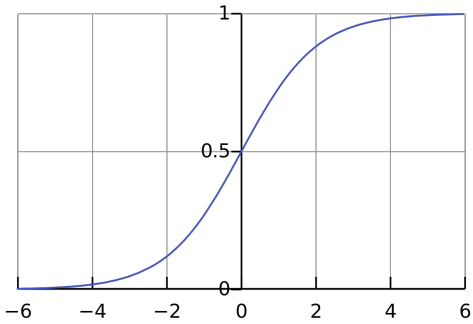

Figure 2.6: Typical curve for sigmoid functions. `Source: Wikipedia.org`

Wacholder establishes a set of five rules for a feasible end state of the mTSP neural network, as represented in the introduced notation:

$$\frac{1}{2}\sum_{k=1}^{n+m-1}\sum_{i=1}^{n+m-1}\sum_{j=1,j\neq i} V_{ki}V_{kj} = 0 \tag{2.17}$$

$$\frac{1}{2}\sum_{i=1}^{n+m-1}\sum_{k=1}^{n+m-1}\sum_{l=1,l\neq k} V_{ki}V_{li} = 0 \tag{2.18}$$

$$\frac{1}{2}\left(\left(\sum_{k=1}^{n+m-1}\sum_{i=1}^{n+m-1} V_{ki}\right) - (n+m-1)\right)^2 = 0 \tag{2.19}$$

$$\frac{1}{2}\sum_{k=n+1}^{n+m-1}\sum_{l=N+1,\ l\neq k}^{n+m-1}\sum_{i=1}^{n+m-1} V_{ki}\sum_{s=1}^{r}(V_{l,i+s} + V_{l,i-s}) = 0 \tag{2.20}$$

$$\frac{1}{2}\left(\left(\sum_{k=n+1}^{n+m-1}\sum_{i=1}^{n+m-1} V_{ki}\right) - (m-1)\right)^2 = 0 \tag{2.21}$$

Assuming that $V_{ki} \in \{0, 1\}$, condition (2.17) holds if and only if each row of neurons has no more than one neuron with activation 1, in other words, $\sum_{i=1}^{n+m-1} V_{ki} \leq 1$. Suppose for example that node k was the i-th and the j-th node visited: then $V_{ki} = V_{kj} = 1$ thus $V_{ki}V_{kj} = 1$ and the sum would become greater than 0. As long as node k is visited only the i-th time, the sum remains 0, because the term $V_{ki}V_{ki}$ is not counted. Similarly, condition (2.18) states that each column of neurons may have no more than one activated neuron. Condition (2.19) holds if and only if the total amount of activated neurons equals $(n + m - 1)$, the number of rows and columns. Combining these three conditions easily implies that a feasible solution must have exactly one activated neuron in each row and one in each column. Condition (2.20) concerns the virtual depot nodes: whereas previous methods were designed so that tours may not be longer than $(p-1)$, condition (2.20) is true if and only if the virtual depot is not revisited before the tour between them counts at least r other nodes. Essentially, it demands that tours may not be shorter than r. Unfortunately, this condition does not appear designed to keep the distance between a virtual depot node and the real depot node at least r (more on this in Sections 3.3 and 4.3). Note for condition (2.20) that the elements $V_{l,i+s}$ and $V_{l,i-s}$ are defined 'modulo $(n + m - 1)$':

in other words, $V_{l,1-1} = V_{l,(n+m-1)}$ and $V_{l,(n+m-1)+3} = V_{l,3}$, et cetera. Finally, condition (2.21) demands that the final $(m-1)$ rows, representing virtual depot nodes, contain a total of $(m-1)$ activated neurons.

The branch and bound algorithm solved an integer linear program to enforce the mTSP's rule set on a matrix of zeros and ones. Greedy search and simulated annealing took a random feasible solution and compared it with similar feasible solutions until some end criterion was reached. Realising that this algorithm models an 'electrical circuit' of neurons, one can actually speak of physical laws which determine the change in voltages over time. The neural networks algorithm relies on the laws of motion that Hopfield establishes for the voltages on neurons as they change in time. Wacholder modifies these into time derivatives for the voltages of neurons in an mTSP neural network, which he uses tot establish an iterative update rule. A neural network is initialised with noisy but unbiased voltages and, using this update rule, 'numerically integrated' to a feasible end state using Euler's Forward Method.

To this end, we define energy functions that correspond to conditions (2.17) through (2.21):

$$
\begin{aligned}
E_1 &= \frac{1}{2}\sum_{k=1}^{n+m-1}\sum_{i=1}^{n+m-1}\sum_{j=1,j\neq i} V_{ki}V_{kj} \\
E_2 &= \frac{1}{2}\sum_{i=1}^{n+m-1}\sum_{k=1}^{n+m-1}\sum_{l=1,l\neq k} V_{ki}V_{li} \\
E_3 &= \frac{1}{2}\left(\left(\sum_{k=1}^{n+m-1}\sum_{i=1}^{n+m-1} V_{ki}\right) - (n+m-1)\right)^2 \\
E_4 &= \frac{1}{2}\sum_{k=n+1}^{n+m-1}\sum_{l=N+1,\ l\neq k}^{n+m-1}\sum_{i=1}^{n+m-1} V_{ki}\sum_{s=1}^{r}(V_{l,i+s} + V_{l,i-s}) \\
E_5 &= \frac{1}{2}\left(\left(\sum_{k=n+1}^{n+m-1}\sum_{i=1}^{n+m-1} V_{ki}\right) - (m-1)\right)^2
\end{aligned}
$$

and the ideal goal is to minimise

$$E_p = \frac{1}{2}\sum_{k=2}^{n}\sum_{l=1,l\neq k}^{n+m-1}\sum_{i=1}^{n+m-1} d_{kl}V_{ki}(V_{l,i+1} + V_{l,i-1})$$

with d the modification of cost matrix C that includes the $(m-1)$ virtual depot points, under the condition that $E_1 = E_2 = E_3 = E_4 = E_5 = 0$ (because this implies that conditions (2.17) through (2.21) are met). E_p has been slightly modified from Wacholder's definition so that E_p equals the cost without departure from and return to depot which we would also find using the other methods. For the numeric process, Wacholder minimises the function

$$J = E_p + \lambda_1 E_1 + \lambda_2 E_2 + \lambda_3 E_3 + \lambda_4 E_4 + \lambda_5 E_5$$

where λ_i is a Lagrangian multiplier that determines how 'important' rule i is. For example, if rule five is 'not very important', it can be given a low value λ_5: if rule five is not violated, then $E_5 = 0$, but if rule five is violated greatly, then $E_5 >> 0$ but $\lambda_5 E_5$ will still be a relatively small term in J. Equally, 'more important' rules can be given higher values of λ_i. However, setting these values *too* high may greatly assist in finding feasible solutions (that's to say, solutions that

violate none of the five rules) but make the cost E_p of these solutions completely irrelevant. It is advised to choose initial values for λ_i carefully and to increment them over the course of the numeric process to push it further and further into a feasible solution.

In order to randomly initialise the network, first u_{00} is determined by setting all activations equal so that they sum up to $(n-m-1)$ and then finding the voltage that causes this activation.

$$u_{00} = g^{-1}\left(\frac{1}{n+m-1}\right) = u_0 \cdot \tanh^{-1}\left(2 \cdot \frac{1}{n+m-1}\right)$$

Here u_0, is a chosen parameter. Some symmetry breaking random noise is recommended to keep the network from remaining stuck in an equilibrium state. Each U_{ki} is initially set to $U_{ki} = u_{00} + \delta_{ki}$, with the noise term δ_{ki} chosen uniformly random in the interval $[-0.1u_0,\ 0.1u_0]$.

Then we determine how each voltage U_{ki} changes in the next iteration step. For this, we examine an update rule derived by Wacholder. It uses the function N, closely related to the partial derivative of J to U_{ki}:

$$\begin{aligned}
N(U_{ki}) &= -U_{ki}\Big(\sum_{l=1,\ l\neq k}^{n+m-1} d_{kl}(V_{l,i+1} + V_{l,i-1}) \\
&+ \lambda_1 \sum_{j=1,\ j\neq i}^{n+m-1} V_{kj} + \lambda_2 \sum_{l=1, l\neq k}^{n+m-1} V_{li} \\
&+ \lambda_3\Big(\sum_{l=1}^{n+m-1}\sum_{j=1}^{n+m-1} V_{lj} - (n+m-1)\Big) \\
&+ \lambda_4 \sum_{l=n+1,\ l\neq k}^{n+m-1}\sum_{s=1}^{r}(V_{l,i+s} + V_{l,i-s}) \\
&+ \lambda_5\Big(\sum_{l=n+1}^{n+m-1}\sum_{j=1}^{n+m-1} V_{lj} - (m-1)\Big)
\end{aligned}$$

for $k = 1, \ldots, n+m-1,\ i = 1, \ldots, n+m-1$. Using Euler's Forward Method, one can then update each U_{ki} and λ_q with

$$U_{ki}^{new} = U_{ki}^{old} + N(U_{ki}^{old}) \quad ; \quad \lambda_q^{new} = \lambda_q^{old} + E_q$$

for $k = 1, \ldots, n+m-1,\ i = 1, \ldots, n+m-1,\ q = 1, \ldots, 5$. As mentioned earlier, each λ_q will be non-decreasing over time, so that the solution may move less and less freely as time progresses. Termination can be called when $\sum_{q=1}^{5} E_q$ drops beneath a given threshold or when enough time has passed to see which state the process is converging to.

2.6.2 Modified problem

The surprisingly poor results that the neural networks method provides for the simple problem (see Section 3.3) make it seem unlucrative to expand the method for the more complicated modified problem.

Chapter 3

Results

In this chapter, six experiments are conducted to answer questions about the effectiveness of methods in certain contexts, under certain parameters and with or without additional constraints. These experiments should help in finding the best method for answering the company's three questions or should answer them directly. For each experiment, $m = 2$ is assumed.

The experiments have been conducted in MATLAB 2015a on a regular Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz 3.70GHz with 8,00 GB of RAM. The code has been made available in the appendix.

Most of the experiments use random subsets of the 69 relevant postal codes, the distances between which have been obtained using Google Maps. Though it would appear more congruent with experiments that seek to answer the modified and fitting problem to use home addresses instead of postal code centers, the postal codes are used regardless; both home addresses and the centers of postal code areas signify singular points on the map of the delivery region. Limited availabilities are randomly included: assuming $m = 2$, the probability that some customer i is available during both shifts is set to 0.6, whereas their availability during only the first shift or only the second shift are both set to 0.2. For the fitting problem, it is assumed by approval of the company that each customer requires ten minutes of service time. These ten minutes of service time, as well as the four hour limit per tour, are only represented in the final experiment, because they matter mainly in the fitting problem.

3.1 Comparing exact methods

The first focus of the research was to find an exact algorithm for efficiently solving the mTSP. A branch and bound implementation was built, but the discussion arose in Section 2.3.1 whether it would be most efficient to search its tree depth-first, width-first or using a 'switch over' method. These three methods are tested and compared here, along with a simple exhaustive search algorithm and MATLAB 2015a's ILP solver `intlinprog`, which uses a combination of Gomory cuts and other cuts before employing branch and bound.

Fifty 'random' networks of size $n = 7$ and $n = 9$ were generated from the data; for every network, branch and bound was solved once using depth-first search, once using width-first search and once using the 'switch over' method. For each value n, a tight value of p and a slightly more lenient value of p were tested. To further place the results in perspective, the networks were also solved with exhaustive search and `intlinprog`. The computation times (in seconds) were recorded and the averages and standard deviations can be observed in Table 3.1.

These tests seem to indicate that width-first search is the most effective of the three search methods. Note also that `intlinprog` is many times faster than the other methods. It will be the exact algorithm used in the following experiments. Tight values of p appear hardest to optimise with and thus the most interesting for most of the following experiments.

Though exhaustive search may seem viable from these test results, a test was done with only three random networks of size $n = 10$. Solving these three problems with exhaustive search took an average of 2926.8 seconds, with a standard deviation of only 0.4099 seconds. The averages of the other methods did not exceed 74 seconds. Though no substantial claims can be made on only three simulations, this result gives an indication of how quickly the computation time for exhaustive search tends to escalate.

	μ	σ
$n = 7$, $p = 4$		
Exhaustive search	0.4215	0.0188
Depth-first branch and bound	11.6480	6.0296
Width-first branch and bound	6.7133	5.8601
Switch over branch and bound	7.8119	5.9197
`intlinprog`	0.0605	0.0396
$n = 7$, $p = 5$		
Exhaustive search	0.4399	0.0095
Depth-first branch and bound	6.8313	6.2623
Width-first branch and bound	4.1652	5.1338
Switch over branch and bound	5.2426	5.7314
`intlinprog`	0.0499	0.0229
$n = 9$, $p = 5$		
Exhaustive search	55.9463	0.6043
Depth-first branch and bound	91.4244	66.1902
Width-first branch and bound	55.3648	63.0052
Switch over branch and bound	60.4036	61.5637
`intlinprog`	0.1282	0.0691
$n = 9$, $p = 7$		
Exhaustive search	56.8798	0.1095
Depth-first branch and bound	39.0187	37.4208
Width-first branch and bound	20.5594	25.8108
Switch over branch and bound	25.5386	31.3314
`intlinprog`	0.0680	0.0648

Table 3.1: Averaged testing results between exact methods.

3.2 Comparing additional constraints

In Section 2.3.3, possible additional constraints for the branch and bound implementation in the simple and modified problem were proposed. Constraint (2.14) gave an upper bound on the sum of potentials, where constraint (2.15) was a symmetry restriction that disallowed that $x_{ij} + x_{ji} \geq 1$. For the modified problem, single tour allocation constraint (2.16) was proposed. In this experiment, the effectiveness of these additional constraints was tested with `intlinprog`. Note that `intlinprog` was chosen for it being shown in Section 3.1 to be far quicker than regular

branch and bound. The results in that section also seemed to indicate the highest computation time for tight values of p: therefore, tight values of p are chosen for this experiment.

Fifty 'random' networks of size $n = 5$, $n = 10$ and $n = 15$ were generated from the data; for every network, `intlinprog` for the simple problem was used once with the inclusion of only (2.14), once with only (2.15), once with both and once with neither. After this, `intlinprog` for the modified problem was used once without the inclusion of (2.16) and once with. The computation times (in seconds) were recorded and the averages and standard deviations can be observed in Table 3.2. For the simple problem, these tests seem to indicate that the inclusion of only the symmetry restriction constraint (2.15) delivers the best results, better than also including the bound on the sum of potentials (2.14). The algorithm for the modified problem appears to perform slightly better without the single tour allocation constraint (2.16).

	μ	σ
$n = 5$, $p = 3$		
Simple without additional constraints	0.0324	0.0196
Simple symmetry restriction	0.0265	0.0168
Simple bound on sum potentials	0.0324	0.0189
Simple with both additional constraints	0.0253	0.0172
Modified without allocation constraint	0.0378	0.0167
Modified with allocation constraint	0.0390	0.0182
$n = 10$, $p = 6$		
Simple without additional constraints	0.2683	0.3350
Simple symmetry restriction	0.1513	0.1294
Simple bound on sum potentials	0.2727	0.3458
Simple with both additional constraints	0.1610	0.1508
Modified without allocation constraint	0.4714	0.6560
Modified with allocation constraint	0.4927	0.7640
$n = 15$, $p = 8$		
Simple without additional constraints	31.9116	59.2456
Simple symmetry restriction	28.3797	62.6817
Simple bound on sum potentials	35.8381	65.4871
Simple with both additional constraints	28.8134	64.3310
Modified without allocation constraint	148.8718	219.6829
Modified with allocation constraint	150.9388	214.0474

Table 3.2: Averaged testing results with or without additional constraints.

3.3 Comparing methods for the simple problem

In order to assess the accuracy of the heuristic methods and the gain in computation speed, it is necessary to observe a problem suitable for both exact methods and heuristic ones. To this end, it was decided to examine random subgraphs of size $n = 10$, $n = 20$ and $n = 30$. The values of p were chosen minimal, to show that even in their most restricted form, methods will generally beat neural networks in accuracy. Neural networks were dropped after $n = 10$, as no feasible solutions have yet been found for $n = 20$. For $n = 30$, a limited version of `intlinprog` has been used that ceases when branch and bound has explored 10^6 leaves. For results with an asterisk, it must be noted that the solutions obtained from this version of `intlinprog` may have

a relative optimality gap of 0% to 4%. The regular difficulty in solving problems of this size is left unexplained, especially as `intlinprog` works fast enough in the case $n = 69$ (as shown in Section 3.4). In this experiment, the averages of both the best costs and of computation times for all discussed simple methods are presented.

For each value of n, greedy search, simulated annealing and neural networks were tested with 'quick parameters' (parameters which ensure quick computation at the possible cost of accuracy), with 'medium parameters' and more rigorous 'slow parameters'.

- In the case of greedy search, the value of the patience parameter τ was incremented between tests. In quick tests, $\tau = 20$; in medium tests, $\tau = 50$; in slow tests, $\tau = 100$.

- In the case of simulated annealing, the amount of steps per temperature were incremented (20, 50, 100), as well as the starting temperature T_0 (15, 20, 25, with $T_0 = 15$ corresponding to initial probabilities near 0.7). The end temperatures were made smaller (0.01, 0.001, 0.0001). Note that $\alpha = 0.95$ was unchanged, for this value appears conventional.

- In the case of neural networks, the starting values for λ_1 through λ_5 were made smaller between tests. High values of these parameters enforced feasible but costly solutions. Low values came with the risk of having to run several times before converging to a feasible solution. Initially, $(\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5) = (200, 200, 500, 10, 10)$, decreasing to $(150, 150, 375, 10, 10)$ and finally $(100, 100, 250, 10, 10)$, the latter seeming to be the approximate minimum for which solutions remained feasible commonly.

Fifty 'random' networks of size $n = 10, 20, 30$ were generated from the data and solved. The best costs and computation times (in seconds) were recorded and the distances to the global optima provided by `intlinprog` were computed; the averages and standard deviations can be observed in Table 3.3. We conclude that for problems of size $n \leq 10$, `intlinprog` is the most efficient method. For problems of size $n = 20$ and $n = 30$, simulated annealing appears most efficient under 'any' choice of parameters, as it approximates the exact solution well within only a fraction of the time. We also observe that greedy search terminates more quickly under the given parameters than simulated annealing, but leaves a far larger optimality gap. As simulated annealing finds better results than greedy search, albeit at the cost of a few more seconds computation time, greedy search is not investigated further for the simple problem.

3.4 Solving the simple problem

Solving the simple problem is now merely a question of using `intlinprog` to determine the solution to the mTSP on the complete graph of 69 postal codes. This, however, also offers the opportunity of comparing the fastest available exact method with the seemingly strongest available heuristic method for a larger value of n. In this experiment, the full mTSP with $n = 69$ is solved once with `intlinprog` and fifty times with simulated annealing (150 steps per temperature, $(T_0, T_{end}) = (30, 0.00001)$). The average computing times and best found costs are stored and their means and standard deviations can be observed in Table 3.4. The solution itself is presented in Figure 4.1. Additionally, the cost found on average is approximately 109.5% of the optimum, arguably more than favourable when the computation time decreases to only 2%. Though this statement means little to the simple problem, as the simple problem concerns only determining an optimal route once, it should be noted that these results speak in favour of simulated annealing for larger problems.

	Cost optimum μ	Cost optimum σ	Time μ	Time σ
$n = 10$, 'quick':				
`intlinprog`	100%	0	0.1934	0.1924
Greedy search	110.6 %	12.9%	0.0406	0.0197
Simulated annealing	100.3%	1.0%	0.9906	0.0.362
Neural networks	198.0%	106.5%	7.8047	5.1389
$n = 10$, 'medium':				
`intlinprog`	100%	0	0.1934	0.1924
Greedy search	102.9%	4.9%	0.4477	0.4246
Simulated annealing	100.3%	0.9%	3.1247	0.0682
Neural networks	231.45%	33.7%	3.4476	0.9813
$n = 10$, 'slow':				
`intlinprog`	100%	0	0.2059	0.1705
Greedy search	101.1%	2.3%	78.3250	114.3078
Simulated annealing	100%	0	7.4740	0.0718
Neural networks	237.3%	47.0%	3.4523	0.9905
$n = 20$, 'quick':				
`intlinprog`	100%	0	792.2199	1603.0
Greedy search	144.5%	28.7%	0.0515	0.0255
Simulated annealing	104.3%	4.0%	1.0000	0.0496
$n = 20$, 'medium':				
`intlinprog`	100%	0	792.2199	1603.0
Greedy search	114.05%	8.5%	0.1716	0.0907
Simulated annealing	101.5%	2.1%	3.1341	0.1126
$n = 20$, 'slow':				
`intlinprog`	100%	0	792.2199	1603.0
Greedy search	104.7%	4.8%	1.0343	0.5464
Simulated annealing	101.5%	2.3%	7.6862	0.1066
$n = 30$, 'quick':				
`intlinprog`	100% – 104%*	0*	317.0237*	362.0043*
Greedy search	157.2% – 163.5%*	18.9% – 19.7%*	0.015	0.0295
Simulated annealing	109.9% – 114.3%*	5.3% – 5.5%*	1.0577	0.0319
$n = 30$, 'medium':				
`intlinprog`	100% – 104%*	0*	317.0237*	362.0043*
Greedy search	134.8% – 140.2% *	13.8% – 14.4%*	0.1794	0.0843
Simulated annealing	103.6% – 107.7%*	3.1% – 3.2%*	3.2776	0.0604
$n = 30$, 'slow':				
`intlinprog`	100% – 104%*	0*	317.0237*	362.0043*
Greedy search	112.3% – 116.8%*	4.1% – 4.3%*	0.5398	0.2644
Simulated annealing	103.3% – 107.4%*	2.5% – 2.6%*	7.9826	0.0826

Table 3.3: Averaged testing results between all methods.

3.5 Comparing methods for the modified problem

Each of the above experiments concerned the simple problem. It is of much interest to compare how the implementations of branch and bound, `intlinprog`, greedy search and simulated annealing for the modified problem perform.

	Cost optimum μ	Cost optimum σ	Time μ	Time σ
`intlinprog`	100%	-	746.8548	-
Simulated annealing	109.5%	12.0277	15.6656	0.1008

Table 3.4: Averaged testing results for $n = 69$.

Fifty 'random' networks of size $n = 10$, $n = 15$ and $n = 20$ were generated from the data; each node of each network was randomly set to be available during both shifts with probability 0.6 and available only on the first shift or second shift with probability 0.2. Again, p was chosen minimal. Greedy search used $\tau = 50$, simulated annealing used fifty steps per temperature and $(T_0, T_{end}) = (15, 0.001)$. The limited version of `intlinprog`, which may leave a relative optimality gap of up to 4%, was used. The results on which this may have influence, are again marked with an asterisk. The best costs and computation times (in seconds) were recorded and the distances to the solutions provided by the limited intlinprog were computed; the averages and standard deviations can be observed in Table 3.5. They seem to show that simulated annealing is the most appropriate algorithm for problems of size $n = 15$ and probably larger.

	Cost optimum μ	Cost optimum σ	Time μ	Time σ
$n = 10$, $p = 6$:				
`intlinprog`	100% – 104%*	0	0.5429*	0.6236*
Greedy search	108.7% – 113.1%*	11.1% – 11.5%*	0.1279	0.0452
Simulated annealing	100% – 104%*	0*	9.8156	1.4433
$n = 15$, $p = 8$:				
`intlinprog`	100% – 104%*	0	130.4933*	130.2915*
Greedy search	117.2% – 121.9%*	7.3% – 7.6%*	0.4274	0.2080
Simulated annealing	102.4% – 106.5%*	3.8% – 4.0%*	23.1209	4.8132
$n = 20$, $p = 11$:				
`intlinprog`	100% – 104%*	0	212.0740*	202.2010*
Greedy search	121.8% – 126.7%*	13.3% – 13.8%*	0.9329	0.3212
Simulated annealing	102.6% – 106.7*	2.9% – 3.0%*	33.9365	9.6129

Table 3.5: Averaged testing results in the modified problem.

3.6 Solving the fitting problem

The fitting problem concerns finding the largest value n for which generally all shifts remain under four hours. In the fitting problem, time spent servicing the customers is included: this is set to be ten minutes per customer. We choose to determine the largest viable value of n by picking a value of n and running a few simulations with that value to determine if this n seems viable. If so, we examine if larger n also seem viable. The more exact algorithm is as follows:

1. Given a value of n, set $p = \lceil n/2 \rceil + 1$. We want to set p only slightly higher than its minimum of $\lfloor n/2 \rfloor + 1$ to keep the length of the tours very close.

2. Determine a random subset of size n of the 69 postal codes, along with random availabilities like in the previous experiment.

3. Solve the modified problem with `intlinprog` and simulated annealing. Add to the costs of each tour ten minutes for each node on that tour. Determine the new maximum tour

cost and whether or not it remains under four hours. Save the maximum tour time with the client service times for both `intlinprog` and simulated annealing, as well as the corresponding tour time without the added service times.

4. Repeat steps 2 and 3 five times. If all tours remain under four hours for at least four of the five random networks using `intlinprog`, increment n by 1 and return to step 1. Otherwise, success rates at this value are approximately lower than 80%: terminate the algorithm and base an advise on the returned maximum tour values with and without client service time.

It must be noted that, with the used data set, nodes cannot be very close to another, because they are set at the centers of different postal codes. Combined with the fact that ten minutes service is time is probably more than what will normally be necessary, this algorithm works from a worst case scenario point of view.

In Table 3.6, we see the maximum tour lengths found with `intlinprog` and simulated annealing for each simulation. The fact that simulated annealing finds lower values sometimes, is because `intlinprog` finds the solution with the lowest *total* costs of tours, not the one with the lowest maximum cost per se. These results seem to indicate that, in the worst case scenario, $n = 23$ is the last 'safe' value for which solutions generally exist, as well as the final value for which simulated annealing will find appropriate solutions on average. Note again that $n = 23$ corresponds to 22 customers, as the first node represents the depot address. The corresponding durations without service times have been included for if the reader wishes to estimate results with lower average service times.

											μ	
$n = 20$, service time:	212	203	219	212	206	192	204	204	219	218	212	206
Without service time:	112	103	119	112	106	92	114	114	119	118	114	108
$n = 21$, service time:	224	225	230	231	212	264	223	223	220	224	222	233
Without service time:	114	115	120	121	102	154	113	113	110	114	112	123
$n = 22$, service time:	235	240	218	241	189	211	221	224	222	214	217	226
Without service time:	125	130	108	131	89	101	121	124	112	104	111	118
$n = 23$, service time:	234	227	231	267	216	239	241	242	224	216	229	238
Without service time:	114	117	121	147	106	119	121	122	104	106	113	122
$n = 24$, service time:	224	227	226	242	242	242	252	260	253	256	239	245
Without service time:	104	107	106	122	122	122	132	140	133	148	119	127

Table 3.6: Rounded maximum tour times in simulations of modified problem. Each left element in a cell is the result for `intlinprog`, each right element the result for simulated annealing. Times are in minutes: maximum tour times should stay under four hours, thus 240 minutes.

Chapter 4

Discussion

In Chapter 1, we pose three problems. Solution methods are proposed in Chapter 2 and experiments with these methods are reported in Chapter 3. In this final chapter, the results are discussed and the questions answered.

4.1 Summary of experimental results

In Section 3.1 it was shown that of the available exact implementations for the simple problem, `intlinprog` is by far the quickest. The results seem to suggest that branch and bound works faster when implemented as handling its active tree width-first than as handling it depth-first.

In Section 3.2, it was shown that of the three additional constraints introduced for branch and bound, only the symmetry restriction constraint (2.15) had a clear positive effect.

In Section 3.3, it was shown that greedy search and especially simulated annealing can accurately approximate exact solutions in relatively little time for simple problems size $n = 20$ under the right choice of parameters. This was further confirmed in Section 3.4 for $n = 69$. Neural networks did not perform well.

In Section 3.4, the simple problem was solved for the entire delivery area. The optimal routes across the 69 relevant postal codes can be viewed in Figure 4.1. Note that the departure from and return to the depot are not taken into the cost function. The departures have been left in the figure to indicate in what direction the tours can best be followed. More specifically, the suggested tours across the postal codes are as follows:

- (8071), 3846, 3841, 3844, 3845, 3853, 3851, 3852, 3847, 3843, 3842, 3848, 3849, 8077, 8072, 8084, 8085, 8081, 8082, 8097, 8079, 8278, 8096, 8095, 8091, 8094, 8276, 8266, 8261, 8262, 8264, 8263, 8261, 8251, 8253, 8254, 8252, 8256.
- (8071), 8267, 8277, 8271, 8274, 8275, 8044, 8045, 8043, 8031, 8032, 8033, 8034, 8025, 8024, 8026, 8013, 8015, 8014, 8016, 8017, 8019, 8042, 8041, 8011, 8012, 8023, 8021, 8022, 8035, 8028.

Observing Figure 4.1 may leave one wondering whether the routes in Dronten and Kampen are truly most efficient, as they do not seem intuitive. We believe that the distances between postal codes that Google provides may be slightly different from what they are in practice. This, because Google may pinpoint the direct center of not only the densely populated area of a postal code, but of this area combined with stretches of countryside that may also be part of the postal

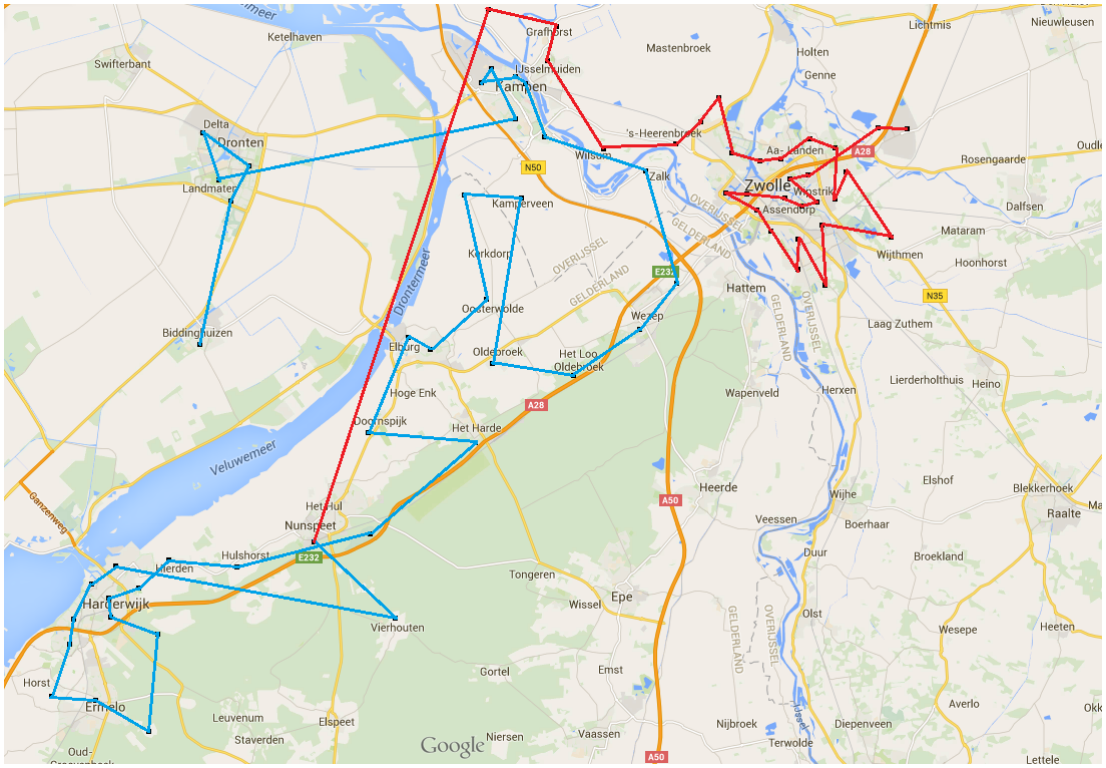

Figure 4.1: The solution to the mTSP with $m = 2$ over the $n = 69$ relevant postal codes. `Source: Google Maps`

code. We advise the company, with their expertise of the area, to slightly deviate from these exact results in such small subsections when this seems more appropriate.

Section 3.5 seems to suggest that simulated annealing will be an appropriate method for building an on the fly solver for the modified problem.

In Section 3.6, a worst case fit was computed and it was investigated how large n may become before the approximate probability of finding a solution with subtours under four hours drops under 80%. In the experiment, less than 80% of the simulations fitted for $n = 24$, coinciding with an average maximum tour time of 239 minutes, whereas $n = 23$ appeared to still be viable.

4.2 Main questions

Let us finally review the questions asked and their answers.

1. *Without prior knowledge of actual customers and their addresses, what is the optimal way to cover the delivery area with a given amount of delivery routes, with optimal meaning fastest?*

   According to the results from Section 3.4, the optimal two tours across the 69 postal codes that form the delivery area are represented in Figure 4.1. Roughly speaking, our advise is to travel in one shift over IJsselmuiden, Wilsum and all of Zwolle and in the other over Vierhouten, Harderwijk, Ermelo, Elburg, Het Loo Oldebroek, Zalk, Kampen

and Dronten, ending in Biddinghuizen before returning to depot. It could also be said that the postal codes can be split over two shifts by covering one side of the river IJssel on each shift.

2. *Given some set of home addresses of customers who may or may not be available at certain days, what are the optimal delivery routes? That's to say, what is the way to divide customers over the delivery shifts and determine delivery routes for each shift with the lowest total amount of time spent traveling between addresses?*

   For building an on the fly solver for this problem, we advise a method based on simulated annealing.

3. *How many deliveries can the company expect to fit within a four hour shift? More specifically, how many customers can the company serve if they have two delivery shifts per week?*

   Based on the results from Section 3.6, our worst case estimate is 22 customers per week, thus 11 customers per shift. This may increase somewhat when assuming shorter service time and less spread out customer coordinates.

4.3 Future research

Though we believe the three questions asked by the company have been answered, other issues may yet be addressed. It has been noted that reviewing the three problems with the Min-Max cost function instead of our modification of the classical cost function may lead to interesting insights. Furthermore, a plethora of heuristic methods for solving the mTSP have been developed and may be compared with the ones reviewed here.

Additionally, the neural networks algorithm may hold untapped potential, both in finding more appropriate parameters to run it with and in modifying the algorithm itself to include more constraints. One explanation for the poor performance of neural networks may lie in the energy functions. The first of the five energy functions flag if the sum of some row becomes greater than 1, but not if such a sum becomes lower than 1. It may seem unwarranted to include an energy function E_6 that flags if the sum of some row is smaller than 1, as this is generally true in the initial state of the neural network, but one could consider including it with a modified update rule for its corresponding Lagrangian multiplier λ_6, namely that the multiplier has a very low value initially but grows to match λ_1 as time progresses. A similar rule E_7 can of course be suggested for the columns. It seems warranted to explore this option, seeing how often the solutions seem to converge to a state with a zero row or column, despite the high values of E_3. Also, replacing Euler's Forward Method by something more sophisticated like Runge-Kutta's fourth method may greatly increase computation speed. Finally, it appears that the current system does not enforce well that $V_{11} = 1$, nor that there must be at least r nodes visited after node 1 before it may return to a depot node. Exploring these three options to increase the performance of the neural network algorithm are suggested as items for future research.

In the more practical sense, other relevant questions for a starting company may be asked. As the eventual delivery area is not (yet) rigidly defined, it may be promising to accept customers in other settlements further from the depot. One could examine the question of how close a potential customer must be to the depot to be worth servicing. Of course, questions of a more statistical nature also remain: for example, how much growth of the customer base is

expected in each area and how long customers will remain customers, based on the age of their child.

Bibliography

[1] Tolga Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209–219, 2006.

[2] John J Hopfield and David W Tank. neural computation of decisions in optimization problems. *Biological cybernetics*, 52(3):141–152, 1985.

[3] András Király and János Abonyi. Optimization of multiple traveling salesmen problem by a novel representation based genetic algorithm. In *Intelligent Computational Optimization in Engineering*, pages 241–269. Springer, 2011.

[4] Clair E Miller, Albert W Tucker, and Richard A Zemlin. Integer programming formulation of traveling salesman problems. *Journal of the ACM (JACM)*, 7(4):326–329, 1960.

[5] Christos H Papadimitriou. The euclidean travelling salesman problem is np-complete. *Theoretical Computer Science*, 4(3):237–244, 1977.

[6] E Wacholder, J Han, and RC Mann. A neural network algorithm for the multiple traveling salesmen problem. *Biological Cybernetics*, 61(1):11–19, 1989.

Appendix A

List of relevant postal codes

These are the 69 four-digit postal codes that combine to form the intended delivery area. They form the basis of solving the simple problem. Each of these settlements are in the Netherlands.

8256, Biddinghuizen	8251, Dronten	8252, Dronten
8253, Dronten	8254, Dronten	8081, Elburg
8082, Elburg	8084, 't Harde	8085, Doornspijk
3851, Ermelo	3852, Ermelo	3853, Ermelo
8071, Nunspeet (depot)	8072, Nunspeet	8076, Nunspeet
8077, Hulshorst	8095, 't Loo Oldebroek	8094, Hattemerbroek
8079, Noordeinde (Gelderland)	8091, Oldebroek	8096, Oldebroek
8097, Oosterwolde (Gelderland)	3841, Harderwijk	3842, Harderwijk
3843, Harderwijk	3844, Harderwijk	3845, Harderwijk
3846, Harderwijk	3847, Harderwijk	3848, Harderwijk
3849, Harderwijk	8011, Zwolle	8012, Zwolle
8013, Zwolle	8014, Zwolle	8015, Zwolle
8016, Zwolle	8017, Zwolle	8019, Zwolle
8021, Zwolle	8022, Zwolle	8023, Zwolle
8024, Zwolle	8025, Zwolle	8026, Zwolle
8028, Zwolle	8031, Zwolle	8032, Zwolle
8033, Zwolle	8034, Zwolle	8035, Zwolle
8041, Zwolle	8042, Zwolle	8043, Zwolle
8044, Zwolle	8045, Zwolle	8261, Kampen
8262, Kampen	8263, Kampen	8264, Kampen
8265, Kampen	8266, Kampen	8267, Kampen
8278, Kampen	8271, IJsselmuiden	8275, IJsselmuiden
8277, IJsselmuiden	8276, Zalk	8274, Wilsum

Appendix B

Alternative method for finding a weighted graph

This section discusses how a weighted graph for the modified problem can be established if direct travel times between addresses are not available from Google Maps or other sources.

One cannot speak of solving an mTSP without describing the graph in which it is to be solved. When envisioning the delivery area as a graph G, it may be tempting to view the streets as arcs and their crosspoints as nodes. One could then assign the time it takes to get from the beginning of the street to its end as a weight of the arc. The nodes may even be given a weight: busy crosspoints may take longer to cross than simple suburban T junctions. It is advised to not take those crosspoint times in consideration yet, however, as the average time to cross over i to j may differ from the average time to cross from i to k and these differences are accounted for in the next step. Figure B.1 shows part of such a graph. Symmetry is assumed: it is assumed that it takes just as long to get from the beginning of some street to its end as vice versa.

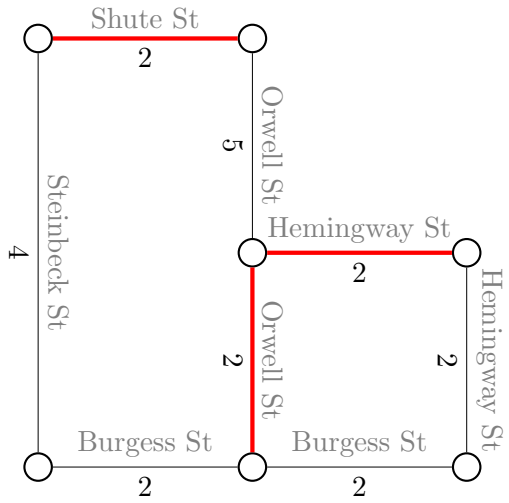

Figure B.1: Part of a fictional street network. Deliveries must be made to the bold arcs.

However, the deliveries must be made at streets, not crosspoints. Therefore, it is necessary to model the streets as nodes, not arcs. Fortunately, this is easily solved by observing the dual of G, G^*: in other words, by making a new graph from G, assigning each arc of G a node in

G^* and connecting nodes in G^* if they share a crosspoint in G. The arcs of this dual can be weighted with the time to get from the middle of one street to the other: in other words, by adding half the time of crossing one street, half the time of crossing the other street and the time of traversing the crosspoint between them. Refer to Figure B.2 for an example of this process.

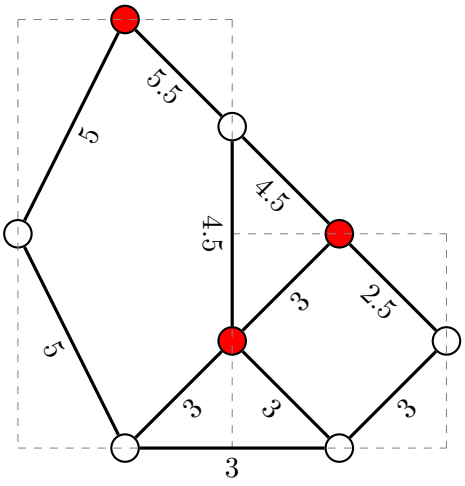

Figure B.2: Weighted dual of the previous network. Deliveries must be made to the filled nodes.

Though this process yields a graph with weighted arcs and nodes for streets, this graph is still not the one to solve the mTSP in. In most of the cases, the company will not have to make a delivery to *every* street. A new graph must be made with the n streets that *are* to be visited (including the depot). As each location can theoretically be visited from all other locations, one could observe the complete graph K_n (the graph containing exactly n nodes and having an arc between each pair of nodes). It may seem counterintuitive to draw a direct connection between street i and k if you cannot do so without passing through street j, but leaving this arc out of the graph may disrupt the mTSP solution process by demanding that j is visited twice. It is thus advisable to use the entire graph K_n. The arc between node i and j of K_n can be weighted with the shortest time from the center of street i to the center of street j.

The needed distances can be easily obtained by a computer from the aforementioned G^* using the Floyd-Warshall algorithm. When supplied with the weights of the present arcs, this algorithm efficiently finds the shortest distance between all pairs of nodes. A sketch of how this well known algorithm works is considered outside of the focus of this report. The interested reader is advised to review the 'Handbook of Graph Theory' (Gross et al, 2003) or other documentation.

What is important to know, is that Floyd-Warshall can be used to find the shortest distance between all pairs of nodes and that this finally leads to a weighted graph K_n over which mTSP can be solved. Such a graph can be viewed in Figure B.3. Note that it is advised to skip the very first step, of manually making a graph with unweighted crosspoints as nodes and streets as arcs: the next step still requires manual entry that hardly depends on the manual entry of the first step, so it is advised to go straight ahead to manually entering the time it takes to get from one street to its neighbouring street. To sum up the process of obtaining a graph in which to solve the mTSP:

1. Examining the delivery area, (mentally) make a graph with a node for each street and an arc between each node if the streets are directly connected. Set up an $s \times s$ matrix C, with

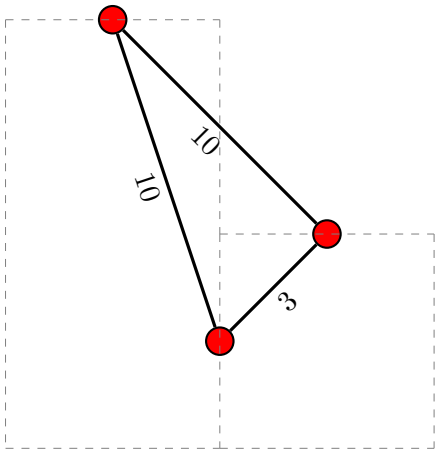

Figure B.3: With the three delivery addresses known (including depot), examine a weighted K_3.

s the total number of observed streets. If street i and j are directly connected, set C_{ij} to be the average time it takes to get from the middle of street i to the middle of street j, keeping the waiting time at their crosspoint in mind.

2. Use Floyd-Warshall to find the other entries of C. Then add very high weights to the diagonal of C.

3. When the delivery addresses are known, and thus the n relevant streets, (mentally) observe the complete graph K_n between them. Get its corresponding cost matrix by trimming down C. This weighted graph is the one to solve the mTSP in.

Note that the first two steps only need to be performed whenever the representation of the delivery area is changed.

Appendix C

Code for exhaustive search and general UI

```
function [ bestroute, bestcost ] = exhaustive( Costs, m, p )

progress = 0;
progressStep = 2500;

n = size(Costs, 1);
route = [ones(1, m) 2:n 1];
bestroute = route;
bestcost = Inf;
NRroutes = factorial(n+m-2);

%disp('Number of routes to check:');
%disp(NRroutes);
%input('Continue? (Press ENTER to continue or Ctrl+C to abort.)');

for i = 1:NRroutes-1

    progress = progress + 1;
    if (progress == progressStep)
        disp(i/(NRroutes-1));
        progress = 0;
    end

    route = next(route);
    cost = Inf;
    if (validsol(route, p))
        mat = route2matrix(route);
        cost = costrouteinternal(Costs, mat);
    end
    if (cost == bestcost)
        bestroute = [bestroute ; route];
    end
    if (cost < bestcost)
        bestroute = route;
        bestcost = cost;
    end
end

function mat = route2matrix( route )
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here

n = max(route);
mat = zeros(n);
for i = 1:length(route)-1
    mat(route(i), route(i+1)) = 1;
end

end
```

```
function c = costrouteinternal( Costs, Broute )
%costroute Compute the cost of a binary route.
%   Detailed explanation goes here

c = 0;
[r, k] = size(Costs);
[s, l] = size(Broute);
for i = 2:r
    for j = 2:k
        c = c + Costs(i,j)*Broute(i,j);
    end
end

end
```

Appendix D

Code for format changing

```
function [ BinaryRoute ] = des2bin( DescriptiveRoute )
%des2bin Convert a descriptive route to its binary route.
%   Example descriptive route:
%   [2 3 4;
%    5 6 0]
%   (in other notation, 1 2 3 4 1 5 6 1)
%
%   Its binary route:
%   [0 1 0 0 1 0;
%    0 0 1 0 0 0;
%    0 0 0 1 0 0;
%    1 0 0 0 0 0;
%    0 0 0 0 0 1;
%    1 0 0 0 0 0]

% Get the m salesmen and n addresses:
D = DescriptiveRoute;
[m, d] = size(D);
n = max(max(D));

% Initialize BinaryRoute (as B):
B = zeros(n);

% For each salesman, fill in B:
for i = 1:m
    for j = 1:d-1
        row = D(i,j);
        col = D(i,j+1);
        if (row ~= 0 && col ~= 0)
            B(row, col) = 1;
        end
    end
end

% Departing from and returning to depot is forgotten. Include:
for i = 1:m
    depart = D(i,1);
    B(1, depart) = 1;

    currow = D(i,:);
    retur = currow(end);
    while (retur == 0);
        currow(end) = [];
        retur = currow(end);
    end
    B(retur, 1) = 1;
end

BinaryRoute = B;

end
```

```
function [ DescriptiveRoute ] = bin2des( BinaryRoute, p )
%bin2des Convert a binary route to a corresponding descriptive route.
%   Example binary route:
%   [0 1 0 0 1 0;
%    0 0 1 0 0 0;
%    0 0 0 1 0 0;
%    1 0 0 0 0 0;
%    0 0 0 0 0 1;
%    1 0 0 0 0 0]
%
%   One of its descriptive routes:
%   [2 3 4;
%    5 6 0]
%   (in other notation, 1 2 3 4 1 5 6 1)

% Get the m salesmen and n addresses:
B = BinaryRoute;
m = sum(B(1,:));
% n = size(B, 1);

% Initialize DescriptiveRoute:
D = zeros(m, p-1);

% For each salesman, fill in D.
for i = 1:m

    % Find the first departure in the top row of B:
    curAddress = 1;
    col = 1;
    while (curAddress == 1)
        col = col + 1;
        if (B(1, col) == 1)
            D(i,1) = col;
            B(1, col) = 0;
            curAddress = col;
        end
    end

    % Then keep finding the next address until return to depot:
    Dcol = 1;
    while (curAddress ~= 1)
        Dcol = Dcol + 1;
        [~, col] = ismember(1, B(curAddress, :));
        if (col ~= 1)
            D(i, Dcol) = col;
        end
        B(curAddress, col) = 0;
        curAddress = col;
    end

end

% % All tours are entered but D is unnecessarily broad. Trim the zero cols:
% done = 0;
% while(~done)
%     if (D(:, end) == zeros(m, 1))
%         D(:, end) = [];
%     else
%         done = 1;
%     end
% end

DescriptiveRoute = D;

end


function [ Droute ] = ilp2desC( m, n, p, ilpsol )
%UNTITLED5 Convert the solution of an ilp to a descriptive route.
%   Detailed explanation goes here
```

```
sol = round(ilpsol);
% Read the potentials:
u = sol(m*n^2 + 1 : end);

Droute = zeros(m, p-1);
for k = 1:m
    solpiece = sol((k-1)*n^2 + 1 : k*n^2);
    for i = 1:n
        for j = 2:n
            index = (i-1)*n + j;
            % If j is reached on day k, add it to row k in Droute. For the
            % col, use the potential of node j.
            if (solpiece(index) == 1)
                col = u(j-1);
                Droute(k, col) = j;
            end
        end
    end
end

end

function [ Clean ] = shiftzerosright( Matrix )
%shiftzerosright For each row, moves all zeros to the right.
%   Detailed explanation goes here

M = Matrix;
[rows, cols] = size(M);

Clean = zeros(size(M));
for i = 1:rows
    index = 1;
    for j = 1:cols
        if (M(i,j) ~= 0)
            Clean(i, index) = M(i,j);
            index = index + 1;
        end
    end
end


end
```

Appendix E

Code for branch and bound

```
function [ bestroute, bestcost ] = BnBmode( Costs, m, p, Mode )
%BnB Solve mTSP with branch-and-bound. Salesman may visit p cities.
%   Detailed explanation goes here

debugleaf = 1;

% Get n from the Costs matrix:
n = size(Costs, 1);

% Get the linear program (A and b):
[A, b] = progger5(m,n,p);
if (strcmp(Mode, 'old'))
    [A, b] = progger4(m,n,p);
end

% Get costs vector c from Costs (adding zeros for u_2, u_3, ..., u_n):
c = reshape(Costs', n^2, 1);
c = [c ; zeros(n-1, 1)];

% Get clone of c with only the cost that aren't to or from depot:
cinternal = c;
cinternal(2:n) = 0;
for i = 2:n
    cinternal( (i-1)*n+1 ) = 0;
end

% Get upper and lower bounds for variables x_ij and u_i:
lb = zeros(n^2 + n-1, 1);
lb(n^2 + 1 : end) = 1;

ub = ones(n^2 + n-1, 1);
ub(n^2 + 1 : end) = p-1;

% Initialize Req as [-1 -1 -1 ; -1 -1 -1; -1 -1 -1]:
Req = -1*ones(n);
% Req represents the current search state and will be updated with 0 and 1.

% Initialize the array of Active leaves as only Req:
Active = cell(1);
Active{1} = Req;

% Initialize the best route and best cost as bogus:
bestroute = zeros(n);
bestcost = Inf;

% With all components of linprog save Aeq and beq in place, start the
% branch-and-bound process. Take the first active leaf, compute its Aeq and
% beq and solve. Determine if leaf spawns new leaves. End when Active is
% empty.
progress = 0;
while(size(Active, 2) ~= 0)
    % Get current leaf:
```

```
Req = Active{1};

% Clear leaf from Active list:
Active(1) = [];

% Solve this leaf:
[Route, ~, cost, exitflag] = solveLeaf( cinternal, A, b, lb, ub, Req );

% Store the first LP solution:
if (sum(sum(debugleaf)) == 1)
    debugleaf = Route;
end

% Check if solution is feasible (otherwise, prune by infeasability):
if (exitflag ~= 1)
    progress = progress + AccountsFor(Req);
    disp(progress);
end

if (exitflag == 1)

    % Check if solution is promising (otherwise, prune by bound):
    if (cost >= bestcost)
        progress = progress + AccountsFor(Req);
        disp(progress);
    end

    if (cost < bestcost)

        % If solution is integer, prune by optimality and update.
        if (integersolution(Route))
            bestcost = cost;
            bestroute = Route;
            progress = progress + AccountsFor(Req);
            disp(progress);
        end

        % If solution is not integer, find first fraction and branch:
        if (~integersolution(Route))
            fracFound = 0;
            row = 1;
            col = 1;
            while (~fracFound)
                if (abs(Route(row,col) - floor(Route(row,col) + 0.5)) > 0.01)
                    fracFound = 1;
                else
                    col = col + 1;
                    if (col > n)
                        row = row + 1;
                        col = 1;
                    end
                end
            end
            % Branch:
            Req0 = Req;
            Req0(row, col) = 0;
            Req1 = Req;
            Req1(row, col) = 1;
            if (strcmp(Mode, 'switch')) % Depth until first solution, then width:
                if (bestcost == Inf)
                    Active = [cell(1,2) Active];
                    Active{1} = Req0;
                    Active{2} = Req1;
                else if (bestcost < Inf)
                        Mode = 'width';
                    end
                end
            else if (strcmp(Mode, 'depth')) % Depth-first search:
                    Active = [cell(1,2) Active];
                    Active{1} = Req0;
```

```
                    Active{2} = Req1;
                else % Width-first search:
                    Active{end+1} = Req0;
                    Active{end+1} = Req1;
                end
            end
        end
    end

    % After this while-loop, bestroute and bestcost are returned.

end
%disp(debugleaf)


function [ A, b ] = progger5( m, n, p )
%mTSPprogrammer Makes LP for mTSP for this m and n. p is max visits.
%   When A and b are returned, use them in linprog. Note that the n
%   addresses include the depot (node 1).

% Initialize A. The number of conditions (rows in A) is 2n (visit and leave)
% + (n-1)(n-2) (node potentials) + 1 (sum node potentials)
% + n^2 (placeholders forced equality).
% The number of variables in solution vector x (columns in A)
% is n^2 (x_ij) + n-1 (u_i).
A = zeros(2*n + (n-1)*(n-2) + 1 + n^2 + (n^2 - n), n^2 + (n-1));

% Initialize b. Vector corresponds to the 2n + (n-1)(n-2) + n^2 conditions.
b = zeros(2*n + (n-1)*(n-2) + 1 + n^2 + (n^2-n), 1);

% Row 1: Let m salesmen exit node 1.
A(1, 2:n) = ones(1,n-1);
b(1) = m;

% Row 2: Let m salesmen enter node 1.
for i = 2:n
    A(2, (i-1)*n+1) = 1;
end
b(2) = m;

% The next n-1 rows: Let 1 salesman exit each node.
curRow = 3;
for i = 2:n
    A(curRow, (i-1)*n+1 : i*n) = ones(1,n);
    b(curRow) = 1;
    curRow = curRow + 1;
end

% The next n-1 rows: Let 1 salesman enter each node.
for i = 2:n
    for j = 1:n
        A(curRow, (j-1)*n+i) = 1;
    end
    b(curRow) = 1;
    curRow = curRow + 1;
end

% The next (n-1)(n-2) rows: u_i - u_j + px_ij <= p-1.
for i = 2:n
    for j = 2:n
        if (i ~= j)
            A(curRow, n^2 + (i-1)) = 1; %u_i
            A(curRow, n^2 + (j-1)) = -1; %-u_j
            A(curRow, (i-1)*n+j) = p; %+px_ij
            b(curRow) = p-1; % <= p-1
            curRow = curRow + 1;
        end
    end
end
```

```
% The next row: u_2 + u_3 + ... + u_n <= BOUND
A(curRow, n^2+1:end) = ones(1, n-1);
dummy1 = floor( (n-1)/(p-1) );
dummy2 = mod(n-1, p-1);
b(curRow) = dummy1*(0.5*p*(p-1)) + 0.5*(dummy2+1)*dummy2;
curRow = curRow + 1;

% The next n^2 rows: stakeholder forced equality (x_ij = 0 or = 1).
% Don't need to be changed from 0 (this happens when branch-and-bounding).
curRow = curRow + n^2;

% The next n^2-n rows: no symmetry
for i = 1:n
    for j = 1:n
        if (i ~= j)
            A(curRow, (i-1)*n+j) = 1;
            A(curRow, (j-1)*n+1) = 1;
            b(curRow) = 2;
            curRow = curRow + 1;
        end
    end
end

end

function [ Route, potentials, cost, exitflag ] = solveLeaf( costs, A, b, lb, ub, Req )
%solLeaf Solves this leaf: all LP info is premade, except leaf specific Reqs.
%   Detailed explanation goes here

% Almost all components for linprog are present. Compute Aeq and beq:
[Aeq, beq] = eqer2(A, b, Req);

% Now perform linprog:
[route, cost, exitflag] = linprog(costs, A, b, Aeq, beq, lb, ub);

% Get Route and potentials from route:
n = size(Req, 2);
Route = reshape(route(1:n^2), n, n)';
potentials = route(n^2+1:end);

end

function [ Aeq, beq ] = eqer2( A, b, Req )
%eqer Makes Aeq and beq respecting forced equalities matrix Req
%   If Req is of the form
%   -1 1 -1
%   0 -1 -1
%   -1 -1 -1
%   this means that you demand x_12 = 1 and x_21 = 0.

% Deduce n from Req:
n = size(Req, 1);

% Initialize Aeq as clone of A, without the u_i-u_j+px_ij<p-1 lines:
Aeq = A;
Aeq(2*n+1:end, :) = 0;

% Do the same for beq:
beq = b;
beq(2*n+1:end) = 0;

% Reshape Req to a vector, and add a condition to Aeq and beq wherever Req
% is not equal to -1:
req = reshape(Req', n^2, 1);
for i = 1:length(req)
    if (req(i) ~= -1)
        % Put the 0 or 1 in beq:
        beq(2*n + (n-1)*(n-2) + i + 1) = req(i);
        % Flag the x_ij in Aeq in its dedicated row:
        row = 2*n + (n-1)*(n-2) + i + 1;
```

```
        col = i;
        Aeq(row, col) = 1;
    end
end

end

function [ percentage ] = AccountsFor( Leaf )
%UNTITLED6 Summary of this function goes here
%   Detailed explanation goes here

[m,n] = size(Leaf);
zerosOrOnes = 0;
for i = 1:m
    for j = 1:n
        if (Leaf(i,j) == 0 || Leaf(i,j) == 1)
            zerosOrOnes = zerosOrOnes + 1;
        end
    end
end

percentage = 0.5^zerosOrOnes;

end
```

Appendix F

Code for calling `intlinprog`

```
function [ bestroute, bestcost ] = ilp2015b( Costs, m, p, AdditionalInequalities )
%ilp2015 Find minimal route with m salesmen to no more than p nodes.
%   Supply a matrix Costs with the first row and column reserved for the
%   depot node. Note that in this route, the traveltime from and to the
%   depot are unimportant. This function differs from BnBmode in that it
%   uses the previously unavailable intlinprog(). All this function
%   therefore does is form a matrix of equalities and one of inequalities
%   and enter those. If AdditionalInequalities = [1 3], then only
%   inequality 1 and 3 are used.

% The format is : [x_11, x_12, x_13, ... x_(n-1)n, x_nn, u_2, u_3, ... u_n]

% Get n from the square Costs matrix and weighten the diagonals:
n = size(Costs, 1);
Costs = Costs + (max(max(Costs)))*10*eye(n);

% Get number of x_ij variables, number of u_i variables and total number:
NRx = n^2;
NRu = n-1;
NRt = NRx + NRu;

% Equality 1: m salesmen leave node 1.
E1 = sparse(1, NRt);
E1(1, 1:n) = 1;
E1(1) = 0;
Beq1 = m;

% Equality 2: m salesmen return to node 1.
E2 = sparse(1, NRt);
piece = [1 sparse(1, n-1)];
E2(1:NRx) = repmat(piece, 1, n);
Beq2 = m;

% Equality 3: 1 salesman leaves each other node.
E3 = sparse(n-1, NRt);
for row = 1:n-1
    cols = row*n + 1 : (row+1)*n;
    E3(row, cols) = 1;
    E3(row, cols(row+1)) = 0;
end
Beq3 = ones(n-1, 1);

% Equality 4: 1 salesman enters each other node.
E4 = sparse(n-1, NRt);
for row = 2:n
    piece = sparse(1,n);
    piece(row) = 1;
    E4(row-1, 1:NRx) = repmat(piece, 1, n);
end
Beq4 = ones(n-1, 1);
```

```
% Inequality 1: u_i - u_j + px_ij <= p-1 (Miller).
I1 = sparse(NRu^2, NRt);
B1 = sparse(NRu^2, 1);
for i = 2:n
    for j = 2:n
        row = (i-2)*(n-1) + j-1;
        I1(row, NRx + (i-1)) = 1;
        I1(row, NRx + (j-1)) = -1;
        I1(row, (i-1)*n + j) = p;
        B1(row) = p-1;
    end
end

% Inequality 2: sum of potentials is smaller than H.
q = mod(n-1, p-1);
H = floor((n-1)/(p-1)) * 0.5*p*(p-1) + 0.5*(q+1)*q;
I2 = [sparse(1, NRx) ones(1, NRu)];
B2 = H;

% Inequality 3: x_ij + x_ji <= 1 for all i,j.
I3 = sparse(n^2, NRt);
for i = 1:n
    for j = 1:n
        row = (i-1)*n + j;
        I3(row, (i-1)*n + j) = 1;
        I3(row, (j-1)*n + i) = 1;
    end
end
B3 = ones(n^2, 1);

% Form inequality matrix A, inequality vector b, equality matrix Aeq and
% equality vector beq.
Aeq = [E1 ; E2 ; E3; E4];
beq = [Beq1 ; Beq2 ; Beq3 ; Beq4];
A = [];
b = [];
if (ismember(1, AdditionalInequalities))
    A = [A ; I1];
    b = [b ; B1];
end
if (ismember(2, AdditionalInequalities))
    A = [A ; I2];
    b = [b ; B2];
end
if (ismember(3, AdditionalInequalities))
    A = [A ; I3];
    b = [b ; B3];
end

% Form cost function f and execute intlinprog.
piece = ones(1, n);
piece(1) = 0;
f = [zeros(1, n) repmat(piece, 1, n-1) zeros(1, NRu)];
f(1:n^2) = f(1:n^2) .* reshape(Costs', 1, n^2);

%[bestroute, bestcost] = intlinprog(f, 1:NRt, A, b, Aeq, beq);
lb = [zeros(NRx, 1) ; ones(NRu, 1)];
ub = [ones(NRx, 1) ; (p-1)*ones(NRu, 1)];
[sol, bestcost] = intlinprog(f, 1:NRt, A, b, Aeq, beq, lb, ub);
bestroute = reshape(sol(1:n^2), n, n)';
end
```

The function `ilp2015bLIM` is identical, save for the last few lines:

```
lb = [zeros(NRx, 1) ; ones(NRu, 1)];
ub = [ones(NRx, 1) ; (p-1)*ones(NRu, 1)];
options = optimoptions('intlinprog', 'MaxNodes', 1e6);
[sol, bestcost] = intlinprog(f, 1:NRt, A, b, Aeq, beq, lb, ub, options);
bestroute = reshape(sol(1:n^2), n, n)';

function [ bestroute, bestcost ] = ilp2015kLIM( Costs, m, p, AdditionalInequalities, ClientPreferences )
```

```
%ilp2015 Find minimal Baby route with m salesmen to no more than p nodes.
%   Supply a matrix Costs with the first row and column reserved for the
%   depot node. Note that in a Baby route, the traveltime from and to the
%   depot are unimportant. This function differs from BnBmode in that it
%   uses the previously unavailable intlinprog(). All this function
%   therefore does is form a matrix of equalities and one of inequalities
%   and enter those. If AdditionalInequalities = [1 3], then only
%   inequality 1 and 3 are used.

% The format is : [x_11, x_12, x_13, ... x_(n-1)n, x_nn, u_2, u_3, ... u_n]

% Get n from the square Costs matrix and weighten the diagonals:
n = size(Costs, 1);
Costs = Costs + (max(max(Costs)))*10*eye(n);

% Get number of x_ij variables, number of u_i variables and total number:
NRx = n^2;
NRm = m*NRx;
NRu = n-1;
NRt = NRm + NRu;

% Equality 1: for each of the m routes, 1 salesman leaves the depot node to
% an allowed node.
E1 = sparse(m, NRt);
for row = 1:m
    columnshift = (row-1)*NRx;
    for j = 1:n
        if (j ~= 1) %ismember(j, CP(row,:)) &&
            col = columnshift + j;
            E1(row, col) = 1;
        end
    end
end
Beq1 = ones(m,1);

% Equality 2: for each of the m routes, 1 salesman returns to depot node
% from an allowed node.
E2 = sparse(m, NRt);
for row = 1:m
    for j = 1:n
        if (j ~= 1) %ismember(j, CP(row,:)) &&
            col = (row-1)*NRx + (j-1)*n + 1;
            E2(row, col) = 1;
            %end
        end
    end
    Beq2 = ones(m,1);

    % Equality 3: each other node is left by 1 salesman on an allowed day.
    E3 = sparse(n-1, NRt);
    for row = 1:n-1
        for i = 1:m
            for j = 1:n
                %if (row+1 ~= j) % && ismember(row+1, CP(i,:)) && ismember(j, CP(i,:))
                col = (i-1)*NRx + row*n + j;
                E3(row, col) = 1;
            end
        end
    end
end
Beq3 = ones(n-1, 1);

% Equality 4: 1 salesman enters each other node.
E4 = sparse(n-1, NRt);
for row = 1:n-1
    for i = 1:m
        for j = 1:n
            %if (row+1 ~= j) % && ismember(row+1, CP(i,:)) && ismember(j, CP(i,:))
            col = (i-1)*NRx + (j-1)*n + row+1;
            E4(row, col) = 1;
```

```
            %end
        end
    end
end
Beq4 = ones(n-1, 1);

% Equality 5: if node j is entered on route k, it must be left on route k.
% So with j and k known, sum(i) x_ijk = sum(l) x_jlk, thus sum - sum =0.
E5 = sparse(n*m, NRt);
Beq5 = zeros(n*m, 1);
for j = 1:n
    for k = 1:m
        row = (k-1)*n + j;
        % sum(i) x_ijk:
        for i = 1:n
            if (i ~= j)
                col = (k-1)*NRx + (i-1)*n + j;
                E5(row, col) = 1;
            end
        end
        % - sum(l) x_jlk:
        for l = 1:n
            if (l ~= j)
                col = (k-1)*NRx + (j-1)*n + l;
                E5(row, col) = -1;
            end
        end
    end
end

% Inequality 1: u_i - u_j + px_ij <= p-1 (Miller).
I1 = sparse(m*NRu^2, NRt);
B1 = (p-1)*ones(m*NRu^2, 1);
for k = 1:m
    rowshift = (k-1)*NRu^2;
    columnshift = (k-1)*NRx;
    for i = 2:n
        for j = 2:n
            row = (i-2)*(n-1) + j-1 + rowshift;
            I1(row, NRm + (i-1)) = 1;
            I1(row, NRm + (j-1)) = -1;
            I1(row, (i-1)*n + j + columnshift) = p;
        end
    end
end

% % Inequality 2: sum(k) x_ij <= 1 (single tour allocation).
I2 = sparse(NRx, NRt);
B2 = ones(NRx, 1);
for i = 1:n
    for j = 1:n
        index = (i-1)*n + j;
        for k = 1:m
            columnshift = (k-1)*NRx;
            I2(index, index + columnshift) = 1;
        end
    end
end

% Form inequality matrix A, inequality vector b, equality matrix Aeq and
% equality vector beq.
Aeq = [E1 ; E2 ; E3; E4 ; E5];
beq = [Beq1 ; Beq2 ; Beq3 ; Beq4 ; Beq5];
A = [];
b = [];
if (ismember(1, AdditionalInequalities))
    A = [A ; I1];
    b = [b ; B1];
end
if (ismember(2, AdditionalInequalities))
```

```
    A = [A ; I2];
    b = [b ; B2];
end
if (ismember(3, AdditionalInequalities))
    A = [A ; I3];
    b = [b ; B3];
end

% Form cost function f and execute branch and bound.
f = zeros(1, NRt);
lb = [zeros(NRm, 1) ; ones(NRu, 1)];
ub = [ones(NRm, 1) ; (p-1)*ones(NRu, 1)];
% Assign x_ijk the cost c_ij (and upper bound 1) when relevant:
for i = 1:n
    for j = 1:n
        for k = 1:m
            %if (i ~= j) %&& ismember(i, CP(k,:)) && ismember(j, CP(k,:))
                index_ijk = (k-1)*NRx + (i-1)*n + j;
                if (i > 1 && j > 1)
                    f( index_ijk ) = Costs(i,j);
                end
                % Disallow certain arcs on certain days according to
                % ClientPreferences:
                if (~ismember(j, ClientPreferences(k,:)))
                    ub( index_ijk ) = 0;
                end
            %end
        end
    end
end

% Use intlinprog to generate optimal solution:
options = optimoptions('intlinprog', 'MaxNodes', 1e6);
[sol, bestcost] = intlinprog(f, 1:NRt, A, b, Aeq, beq, lb, ub, options);
bestroute = sol;

% % Convert the solution vector sol to a Droute:
% bestroute = ilp2des(m, n, p, sol);

end
```

Appendix G

Code for greedy search and simulated annealing

```
function [ foundDroute, foundcost ] = GreedySearch( m, p, C, trials, allowance)
%GreedySearch Just looks for the best neighbour until it's locally optimal.
% Pick a random neighbour. If better, accept. If not, try again. If there
% has been no success in allowance tries, consider this solution locally
% optimal. Try this process a trials amount of times.

% Get n from C:
n = size(C, 1);

% Initialize the best found route and best cost poorly:
foundDroute = zeros(m, p-1);
foundcost = Inf;

% Perform process trials times:
printcounter = 0;
for i = 1:trials
    % Indicate progress.
    printcounter = printcounter + 1;
    if (printcounter == 20)
        disp(i/trials);
        printcounter = 0;
    end

    % Initialize the temporary best randomly.
    tempDroute = randDroute(m, n, p);
    tempcost = costrouteinternal(C, des2bin(tempDroute));

    allowancecounter = 0;
    while (allowancecounter < allowance)
        % Find a random neighbour.
        neighDroute = randop(tempDroute);
        neighcost = costrouteinternal(C, des2bin(neighDroute));
        % If this cost is better than the tempcost, change tempDroute and
        % reset counter.
        if (neighcost <= tempcost)
            tempDroute = neighDroute;
            tempcost = neighcost;
            allowancecounter = 0;
        end
        % Otherwise, add to the allowance counter.
        if (neighcost > tempcost)
            allowancecounter = allowancecounter + 1;
        end
        % If a change has occured, check if this is the best yet.
        if (neighcost == tempcost)
            if (tempcost < foundcost)
                foundDroute = tempDroute;
                foundcost = tempcost;
```

```
            end
        end
    end
end

function [ foundDroute, foundcost ] = greedyk( m, p, C, trials, allowance, ClientPreferences)
%GreedySearch Just looks for the best neighbour until it's locally optimal.
% Pick a random neighbour. If better, accept. If not, try again. If there
% has been no success in allowance tries, consider this solution locally
% optimal. Try this process a trials amount of times.

% Get n from C:
n = size(C, 1);

% Initialize the best found route and best cost poorly:
foundDroute = zeros(m, p-1);
foundcost = Inf;

% Perform process trials times:
printcounter = 0;
for i = 1:trials
    % Indicate progress.
    printcounter = printcounter + 1;
    if (printcounter == 20)
        disp(i/trials);
        printcounter = 0;
    end

    % Initialize the temporary best randomly.
    tempDroute = randDroutek(m, n, p, ClientPreferences);
    tempcost = costrouteinternal(C, des2bin(tempDroute));

    allowancecounter = 0;
    while (allowancecounter < allowance)
        % Find a random neighbour.
        neighDroute = randopk(tempDroute, ClientPreferences);
        neighcost = costrouteinternal(C, des2bin(neighDroute));
        % If this cost is better than the tempcost, change tempDroute and
        % reset counter.
        if (neighcost < tempcost)
            tempDroute = neighDroute;
            tempcost = neighcost;
            allowancecounter = 0;
        end
        % Otherwise, add to the allowance counter.
        if (neighcost >= tempcost)
            allowancecounter = allowancecounter + 1;
        end
        % If a change has occured, check if this is the best yet.
        if (neighcost == tempcost)
            if (tempcost < foundcost)
                foundDroute = tempDroute;
                foundcost = tempcost;
            end
        end
    end
end

function [ foundDroute, foundcost ] = siman6( m, p, C, AllOverTrials, stepsPerTemp, startingTemperature, endTemperature )
%SimulatedAnnealing Solve the mTSP with simulated annealing.
%   Attempt to solve the mTSP with m salesmen, costs C and p the number of
%   cities any salesman may visit, including the return to depot. If the
%   vector Odds equals [1 0.5 0], a random starting solution will be
%   generated and the first random neighbouring solution will be accepted
%   with 100% certainty, the second will be accepted when better or at a
%   50% odds and the third will only be accepted when it's better. Repeat
%   this process a trials amount of times and return the best ever found.

debugodds = zeros(1,500*stepsPerTemp);
debugaccepted = zeros(1,500*stepsPerTemp);
```

```
debugcounter = 0;
nexttempcounter = 0;

% Warn for inappropriate parameters:
if (endTemperature <= 0)
    disp('Warning: endTemperature <= 0. While-loop never ends.');
end

% Get n from C:
n = size(C, 1);

% Initialize the best found route and best cost poorly:
foundDroute = zeros(m, p-1);
foundcost = Inf;

% Perform process trials times:
printcounter = 0;
for i = 1:AllOverTrials
    disp(i)
    stepsinrun = 0;
    % Indicate progress.
    printcounter = printcounter + 1;
    if (printcounter == 100)
        disp(i/AllOverTrials);
        printcounter = 0;
    end

    % Initialize the temporary best randomly.
    tempDroute = randDroute(m, n, p);
    tempcost = costrouteinternal(C, des2bin(tempDroute));
    temperature = startingTemperature;
    % Until running is set to 0, do the following:
    % - Find a random neighbour.
    % - If the neighbour has a better cost, let tempDroute become this.
    % - If not, let tempDroute become this against odds Odds(j).
    % - Keep checking for the best found yet.
    while(temperature > endTemperature)
        stepsinrun = stepsinrun + 1;
        %odds = 1;
        % Find a random neighbour.
        neighDroute = randop(tempDroute);
        neighcost = costrouteinternal(C, des2bin(neighDroute));
        % If this cost is better than the tempcost, change tempDroute.
        if (neighcost <= tempcost)
            tempDroute = neighDroute;
            tempcost = neighcost;
        end
        % Otherwise, change against computed odds.
        if (neighcost > tempcost)
            odds = exp((tempcost - neighcost)/temperature);

            if (i == AllOverTrials)
                debugcounter = debugcounter + 1;
                debugodds(debugcounter) = odds;
            end

            if(rand(1) <= odds)

                if (i == AllOverTrials)
                    debugaccepted(debugcounter) = debugodds(debugcounter);
                end

                tempDroute = neighDroute;
                tempcost = neighcost;
            end
        end
        % If a change has occured, check if this is the best yet.
        if (neighcost == tempcost)
            if (tempcost < foundcost)
                foundDroute = tempDroute;
```

```
                foundcost = tempcost;
            end
        end
        % Update temperature and check for termination.
        nexttempcounter = nexttempcounter + 1;
        if (nexttempcounter >= stepsPerTemp)
            temperature = 0.95*temperature;
            nexttempcounter = 0;
            disp('Temperature: ');
            disp(temperature);
        end
    end
end
%if (debugcounter < length(debugodds))
%    debugodds = debugodds(1:debugcounter);
%    debugaccepted = debugaccepted(1:debugcounter);
%end
%plot(debugodds);
%hold on
%scatter(1:debugcounter, debugaccepted, 'r');


function [ foundDroute, foundcost ] = siman7k( m, p, C, AllOverTrials, stepsPerTemp, alpha, startingTemperature, endTemperature
%SimulatedAnnealing Solve the mTSP with simulated annealing.
%   Attempt to solve the mTSP with m salesmen, costs C and p the number of
%   cities any salesman may visit, including the return to depot. If the
%   vector Odds equals [1 0.5 0], a random starting solution will be
%   generated and the first random neighbouring solution will be accepted
%   with 100% certainty, the second will be accepted when better or at a
%   50% odds and the third will only be accepted when it's better. Repeat
%   this process a trials amount of times and return the best ever found.

debugodds = zeros(1,500*stepsPerTemp);
debugaccepted = zeros(1,500*stepsPerTemp);
debugcounter = 0;
nexttempcounter = 0;

% Warn for inappropriate parameters:
if (endTemperature <= 0)
    disp('Warning: endTemperature <= 0. While-loop never ends.');
end

% Get n from C:
n = size(C, 1);

% Initialize the best found route and best cost poorly:
foundDroute = zeros(m, p-1);
foundcost = Inf;

% Perform process trials times:
printcounter = 0;
for i = 1:AllOverTrials
    disp(i)
    stepsinrun = 0;
    % Indicate progress.
    printcounter = printcounter + 1;
    if (printcounter == 100)
        disp(i/AllOverTrials);
        printcounter = 0;
    end

    % Initialize the temporary best randomly.
    tempDroute = randDroutek(m, n, p, ClientPreferences);
    tempcost = costrouteinternal(C, des2bin(tempDroute));
    temperature = startingTemperature;
    % Until running is set to 0, do the following:
    % - Find a random neighbour.
    % - If the neighbour has a better cost, let tempDroute become this.
    % - If not, let tempDroute become this against odds Odds(j).
    % - Keep checking for the best found yet.
    while(temperature > endTemperature)
```

```
        stepsinrun = stepsinrun + 1;
        %odds = 1;
        % Find a random neighbour.
        neighDroute = randopk(tempDroute, ClientPreferences);
        neighcost = costrouteinternal(C, des2bin(neighDroute));
        % If this cost is better than the tempcost, change tempDroute.
        if (neighcost <= tempcost)
            tempDroute = neighDroute;
            tempcost = neighcost;
        end
        % Otherwise, change against computed odds.
        if (neighcost > tempcost)
            odds = exp((tempcost - neighcost)/temperature);

            if (i == AllOverTrials)
                debugcounter = debugcounter + 1;
                debugodds(debugcounter) = odds;
            end

            if(rand(1) <= odds)

                if (i == AllOverTrials)
                    debugaccepted(debugcounter) = debugodds(debugcounter);
                end

                tempDroute = neighDroute;
                tempcost = neighcost;
            end
        end
        % If a change has occured, check if this is the best yet.
        if (neighcost == tempcost)
            if (tempcost < foundcost)
                foundDroute = tempDroute;
                foundcost = tempcost;
            end
        end
        % Update temperature and check for termination.
        nexttempcounter = nexttempcounter + 1;
        if (nexttempcounter >= stepsPerTemp)
            temperature = alpha*temperature;
            nexttempcounter = 0;
            disp('Temperature: ');
            disp(temperature);
        end
    end
end
%if (debugcounter < length(debugodds))
%    debugodds = debugodds(1:debugcounter);
%    debugaccepted = debugaccepted(1:debugcounter);
%end
%plot(debugodds);
%hold on
%scatter(1:debugcounter, debugaccepted, 'r');


function [ Droute ] = randDroute( m, n, p )
%randDroute Initialize a random descriptive route.
%   Detailed explanation goes here

% Initialize Droute:
Droute = zeros(m, p - 1);

% Create one big random permutation of 2:n, later to be divided:
perm = randperm(n-1);
perm = perm + 1;

% The first few elements of perm are placed in the first row of Droute, the
% next few in the second row, et cetera. To determine at which element to
% cut off this section of perm and begin the next, find m - 1 different
% random integers smaller than n - 1 and order them:
cutoffs = randi(n-1, 1, m-1);
```

```
cutoffs = [0 sort(cutoffs) n-1];

% If it contains duplicates or cutoff distances larger than p-1, try again:
duplicates = (length(cutoffs) ~= length(unique(cutoffs)));
dummy = dist(cutoffs);
toolarge = (max(diag(dummy(2:end, 1:end-1))) > p-1 );

while (duplicates || toolarge)
    cutoffs = randi(n-2, 1, m-1);
    cutoffs = [0 sort(cutoffs) n-1];
    duplicates = (length(cutoffs) ~= length(unique(cutoffs)));
    dummy = dist(cutoffs);
    toolarge = (max(diag(dummy(2:end, 1:end-1))) > p-1 );
end

% Now fill in Droute with sections of perm, the sizes of the sections being
% determined from cutoffs:
for i = 1:m
    a = cutoffs(i)+1;
    b = cutoffs(i+1);
    sec = perm(a:b);
    l = length(sec);
    Droute(i, 1:l) = sec;
end


function [ newDroute ] = randop( Droute )
%randop Randomly perform op1, op2, op3, op4 or op5.
%   Detailed explanation goes here

newDroute = Droute;

% Obtain m, n and p from Droute:
[m, P] = size(Droute);
p = P + 1;

% Randomly select which operation to perform, with 4 or 5 only if m > 1:
if (m == 1)
    op = randi(3);
end
if (m > 1 && ~ismember(0, Droute))
    op = randi(4);
end
if (m > 1 && ismember(0, Droute))
    op = randi(5);
end

% If op = 1, randomly perform operation 1:
if (op == 1)
    % Randomly select which row to change.
    row = randi(m);
    % Get the length of the non-zero part.
    nonzero = unique(Droute(row,:));
    if (nonzero(1) == 0)
        nonzero(1) = [];
    end
    l = length(nonzero);
    % Get a random pair out of 1:l.
    NRpairs = 0.5*l*(l-1);
    if (NRpairs > 0)
        [a,b] = number2pair(randi(NRpairs), l);
        % Perform op1.
        newDroute = op1(Droute, row, a, b);
    end
    if (NRpairs <= 0)
        newDroute = Droute;
    end
end

% If op = 2, randomly perform operation 2:
if (op == 2)
```

```
    % Randomly select which row to change.
    row = randi(m);
    % Get the length of the non-zero part.
    nonzero = unique(Droute(row,:));
    if (nonzero(1) == 0)
        nonzero(1) = [];
    end
    l = length(nonzero);
    % Randomly find a <= b < c <= d, representing the columns a:b and c:d.
    randsecs = sort(randi(l, 1, 4));
    while (~(randsecs(2) < randsecs(3)))
        randsecs = sort(randi(l, 1, 4));
    end
    a = randsecs(1);
    b = randsecs(2);
    c = randsecs(3);
    d = randsecs(4);
    % Perform op2.
    newDroute = op2(Droute, row, a:b, c:d);
end

% If op = 3, randomly perform operation 3:
if (op == 3)
    % Randomly select which row to change.
    row = randi(m);
    % Get the length of the non-zero part.
    nonzero = unique(Droute(row,:));
    if (nonzero(1) == 0)
        nonzero(1) = [];
    end
    l = length(nonzero);
    % Get a random pair out of 1:l.
    randsec = sort(randi(l, 1, 2));
    % Depending on the length of the section defined this way, only some
    % spots are allowed. Determine those.
    lengthsec = randsec(2) - randsec(1) + 1;
    MAXtocol = l - lengthsec + 1;
    % Perform op3.
    newDroute = op3(Droute, row, randsec(1):randsec(2), randi(MAXtocol));
end

% If op = 4, randomly perform operation 4:
if (op == 4)
    % Randomly select two different rows to change.
    row1 = randi(m);
    row2 = randi(m);
    while (row1 == row2)
        row2 = randi(m);
    end
    % Get the length of both non-zero parts.
    nonzero1 = unique(Droute(row1,:));
    if (nonzero1(1) == 0)
        nonzero1(1) = [];
    end
    l1 = length(nonzero1);
    nonzero2 = unique(Droute(row2,:));
    if (nonzero2(1) == 0)
        nonzero2(1) = [];
    end
    l2 = length(nonzero2);
    % Get a random pair from 1:l1 and one from 1:l2. Make sure the switch
    % between these two sections doesn't lead to illegal sizes.
    randsec1 = sort(randi(l1, 1, 2));
    randsec2 = sort(randi(l2, 1, 2));
    lr1 = randsec1(2) - randsec1(1) + 1;
    lr2 = randsec2(2) - randsec2(1) + 1;
    while (l1 - lr1 + lr2 > p - 1 || l2 - lr2 + lr1 > p - 1)
        randsec1 = sort(randi(l1, 1, 2));
        randsec2 = sort(randi(l2, 1, 2));
        lr1 = randsec1(2) - randsec1(1) + 1;
```

```
        lr2 = randsec2(2) - randsec2(1) + 1;
    end
    % With these valid pairs, perform op4.
    a = randsec1(1);
    b = randsec1(2);
    c = randsec2(1);
    d = randsec2(2);
    newDroute = op4(Droute, row1, row2, a:b, c:d);
end

% If op = 5, randomly perform operation 5:
if (op == 5)
    % Randomly select two different rows to change.
    row1 = randi(m);
    row2 = randi(m);
    while (row1 == row2)
        row2 = randi(m);
    end
    % Get the length of both non-zero parts.
    nonzero1 = unique(Droute(row1,:));
    if (nonzero1(1) == 0)
        nonzero1(1) = [];
    end
    l1 = length(nonzero1);
    nonzero2 = unique(Droute(row2,:));
    if (nonzero2(1) == 0)
        nonzero2(1) = [];
    end
    l2 = length(nonzero2);
    % Check if row1 has more than one element.
    ok1 = (l1 > 1);
    % If so, select a random region from row1 no larger than l1-1 or
    % p-1-l2.
    if (ok1)
        nlt = min(l1-1, p-1-l2);
        ok2 = (nlt > 0);
        if (ok2)
            lengthsec = randi(nlt);
            leftbound = randi(l1-lengthsec+1);
            rightbound = leftbound+lengthsec-1;
            cols = leftbound:rightbound;
            % Then perform op5.
            newDroute = op5(Droute, row1, row2, cols);
        end
    end

% randop may have lead to nothing. In that case, repeat.
if (sum(sum(Droute == newDroute)) == m*P)
    newDroute = randop(Droute);
end

end


function [ newDroute ] = op1( Droute, row, startcol, endcol )
%op1 Operation 1/5: Intraroute inversion.
%   Given a descriptive route and row, invert the sequence starting at
%   startcol and ending at endcol.
%   Example:
%   A = [ 11 13 15 17 19 ; 8 8 8 8 8 ]
%   op1(A, 1, 2, 5)
%   ans = [ 11 19 17 15 13 ; 8 8 8 8 8 ]

% Find the section to be inverted and invert:
relevantsubmatrix = Droute(row, startcol:endcol);
flipped = fliplr(relevantsubmatrix);

% Let newDroute be a clone of Droute with the section flipped:
newDroute = Droute;
newDroute(row, startcol:endcol) = flipped;
```

```
end

function [ newDroute ] = op2( Droute, row, Firstcols, Seccols )
%op2 Operation 2/5: Intraroute switching.
%   Given a descriptive route and row, switch the locations of the sequence
%   at Firstcols and the sequence at Seccols.
%   Example:
%   A = [ 11 13 15 17 19 ; 8 8 8 8 8 ]
%   op2(A, 1, 1:3, 4:5)
%   ans = [ 17 19 11 13 15 ; 8 8 8 8 8 ]

% Obtain the relevant sections:
Firstsec = Droute(row, Firstcols);
Secsec = Droute(row, Seccols);

% Also obtain the part of the relevant row before the first section,
% between the two sections and after the second section:
Presec = [];
if (Firstcols(1) ~= 1)
    Presec = Droute(row, 1: (Firstcols(1)-1) );
end
Intersec = [];
if (Firstcols(end) < Seccols(1))
    Intersec = Droute(row, (Firstcols(end) + 1) : (Seccols(1) - 1) );
end
Postsec = [];
if (Seccols(end) ~= size(Droute, 2))
    Postsec = Droute(row, (Seccols(end) + 1) : end);
end

% Clone newDroute out of Droute but replace the relevant row with a
% concatination of the five sections:
newDroute = Droute;
newDroute(row, :) = [Presec Secsec Intersec Firstsec Postsec];

end

function [ newDroute ] = op3( Droute, row, Fromcols, Tocol )
%op3 Operation 3/5: Intraroute insertion.
%   Given a descriptive route and row, take the sequence at Fromcols and
%   have it start at position Tocol.
%   Example:
%   A = [ 11 13 15 17 19 ; 8 8 8 8 8 ]
%   op3(A, 1, 1:2, 3)
%   ans = [ 15 17 11 13 19 ; 8 8 8 8 8 ]

% Obtain the relevant row:
relrow = Droute(row,:);

% Take out the To Be Inserted columns:
tbi = relrow(Fromcols);
relrow(Fromcols) = [];

% Initialize the new row:
newrow = zeros(1, size(Droute, 2));

% Overlay the extracted tbi at the proper place:
l = length(tbi);
newrow( Tocol:(Tocol+l-1) ) = tbi;

% Fill up the remaining zeros with what's left in the relevant row:
for i = 1:size(Droute,2)
    if (newrow(i) == 0 && ~isempty(relrow))
        newrow(i) = relrow(1);
        relrow(1) = [];
    end
end

% Clone newDroute out of Droute but replace the relevant row with the new:
newDroute = Droute;
```

```
newDroute(row, :) = newrow;

end

function [ newDroute ] = op4( Droute, row1, row2, cols1, cols2 )
%op4 Operation 4/5: Interroute switching.
%   Given a descriptive route and two rows, exchange the sequence at
%   (row1, cols1) with the one at (row2, cols2) (given that this doesn't
%   cause newDroute to be wider than Droute (width remains p-1)).
%   Example:
%   A = [ 11 13 15 17 19 ; 8 8 8 8 8 ]
%   op4(A, 1, 2, 1:2, 4:5)
%   ans = [ 8 8 15 17 19 ; 8 8 8 11 13 ]

% Obtain the relevant rows and p:
relrow1 = Droute(row1, :);
relrow2 = Droute(row2, :);

% Extract the relevant sections:
sec1 = relrow1(cols1);
sec2 = relrow2(cols2);
relrow1(cols1) = [];
relrow2(cols2) = [];

% Place each extracted section at the start of the other trimmed row:
relrow1 = [sec2 relrow1];
relrow2 = [sec1 relrow2];

% If relrows are too short or long, fix this by adding or removing zeros:
properwidth = size(Droute, 2);
difference1 = length(relrow1) - properwidth;
difference2 = length(relrow2) - properwidth;
if (difference1 > 0)
    relrow1 = relrow1(1:properwidth);
elseif (difference1 < 0)
        relrow1 = [relrow1 zeros(1, abs(difference1))];
end
if (difference2 > 0)
    relrow2 = relrow2(1:properwidth);
elseif (difference2 < 0)
        relrow2 = [relrow2 zeros(1, abs(difference2))];
end

% Use operation 3 (intraroute insertion) to get them to the right place:
relrow1 = op3(relrow1, 1, 1:length(sec2), cols1(1));
relrow2 = op3(relrow2, 1, 1:length(sec1), cols2(1));

% Clone newDroute out of Droute but replace the relevant rows with the new:
newDroute = Droute;
newDroute(row1, :) = relrow1;
newDroute(row2, :) = relrow2;

end

function [ newDroute ] = op5( Droute, row1, row2, cols1)
%op4 Operation 5/5: Interroute transfer.
%   Given a descriptive route and two rows, remove the sequence at
%   (row1, cols1) and staple it to the end of row2(given that this doesn't
%   cause newDroute to be wider than Droute (width remains p-1)).
%   Example:
%   A = [ 11 13 15 17 19 ; 8 9 0 0 0 ]
%   op5(A, 1, 2, 1:2)
%   ans = [ 15 17 19 0 0 ; 8 9 11 13 0 ]

% If all goes wrong, return Droute:
newDroute = Droute;

% Obtain the relevant rows and p:
relrow1 = Droute(row1, :);
relrow2 = Droute(row2, :);
```

```
p = size(Droute, 2) + 1;

% Check if a valid regions of columns is being extracted from row1:
size1 = length(unique(relrow1));
if (ismember(0, relrow1))
    size1 = size1 - 1;
end
ok1 = (cols1(1) >= 1 && cols1(end) <= size1 && size1 - length(cols1) >= 1);

% Count the non-zero elements in row2, to see if stapling is allowed:
size2 = length(unique(relrow2));
if (ismember(0, relrow2))
    size2 = size2 - 1;
end
ok2 = (size2 + length(cols1) <= p-1);

% If these sizes are legal, perform the operation:
if (ok1 && ok2)

    % Extract the nonzeros that remain in row1, the nonzeros that move to
    % row2 and the nonzeros of row2.
    sec1 = relrow1(cols1);
    sec2 = relrow1(1:size1);
    sec2(cols1) = [];
    sec3 = relrow2(1:size2);

    % Form newDroute by replacing the changed rows:
    %newDroute = Droute;
    newrow1 = [sec2, zeros(1, (p-1)-length(sec2))];
    newrow2 = [sec3, sec1, zeros(1, (p-1)-length(sec3)-length(sec1))];
    newDroute(row1,:) = newrow1;
    newDroute(row2,:) = newrow2;
end

if (~(ok1 && ok2))
    disp('Warning: illegal op5 request. op5 not performed.');
end

end


function [ newDroute ] = randopk( Droute, ClientPreferences )
%randop Randomly perform op1, op2, op3, op4 or op5.
%   Detailed explanation goes here

newDroute = Droute;

% Obtain m, n and p from Droute:
[m, P] = size(Droute);
p = P + 1;

% Randomly select which operation to perform, with 4 or 5 only if m > 1:
if (m == 1)
    op = randi(3);
end
if (m > 1 && ~ismember(0, Droute))
    op = randi(4);
end
if (m > 1 && ismember(0, Droute))
    op = randi(5);
end

% If op = 1, randomly perform operation 1:
if (op == 1)
    % Randomly select which row to change.
    row = randi(m);
    % Get the length of the non-zero part.
    nonzero = unique(Droute(row,:));
    if (nonzero(1) == 0)
        nonzero(1) = [];
    end
```

```
    l = length(nonzero);
    % Get a random pair out of 1:l.
    NRpairs = 0.5*l*(l-1);
    if (NRpairs > 0)
        [a,b] = number2pair(randi(NRpairs), l);
        % Perform op1.
        newDroute = op1(Droute, row, a, b);
    end
    if (NRpairs <= 0)
        newDroute = Droute;
    end
end

% If op = 2, randomly perform operation 2:
if (op == 2)
    % Randomly select which row to change.
    row = randi(m);
    % Get the length of the non-zero part.
    nonzero = unique(Droute(row,:));
    if (nonzero(1) == 0)
        nonzero(1) = [];
    end
    l = length(nonzero);
    % Randomly find a <= b < c <= d, representing the columns a:b and c:d.
    randsecs = sort(randi(l, 1, 4));
    while (~(randsecs(2) < randsecs(3)))
        randsecs = sort(randi(l, 1, 4));
    end
    a = randsecs(1);
    b = randsecs(2);
    c = randsecs(3);
    d = randsecs(4);
    % Perform op2.
    newDroute = op2(Droute, row, a:b, c:d);
end

% If op = 3, randomly perform operation 3:
if (op == 3)
    % Randomly select which row to change.
    row = randi(m);
    % Get the length of the non-zero part.
    nonzero = unique(Droute(row,:));
    if (nonzero(1) == 0)
        nonzero(1) = [];
    end
    l = length(nonzero);
    % Get a random pair out of 1:l.
    randsec = sort(randi(l, 1, 2));
    % Depending on the length of the section defined this way, only some
    % spots are allowed. Determine those.
    lengthsec = randsec(2) - randsec(1) + 1;
    MAXtocol = l - lengthsec + 1;
    % Perform op3.
    newDroute = op3(Droute, row, randsec(1):randsec(2), randi(MAXtocol));
end

% If op = 4, randomly perform operation 4k:
if (op == 4)
    % Randomly select two different rows to change.
    row1 = randi(m);
    row2 = randi(m);
    while (row1 == row2)
        row2 = randi(m);
    end
    % Get the length of both non-zero parts.
    nonzero1 = unique(Droute(row1,:));
    if (nonzero1(1) == 0)
        nonzero1(1) = [];
    end
    l1 = length(nonzero1);
```

```
    nonzero2 = unique(Droute(row2,:));
    if (nonzero2(1) == 0)
        nonzero2(1) = [];
    end
    l2 = length(nonzero2);
    % Check which elements of row1 may move to row2 and vice versa without
    % violating the ClientPreferences.
    legal = [ismember(Droute(row1, :), ClientPreferences(row2, :)) ;
        ismember(Droute(row2, :), ClientPreferences(row1, :))];
    % Continue only if a legal exchange exists.
    if (max(legal(1, :)) > 0 && max(legal(2, :)) > 0)
        % Get a random legal pair from row1 and one from row2.
        randsec1 = randomlegalpair(legal(1, :));
        randsec2 = randomlegalpair(legal(2, :));
        lr1 = randsec1(2) - randsec1(1) + 1;
        lr2 = randsec2(2) - randsec2(1) + 1;
        % If this switch would lead to illegal sizes, try again.
        while (l1 - lr1 + lr2 > p - 1 || l2 - lr2 + lr1 > p - 1)
            randsec1 = randomlegalpair(legal(1, :));
            randsec2 = randomlegalpair(legal(2, :));
            lr1 = randsec1(2) - randsec1(1) + 1;
            lr2 = randsec2(2) - randsec2(1) + 1;
        end
        % With these valid pairs, perform op4.
        a = randsec1(1);
        b = randsec1(2);
        c = randsec2(1);
        d = randsec2(2);
        newDroute = op4(Droute, row1, row2, a:b, c:d);
    end
end

% If op = 5, randomly perform operation 5:
if (op == 5)
    % Randomly select two different rows to change.
    row1 = randi(m);
    row2 = randi(m);
    while (row1 == row2)
        row2 = randi(m);
    end
    % Get the length of both non-zero parts.
    nonzero1 = unique(Droute(row1,:));
    if (nonzero1(1) == 0)
        nonzero1(1) = [];
    end
    l1 = length(nonzero1);
    nonzero2 = unique(Droute(row2,:));
    if (nonzero2(1) == 0)
        nonzero2(1) = [];
    end
    l2 = length(nonzero2);
    % Check if row1 has more than one element.
    ok1 = (l1 > 1);
    % Check if a legal exchange from row1 to row2 exists.
    legal = ismember(Droute(row1, :), ClientPreferences(row2, :));
    ok2 = max(legal) > 0;
    % If so, select a random region from row1 no larger than l1-1 or
    % p-1-l2.
    if (ok1 && ok2)
        nlt = min(l1-1, p-1-l2);
        ok3 = (nlt > 0);
        if (ok3)
            randompair = randomlegalpair(legal);
            while (randompair(2) - randompair(1) + 1 > nlt)
                randompair = randomlegalpair(legal);
            end
            cols = randompair(1):randompair(2);
            % Then perform op5.
            newDroute = op5(Droute, row1, row2, cols);
        end
```

```
    end

    % randopk may have lead to nothing. In that case, repeat.
    if (sum(sum(Droute == newDroute)) == m*P)
        newDroute = randopk(Droute, ClientPreferences);
    end

end


function [ newDroute ] = op4k( Droute, row1, row2, cols1, cols2, ClientPreferences )
%op4 Operation 4/5: Interroute switching.
%   Given a descriptive route and two rows, exchange the sequence at
%   (row1, cols1) with the one at (row2, cols2) (given that this doesn't
%   cause newDroute to be wider than Droute (width remains p-1)).
%   Example:
%   A = [ 11 13 15 17 19 ; 8 8 8 8 8 ]
%   op4(A, 1, 2, 1:2, 4:5)
%   ans = [ 8 8 15 17 19 ; 8 8 8 11 13 ]

% Obtain the relevant rows and p:
relrow1 = Droute(row1, :);
relrow2 = Droute(row2, :);

% Extract the relevant sections:
sec1 = relrow1(cols1);
sec2 = relrow2(cols2);
relrow1(cols1) = [];
relrow2(cols2) = [];

% Check if this operation would not put clients on an unwanted day:
daysok = 1;
for i = 1:length(sec1)
    if (~ismember(sec1(i), ClientPreferences(row2, :)))
        daysok = 0;
    end
end
for i = 1:length(sec2)
    if (~ismember(sec2(i), ClientPreferences(row1, :)))
        daysok = 0;
    end
end

% Only continue if the new days are indeed okay. Otherwise, return current
% route.
newDroute = Droute;

if (daysok)
    % Place each extracted section at the start of the other trimmed row:
    relrow1 = [sec2 relrow1];
    relrow2 = [sec1 relrow2];

    % If relrows are too short or long, fix this by adding or removing zeros:
    properwidth = size(Droute, 2);
    difference1 = length(relrow1) - properwidth;
    difference2 = length(relrow2) - properwidth;
    if (difference1 > 0)
        relrow1 = relrow1(1:properwidth);
    elseif (difference1 < 0)
        relrow1 = [relrow1 zeros(1, abs(difference1))];
    end
    if (difference2 > 0)
        relrow2 = relrow2(1:properwidth);
    elseif (difference2 < 0)
        relrow2 = [relrow2 zeros(1, abs(difference2))];
    end

    % Use operation 3 (intraroute insertion) to get them to the right place:
    relrow1 = op3(relrow1, 1, 1:length(sec2), cols1(1));
    relrow2 = op3(relrow2, 1, 1:length(sec1), cols2(1));
```

```
    % Clone newDroute out of Droute but replace the relevant rows with the new:
    %newDroute = Droute;
    newDroute(row1, :) = relrow1;
    newDroute(row2, :) = relrow2;
end

end


function [ newDroute ] = op5k( Droute, row1, row2, cols1, ClientPreferences)
%op4 Operation 5/5: Interroute transfer.
%   Given a descriptive route and two rows, remove the sequence at
%   (row1, cols1) and staple it to the end of row2(given that this doesn't
%   cause newDroute to be wider than Droute (width remains p-1)).
%   Example:
%   A = [ 11 13 15 17 19 ; 8 9 0 0 0 ]
%   op5(A, 1, 2, 1:2)
%   ans = [ 15 17 19 0 0 ; 8 9 11 13 0 ]

% If all goes wrong, return Droute:
newDroute = Droute;

% Obtain the relevant rows and p:
relrow1 = Droute(row1, :);
relrow2 = Droute(row2, :);
p = size(Droute, 2) + 1;

% Check if a valid regions of columns is being extracted from row1:
size1 = length(unique(relrow1));
if (ismember(0, relrow1))
    size1 = size1 - 1;
end
ok1 = (cols1(1) >= 1 && cols1(end) <= size1 && size1 - length(cols1) >= 1);

% Count the non-zero elements in row2, to see if stapling is allowed:
size2 = length(unique(relrow2));
if (ismember(0, relrow2))
    size2 = size2 - 1;
end
ok2 = (size2 + length(cols1) <= p-1);

% Check if this operation would not place clients on unwanted days:
ok3 = 1;
if (ok1)
    tempsec = relrow1(cols1);
    for i = 1:length(tempsec)
        if (~ismember(tempsec(i), ClientPreferences(row2, :)))
            ok3 = 0;
        end
    end
end

% If these sizes are legal, perform the operation:
if (ok1 && ok2 && ok3)

    % Extract the nonzeros that remain in row1, the nonzeros that move to
    % row2 and the nonzeros of row2.
    sec1 = relrow1(cols1);
    sec2 = relrow1(1:size1);
    sec2(cols1) = [];
    sec3 = relrow2(1:size2);

    % Form newDroute by replacing the changed rows:
    %newDroute = Droute;
    newrow1 = [sec2, zeros(1, (p-1)-length(sec2))];
    newrow2 = [sec3, sec1, zeros(1, (p-1)-length(sec3)-length(sec1))];
    newDroute(row1,:) = newrow1;
    newDroute(row2,:) = newrow2;
end

if (~(ok1 && ok2))
```

```
    disp('Warning: illegal op5 request. op5 not performed.');
end

end

function pair = randomlegalpair( legalvector )
%legalvector For use in randopk. Returns 12 vector (e.g. [4 6]).
%   If legalvector = [0 1 0 1 1 1], its sections of ones are
%   [2 2], [4 4], [4 5], [4 6], [5 5], [5 6], [6 6].
%   Randomly select one of these with equal odds.

% Initialize a sufficiently large storage matrix for all legal pairs.
AllPairs = zeros(sum(legalvector)^2, 2);
paircounter = 0;

% For each pair, check if it's legal.
for i = 1:length(legalvector)
    for j = 1:length(legalvector)
        if (i <= j)
            % Check if columns i to j consist completely of ones:
            if (~ismember(0, legalvector(i:j)))
                paircounter = paircounter + 1;
                AllPairs(paircounter, :) = [i j];
            end
        end
    end
end

% Now return a randomly picked one:
row = randi(paircounter);
pair = AllPairs(row, :);

end
```

Appendix H

Code for neural networks

```
function [ route, cost ] = NNexperi( Costs, m, r, lambdas, u0, dt )
%NeuralNetwork Attempt to solve the mTSP with a neural network.
%   Detailed explanation goes here
%   Example:  [x,y] = NNexperi(trivial(9, 2, 5, 0), 2, 4, 1.5*[30 30 10 10 10], 0.02, 0.0001)

% Abbreviate variables:
C = Costs;
n = size(C, 1);
S = m + n - 1;
%treshold = 0.0001;

% Make a modified distance matrix d by setting C's diagonal to zero and
% adding m-1 fictional cities at the location of city 1:
C(1:n+1:n*n) = 0;
d = [C, repmat(C(:,1), 1, m-1) ; repmat(C(1,:), m-1, 1) 10000*ones(m-1)];

validated = 0;
patiencecounter = 0;
while (~validated)

    % Randomly initialize neural network:
    V = (1/(S))*ones(S);
    u00 = asigmoid(V(1,1), u0);
    U = u00*ones(S) + (0.2*u0*rand(S) - 0.1*u0);
    V = sigmoid(U, u0);
    L = lambdas;

    % Perform gradient descent until the energy stops changing significantly:
    for iteration = 1:2000
        [~, E1, E2, E3, E4, E5] = energy(V, d, L, n, m, r);
        % Determine gradient of U:
        R = zeros(size(U));
        for k = 1:S
            for i = 1:S
                % Determine sump:
                sump = 0;
                for l = 1:S
                    if (l ~= k)
                        nex = i+1;
                        if (nex > S)
                            nex = 1;
                        end
                        prev = i-1;
                        if (prev < 1)
                            prev = S;
                        end
                        sump = sump + d(k,l)*(V(l,nex) + V(l,prev));
                    end
                end
                % Determine sum1:
                sum1 = sum(V(k,:)) - V(k,i);
                % Determine sum2:
```

```
                sum2 = sum(V(:,i)) - V(k,i);
                % Determine sum3:
                sum3 = sum(sum(V)) - (n+m-1);
                % Determine sum4:
                sum4 = 0;
                for l = n+1:S
                    subsum = 0;
                    nex = i;
                    prev = i;
                    for s = 1:r
                        nex = nex + 1;
                        if (nex > S)
                            nex = 1;
                        end
                        prev = prev - 1;
                        if (prev < 1)
                            prev = S;
                        end
                        subsum = subsum + V(l,nex) + V(l,prev);
                    end
                    sum4 = sum4 + subsum;
                end
                % Determine sum5:
                sum5 = sum(sum(V(n+1:end, :))) - (m-1);

                R(k,i) = sump + L*[sum1; sum2; sum3; sum4; sum5];
            end
        end
        % Perform Euler Forward:
        U = U - dt*(U+R);
        L = L + dt*[E1 E2 E3 E4 E5];
        V = sigmoid(U, u0);
    end

    % Check if this network is even valid:
    if (~validNN(V))
        %disp('Found neural network detected as invalid: ');
        %disp(V);
        %again = input('Retry? (1/0) ');
        patiencecounter = patiencecounter + 1;
        if (patiencecounter < 5)
            again = 1;
        else
            again = 0;
            patiencecounter = 0;
        end
    end
    if (validNN(V) || ~again)
        validated = 1;
    end
end

% When while-loop has terminated, check properness.
if (validNN(V))
    route = round(V + 0.48);
    cost = energy(route, d, L, n, m, r);
else
    route = [];
    cost = 0;
end

end


function [ Ep, E1, E2, E3, E4, E5 ] = energy( Activations, ModifiedDistances, lambdas, n, m, r )
%energy The energy in the neural network. Cost to be minimized.
%   Detailed explanation goes here

% Declare variables:
V = Activations;
d = ModifiedDistances;
```

```
S = n + m - 1;
l1 = lambdas(1);
l2 = lambdas(2);
l3 = lambdas(3);
l4 = lambdas(4);
l5 = lambdas(5);

% Ep
Ep = 0;
for k = 2:n
    for l = 1:S
        if (l ~= k)
            for i = 1:S
                nex = i + 1;
                if (nex > S)
                    nex = 1;
                end
                prev = i - 1;
                if (prev < 1)
                    prev = S;
                end
                Ep = Ep + d(k,l)*V(k,i)*(V(l,nex) + V(l,prev));
            end
        end
    end
end
Ep = 0.5* Ep;

%E1
E1 = 0;
for k = 1:S
    for i = 1:S
        for j = 1:S
            if (i ~= j)
                E1 = E1 + V(k,i)*V(k,j);
            end
        end
    end
end
E1 = 0.5*l1*E1;

%E2
E2 = 0;
for i = 1:S
    for k = 1:S
        for l = 1:S
            if (l ~= k)
                E2 = E2 + V(k,i)*V(l,i);
            end
        end
    end
end
E2 = 0.5*l2*E2;

%E3
E3 = 0.5*l3*(sum(sum(V)) - S)^2;

%E4
E4 = 0;
for k = n+1:S
    for l = n+1:S
        if (l ~= k)
            for i = 1:S
                insum = 0;
                nex = i;
                prev = i;
                for s = 1:r
                    nex = nex + 1;
                    if (nex > S)
                        nex = 1;
```

```
                    end
                    prev = prev - 1;
                    if (prev < 1)
                        prev = S;
                    end
                    insum = insum + V(l, nex) + V(l, prev);
                end
                E4 = E4 + V(k,i)*insum;
            end
        end
    end
end
E4 = 0.5*l4*E4;

%E5
E5 = 0.5*l5*(sum(sum(V(n+1:S,:))) - (m-1))^2;

end

function [ Activations ] = sigmoid( Voltages, u0 )
%UNTITLED6 Summary of this function goes here
%   Detailed explanation goes here

Activations = 0.5*(1 + tanh(Voltages/u0));

end

function [ Voltages ] = asigmoid( Activations, u0 )
%UNTITLED7 Summary of this function goes here
%   Detailed explanation goes here

Voltages = u0*atanh(2*Activations - 1);

end

function [ NeuralNetwork ] = initializeNN( n, m, u0, u00 )
%initializeNN Initialize a neural network randomly around u00.
%   Detailed explanation goes here

NeuralNetwork = u00 + 0.2*u0*rand(n+m-1) - 0.1*u0*ones(n+m-1);

NeuralNetwork(1,:) = [1 zeros(1, n+m-2)];
NeuralNetwork(:,1) = [1; zeros(n+m-2, 1)];

end

function [ bool ] = validNN( Activations )
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here

nn = round(Activations + 0.48);
n = size(nn, 1);

% Check if each row has exactly one 1:
check1 = min(sum(nn, 2) == ones(n, 1));

% Check if each column has exactly one 1:
check2 = min(sum(nn, 1) == ones(1, n));

bool = min(check1, check2);

end
```

Appendix I

Code for performing tests

```
function [ CostMatrix ] = CostFromCSV3( costCSV, addressVector, rowskip, colskip )
%CostFromCSV2 Gets a cost matrix from the csv, given a subset of addresses.
%   addressVector may contain an entry n times if the corresponding address
%   is to be visited n times (or for some other reason needs n times the
%   regular time to be handled).

% Read the csv into matrix M:
M = dlmread(costCSV, ';', rowskip, colskip);
n = length(addressVector);

% Fill in CostMatrix by collecting data from M:
CostMatrix = zeros(n);
for i = 1:n
    for j = 1:n
        row = addressVector(i);
        col = addressVector(j);
        CostMatrix(i,j) = M(row, col);
    end
end

end

function [ Subcosts ] = randomsubsetcosts( Costs, depotindex, n )
%randomsubsetcosts Create a random cost submatrix of Costs, size n.
%   Detailed explanation goes here

N = size(Costs, 1);
P = randperm(N);

% Construct the list of size n of randomly selected nodes. The first node
% is the given depot.
p = zeros(1, n);
p(1) = depotindex;
indexP = 0;
indexp = 1;
while (ismember(0, p))
    indexP = indexP + 1;
    x = P(indexP);
    if (~ismember(x, p))
        indexp = indexp + 1;
        p(indexp) = x;
    end
end

% Then, fill Subcosts with the right entries of Costs.
Subcosts = zeros(n);
for i = 1:n
    for j = 1:n
        nodei = p(i);
        nodej = p(j);
        Subcosts(i, j) = Costs(nodei, nodej);
    end
```

```
end

end


% This script performs all tests.
% Obtain cost matrix from CSV, reading the depot row 13 first:
C = CostFromCSV3('4dFLATdec2.csv', [13 1:12 14:69], 1, 1);
depotindex = 1;
m = 2;

% Run Test 1 with m = 2, n = {7, 9, 11} and two values of p for each n.
% Costs will by definition be equal and routes similar: only record the
% different computation times.

[ ~, ~, ~, T1ex7_4, ~, ~, T1de7_4, ~, ~, T1wi7_4, ~, ~, T1sw7_4, ~, ~, ...
    T1ilp7_4 ] = Test1( C, 1, 7, m, 4 );

[ ~, ~, ~, T1ex7_5, ~, ~, T1de7_5, ~, ~, T1wi7_5, ~, ~, T1sw7_5, ~, ~, ...
    T1ilp7_5 ] = Test1( C, 1, 7, m, 5 );

[ ~, ~, ~, T1ex9_5, ~, ~, T1de9_5, ~, ~, T1wi9_5, ~, ~, T1sw9_5, ~, ~, ...
    T1ilp9_5 ] = Test1( C, 1, 9, m, 5 );

[ ~, ~, ~, T1ex9_7, ~, ~, T1de9_7, ~, ~, T1wi9_7, ~, ~, T1sw9_7, ~, ~, ...
    T1ilp9_7 ] = Test1( C, 1, 9, m, 7 );

% Run Test 2 with m = 2, n = {5, 10, 15, 20} and p minimal:

[ ~, ~, ~, ~, T2sw5, ~, ~, T2su5, ~, ~, T2ss5, ~, ~, T2sb5, ~, ~, ...
T2mw5, ~, ~, T2ms5 ] = Test2( C, 1, 5, m, 3 );

[ ~, ~, ~, ~, T2sw10, ~, ~, T2su10, ~, ~, T2ss10, ~, ~, T2sb10, ~, ~, ...
T2mw10, ~, ~, T2ms10 ] = Test2( C, 1, 10, m, 6 );

[ ~, ~, ~, ~, T2sw15, ~, ~, T2su15, ~, ~, T2ss15, ~, ~, T2sb15, ~, ~, ...
T2mw15, ~, ~, T2ms15 ] = Test2( C, 1, 15, m, 8 );

% Run Test 3 with m = 2, n = {10, 20, 30}, p minimal (floor(n/2) + 1)
% and R maximal (ceil(n/2) - 1). Run once with 'quick parameters', once
% with 'medium parameters' and once with 'slow parameters'. Record costs
% and computation times.
[ ~, ~, C3silp_10q, T3silp_10q, ~, C3sgree_10q, T3sgree_10q, ...
    ~, C3ssim_10q, T3ssim_10q, ~, C3sneu_10q, T3sneu_10q] = ...
    Test3( C, 1, 10, 2, 6, 20, 20, 15, 0.01, 4, 0.02, 0.00001, ...
    [200 200 500 10 10], 1, 1 );

[ ~, ~, C3silp_10m, ~, ~, C3sgree_10m, T3sgree_10m, ...
    ~, C3ssim_10m, T3ssim_10m, ~, C3sneu_10m, T3sneu_10m] = ...
    Test3( C, 1, 10, 2, 6, 50, 50, 20, 0.001, 4, 0.02, 0.00001, ...
    [150 150 375 10 10], 1, 1 );

[ ~, ~, C3silp_10s, ~, ~, C3sgree_10s, T3sgree_10s, ...
    ~, C3ssim_10s, T3ssim_10s, ~, C3sneu_10s, T3sneu_10s] = ...
    Test3( C, 1, 10, 2, 6, 100, 100, 25, 0.0001, 4, 0.02, 0.00001, ...
    [100 100 250 10 10], 1, 1 );

[ ~, ~, C3silp_20q, T3silp_20q, ~, C3sgree_20q, T3sgree_20q, ...
    ~, C3ssim_20q, T3ssim_20q, ~, ~, ~] = ...
    Test3( C, 1, 20, 2, 11, 20, 20, 15, 0.01, 9, 0.02, 0.00001, ...
    [200 200 500 10 10], 1, 0 );

[ ~, ~, C3silp_20m, ~, ~, C3sgree_20m, T3sgree_20m, ...
    ~, C3ssim_20m, T3ssim_20m, ~, ~, ~] = ...
    Test3( C, 1, 20, 2, 11, 50, 50, 20, 0.001, 9, 0.02, 0.00001, ...
    [150 150 375 10 10], 1, 0 );

[ ~, ~, C3silp_20s, ~, ~, C3sgree_20s, T3sgree_20s, ...
    ~, C3ssim_20s, T3ssim_20s, ~, ~, ~] = ...
    Test3( C, 1, 20, 2, 11, 100, 100, 25, 0.0001, 9, 0.02, 0.00001, ...
    [100 100 250 10 10], 1, 0 );
```

```
[ ~, ~, C3silp_30q, T3silp_30q, ~, C3sgree_30q, T3sgree_30q, ...
    ~, C3ssim_30q, T3ssim_30q, ~, ~, ~] = ...
    Test3( C, 1, 30, 2, 16, 20, 20, 15, 0.01, 14, 0.02, 0.00001, ...
    [200 200 500 10 10], 1, 0 );

[ ~, ~, CSilp_30m, TSilp_30q, ~, C3sgree_30m, T3sgree_30m, ...
    ~, C3ssim_30m, T3ssim_30m, ~, ~, ~] = ...
    Test3( C, 1, 30, 2, 16, 50, 50, 20, 0.001, 14, 0.02, 0.00001, ...
    [150 150 375 10 10], 1, 0 );

[ ~, ~, C3silp_30s, TS3silp_30s, ~, C3sgree_30s, T3sgree_30s, ...
    ~, C3ssim_30s, T3ssim_30s, ~, ~, ~] = ...
    Test3( C, 1, 30, 2, 16, 100, 100, 25, 0.0001, 14, 0.02, 0.00001, ...
    [100 100 250 10 10], 1, 0 );

% Run test 4a to solve the mTSP over the complete area:
[ R4a, C4a, T4a ] = Test4a(C, 2, 40);

% Run test 4b to see how well simulated annealing compares:
[~, ~, C4rs, T4rs] = Test4b( C, 2, 40, 150, 30, 0.00001 );

% Run test 5 to compare methods in the modified problem:
[ ~, ~, ~, C5milp_10, T5milp_10, ~, C5mgree_10, T5mgree_10, ~, C5msim_10, ...
    T5msim_10] = Test5( C, 1, 10, 2, 6, 50, 50, 15, 0.001 );

[ ~, ~, ~, C5milp_15, T5milp_15, ~, C5mgree_15, T5mgree_15, ~, C5msim_15, ...
    T5msim_15] = Test5( C, 1, 15, 2, 8, 50, 50, 15, 0.001 );

[ ~, ~, ~, C5milp_20, T5milp_20, ~, C5mgree_20, T5mgree_20, ~, C5msim_20, ...
    T5msim_20] = Test5( C, 1, 20, 2, 11, 50, 50, 15, 0.001 );

% Run test 6 to solve the fitting problem:
[ M6ilp, M6siman, C6ilp, C6siman ] = Test6( C );


function [ Subcosts1, R1ex, C1ex, T1ex, R1de, C1de, T1de, ...
    R1wi, C1wi, T1wi, R1sw, C1sw, T1sw, R1ilp, C1ilp, T1ilp ] = ...
    Test1( Costs, depotindex, n, m, p )
%Test1 Performs the experiment from Section 3.1.
%   Compares, for the given values of n and p, how well a number of exact
%   methods work. Fifty simulations are created from random subsets of
%   Costs of size n. For each simulation i, first exhaustive search is run:
%   the found solution is stored as cell R1ex{i}, the found cost as double
%   C1ex(i) and the required computation time as double T1ex(i). Doing the
%   same for depth-first branch and bound, width-first branch and bound,
%   switch over branch and bound and intlinprog, the data is collected and
%   returned by the function to be further examined and processed.

% Example usage: given a square cost matrix C of size 1111, of which row
% and column 1 represent the depot, to get only the different computation
% times for n = 7 (and m = 2, p = 4):
%   [ ~, ~, ~, T1ex7, ~, ~, T1de7, ~, ~, T1wi7, ~, ~, T1sw7, ~, ~, T1ilp7 ]
%   = Test1( C, 1, 7, 2, 4 )

runs = 50;
if (n > 9)
    runs = 3;
end

% Initialize storage vectors:
Subcosts1 = cell(1, runs);
R1ex = cell(1, runs);
R1de = cell(1, runs);
R1wi = cell(1, runs);
R1sw = cell(1, runs);
R1ilp = cell(1, runs);
C1ex = zeros(1, runs);
C1de = zeros(1, runs);
C1wi = zeros(1, runs);
C1sw = zeros(1, runs);
```

```
C1ilp = zeros(1, runs);
T1ex = zeros(1, runs);
T1de = zeros(1, runs);
T1wi = zeros(1, runs);
T1sw = zeros(1, runs);
T1ilp = zeros(1, runs);

for i = 1:runs
    disp(i);
    disp([n, m, p]);
    % Get random subnetwork (respecting the depotindex):
    CR = randomsubsetcosts(Costs, depotindex, n) + 100*eye(n);
    Subcosts1{i} = CR;

    t0 = cputime;
    [r, c] = exhaustive(CR, m, p);
    t1 = cputime - t0;
    R1ex{i} = r;
    C1ex(i) = c;
    T1ex(i) = t1;

    t0 = cputime;
    [r, c] = BnBmode(CR, m, p, 'depth');
    t1 = cputime - t0;
    R1de{i} = r;
    C1de(i) = c;
    T1de(i) = t1;

    t0 = cputime;
    [r, c] = BnBmode(CR, m, p, 'width');
    t1 = cputime - t0;
    R1wi{i} = r;
    C1wi(i) = c;
    T1wi(i) = t1;

    t0 = cputime;
    [r, c] = BnBmode(CR, m, p, 'switch');
    t1 = cputime - t0;
    R1sw{i} = r;
    C1sw(i) = c;
    T1sw(i) = t1;

    t0 = cputime;
    [r, c] = ilp2015b(CR, m, p, 1);
    t1 = cputime - t0;
    R1ilp{i} = r;
    C1ilp(i) = c;
    T1ilp(i) = t1;
end

disp('Test 1 completed.');

end


function [ Subcosts2, Availab2, R2sw, C2sw, T2sw, R2su, C2su, T2su, R2ss, ...
    C2ss, T2ss, R2sb, C2sb, T2sb, R2mw, C2mw, T2mw, R2ms, C2ms, T2ms ] = ...
    Test2( Costs, depotindex, n, m, p )
%Test2 Performs the experiment from Section 3.2.
%   Compares, for the given values of n and p, how much additional
%   constraints contribute to ILP formulations. Fifty simulations are
%   created from random subsets of Costs of size n. For each simulation i,
%   first simple intlinprog is run with only subtour elimination constraint:
%   the found solution is stored as cell R2sw{i}, the found cost as double
%   C2sw(i) and the required computation time as double T2sw(i). Doing the
%   same for simple with upper bound on potentials, simple with restriction
%   of symmetry, simple with both, modified with only subtour elimination
%   and modified with single tour allocation, the data is collected and
%   returned by the function to be further examined and processed.

% Example usage: given a square cost matrix C of size 1111, of which row
```

```
% and column 1 represent the depot, to get only the different computation
% times for n = 7 (and m = 2, p = 4):
%   [ ~, ~, ~, ~, T2sw7, ~, ~, T2su7, ~, ~, T2ss7, ~, ~, T2sb7, ~, ~, ...
%   T2mw7, ~, ~, T2ms7 ] = Test2( C, 1, 7, 2, 4 )

runs = 50;

% Initialize storage vectors:
Subcosts2 = cell(1, runs);
Availab2 = cell(1, runs);
R2sw = cell(1, runs);
R2su = cell(1, runs);
R2ss = cell(1, runs);
R2sb = cell(1, runs);
R2mw = cell(1, runs);
R2ms = cell(1, runs);
C2sw = zeros(1, runs);
C2su = zeros(1, runs);
C2ss = zeros(1, runs);
C2sb = zeros(1, runs);
C2mw = zeros(1, runs);
C2ms = zeros(1, runs);
T2sw = zeros(1, runs);
T2su = zeros(1, runs);
T2ss = zeros(1, runs);
T2sb = zeros(1, runs);
T2mw = zeros(1, runs);
T2ms = zeros(1, runs);

for i = 1:runs
    disp(i);
    disp([n, m, p]);
    % Get random subnetwork (respecting the depotindex):
    CR = randomsubsetcosts(Costs, depotindex, n) + 100*eye(n);
    Subcosts2{i} = CR;
    % Get random customer availabilty, assuming m = 2:
    CA = -1*ones(m, n);
    CA(:, 1) = 1;
    for j = 2:n
        r = rand(1);
        if (r <= 0.60)
            CA(:, j) = j;
        end
        if (r > 0.60 && r <= 0.80)
            CA(1, j) = j;
        end
        if (r > 0.80)
            CA(2, j) = j;
        end
    end
    if (m > 2)
        for j = 3:m
            CA(m, :) = 1:n;
        end
    end
    Availab2{i} = CA;

    t0 = cputime;
    [r, c] = ilp2015b(CR, m, p, 1);
    t1 = cputime - t0;
    R2sw{i} = r;
    C2sw(i) = c;
    T2sw(i) = t1;

    t0 = cputime;
    [r, c] = ilp2015b(CR, m, p, [1 2]);
    t1 = cputime - t0;
    R2su{i} = r;
    C2su(i) = c;
    T2su(i) = t1;
```

```
    t0 = cputime;
    [r, c] = ilp2015b(CR, m, p, [1 3]);
    t1 = cputime - t0;
    R2ss{i} = r;
    C2ss(i) = c;
    T2ss(i) = t1;

    t0 = cputime;
    [r, c] = ilp2015b(CR, m, p, [1 2 3]);
    t1 = cputime - t0;
    R2sb{i} = r;
    C2sb(i) = c;
    T2sb(i) = t1;

    t0 = cputime;
    [r, c] = ilp2015k(CR, m, p, 1, CA);
    t1 = cputime - t0;
    R2mw{i} = r;
    C2mw(i) = c;
    T2mw(i) = t1;

    t0 = cputime;
    [r, c] = ilp2015k(CR, m, p, [1 2], CA);
    t1 = cputime - t0;
    R2ms{i} = r;
    C2ms(i) = c;
    T2ms(i) = t1;
end

disp('Test 2 completed.');

end


function [ Subcosts3, R3silp, C3silp, T3silp, R3sgree, ...
    C3sgree, T3sgree, R3ssim, C3ssim, T3ssim, R3sneu, C3sneu, T3sneu] ...
    = Test3( Costs, depotindex, n, m, p, tauG, tauS, T0, Tend, R, u0, dt, lambdas, doILP, doNN )
%Test3 Performs the experiment from Section 3.3.
%   Compares, for the simple problem, how well four different
%   implementations perform for random subnetworks. Fifty simulations are
%   created from random subsets of Costs of size n. For each simulation i,
%   first simple intlinprog is run: the found solution is stored as cell
%   R3silp{i}, the found cost as double C3silp(i) and the required
%   computation time as double T3silp(i). Doing the same for simple greedy
%   search, simple simulated annealing and simple neural networks, the data
%   is collected and returned by the function to be further examined and
%   processed.

% Example usage: to get only the different best costs and computation
% times for n = 10 (and m = 2, p = 6):
%   [ ~, ~, C3silp, T3silp, ~, C3sgree, T3sgree, ~, C3ssim, T3ssim, ~, ...
%   C3sneu, T3sneu] = Test3( C, 1, 10, 2, 6, 50, 50, 15, 0.001, 1, 0.02, ...
%   0.00001, [20 20 50 10 10], 1, 1 )

runs = 50;

% Initialize storage vectors:
Subcosts3 = cell(1, runs);
R3silp = cell(1, runs);
R3sgree = cell(1, runs);
R3ssim = cell(1, runs);
R3sneu = cell(1, runs);
C3silp = zeros(1, runs);
C3sgree = zeros(1, runs);
C3ssim = zeros(1, runs);
C3sneu = zeros(1, runs);
T3silp = zeros(1, runs);
T3sgree = zeros(1, runs);
T3ssim = zeros(1, runs);
T3sneu = zeros(1, runs);
```

```
for i = 1:runs
    disp(i);
    disp(n);
    % Get random subnetwork (respecting the depotindex):
    CR = randomsubsetcosts(Costs, depotindex, n) + 100*eye(n);
    Subcosts3{i} = CR;

    if (doILP)
        t0 = cputime;
        if (n > 20)
            [r, c] = ilp2015bLIM(CR, m, p, [1 3]);
        else
            [r, c] = ilp2015b(CR, m, p, [1 3]);
        end
        t1 = cputime - t0;
        R3silp{i} = r;
        C3silp(i) = c;
        T3silp(i) = t1;
    end

    t0 = cputime;
    [r, c] = GreedySearch(m, p, CR, 1, tauG);
    t1 = cputime - t0;
    R3sgree{i} = r;
    C3sgree(i) = c;
    T3sgree(i) = t1;

    t0 = cputime;
    [r, c] = siman6(m, p, CR, 1, tauS, T0, Tend);
    t1 = cputime - t0;
    R3ssim{i} = r;
    C3ssim(i) = c;
    T3ssim(i) = t1;

    if (doNN)
        t0 = cputime;
        [r, c] = NNexperi(CR, 2, R, lambdas, u0, dt);
        t1 = cputime - t0;
        R3sneu{i} = r;
        C3sneu(i) = c;
        T3sneu(i) = t1;
    end
end

disp('Test 3 completed.');

end

function [ R4a, C4a, T4a ] = Test4a( Costs, m, p )
%Test4a Solves the simple mTSP on Costs with intlinprog.
%   Example usage: [r, c, t] = Test4a(Costs, 2, 40);

t0 = cputime;
[R4a, C4a] = ilp2015b(Costs, m, p, [1 3]);
t1 = cputime - t0;
T4a = t1;

end

function [ Subcosts4, R4ssim, C4ssim, T4ssim] ...
    = Test4b( Costs, m, p, tauS, T0, Tend)
%Test4b Performs the second experiment from Section 3.4.
%   Attempts to solve the simple problem with simulated annealing so it can
%   be compared to intlinprog's performance.

% Example usage:
%   [~, ~, C4, T4] = Test4b( Costs, 2, 40, 100, 15, 0.001 );

runs = 50;
```

```
% Initialize storage vectors:
Subcosts4 = cell(1, runs);
%R4sgree = cell(1, runs);
R4ssim = cell(1, runs);
%C4sgree = zeros(1, runs);
C4ssim = zeros(1, runs);
%T4sgree = zeros(1, runs);
T4ssim = zeros(1, runs);

for i = 1:runs
    disp(i);

%     t0 = cputime;
%     [r, c] = GreedySearch(m, p, CR, 1, tauG);
%     t1 = cputime - t0;
%     R4sgree{i} = r;
%     C4sgree(i) = c;
%     T4sgree(i) = t1;

    t0 = cputime;
    [r, c] = siman6(m, p, Costs, 1, tauS, T0, Tend);
    t1 = cputime - t0;
    R4ssim{i} = r;
    C4ssim(i) = c;
    T4ssim(i) = t1;
end

disp('Test 4 completed.');

end


function [ Subcosts5, Availab5, R5milp, C5milp, T5milp, R5mgree, ...
    C5mgree, T5mgree, R5msim, C5msim, T5msim] ...
    = Test5( Costs, depotindex, n, m, p, tauG, tauS, T0, Tend )
%Test3 Performs the experiment from Section 3.3.
%   Compares, for the simple problem, how well four different
%   implementations perform for random subnetworks. Fifty simulations are
%   created from random subsets of Costs of size n. For each simulation i,
%   first simple intlinprog is run: the found solution is stored as cell
%   R3silp{i}, the found cost as double C3silp(i) and the required
%   computation time as double T3silp(i). Doing the same for simple greedy
%   search, simple simulated annealing and simple neural networks, the data
%   is collected and returned by the function to be further examined and
%   processed.

% Example usage: to get only the different best costs and computation
% times for n = 10 (and m = 2, p = 6):
%   [ ~, ~, ~, C5milp, T5milp, ~, C5mgree, T5mgree, ~, C5msim, T5msim] ...
%   = Test5( C, 1, 10, 2, 6, 50, 50, 15, 0.001 )

runs = 50;

% Initialize storage vectors:
Subcosts5 = cell(1, runs);
Availab5 = cell(1, runs);
R5milp = cell(1, runs);
R5mgree = cell(1, runs);
R5msim = cell(1, runs);
C5milp = zeros(1, runs);
C5mgree = zeros(1, runs);
C5msim = zeros(1, runs);
T5milp = zeros(1, runs);
T5mgree = zeros(1, runs);
T5msim = zeros(1, runs);

for i = 1:runs
    disp(i);
    % Get random subnetwork (respecting the depotindex):
    CR = randomsubsetcosts(Costs, depotindex, n) + 100*eye(n);
```

```
    Subcosts5{i} = CR;
    % Get random customer availabilty, assuming m = 2:
    CA = -1*ones(m, n);
    CA(:, 1) = 1;
    for j = 2:n
        r = rand(1);
        if (r <= 0.60)
            CA(:, j) = j;
        end
        if (r > 0.60 && r <= 0.80)
            CA(1, j) = j;
        end
        if (r > 0.80)
            CA(2, j) = j;
        end
    end
    if (m > 2)
        for j = 3:m
            CA(m, :) = 1:n;
        end
    end
    Availab5{i} = CA;

    t0 = cputime;
    [r, c] = ilp2015kLIM(CR, m, p, 1, CA);
    t1 = cputime - t0;
    R5milp{i} = r;
    C5milp(i) = c;
    T5milp(i) = t1;

    t0 = cputime;
    [r, c] = greedyk(m, p, CR, 1, tauG, CA);
    t1 = cputime - t0;
    R5mgree{i} = r;
    C5mgree(i) = c;
    T5mgree(i) = t1;

    t0 = cputime;
    [r, c] = siman7k(m, p, CR, 1, tauS, 0.95, T0, Tend, CA);
    t1 = cputime - t0;
    R5msim{i} = r;
    C5msim(i) = c;
    T5msim(i) = t1;
end

disp('Test 5 completed.');

end


function [ M6ilp, M6siman, C6ilp, C6siman ] = Test6( Costs )
%Test6 Solves fitting problem with intlinprog and simulated annealing.
%   m = 2 is assumed.
%   Example usage: [a, b, c, d] = Test6( Costs, 2 );

C = Costs;
n = 6;
stop = 0;
M6ilp = [];
M6siman = [];
C6ilp = [];
C6siman = [];
ilpAccepted = [];

while(~stop)
    M6ilp = [M6ilp ; zeros(1, 5)];
    M6siman = [M6siman ; zeros(1, 5)];
    C6ilp = [C6ilp ; zeros(1, 5)];
    C6siman = [C6siman ; zeros(1, 5)];
    ilpAccepted = [ilpAccepted ; zeros(1, 5)];
    n = n + 1;
```

```
disp(n);
p = ceil(n/2) + 1;

for i = 1:5
    % Get random submatrix:
    CR = randomsubsetcosts(C, 13, n) + 100*eye(n);
    % Get random availabilities:
    CA = -1*ones(2, n);
    CA(:, 1) = 1;
    for j = 2:n
        r = rand(1);
        if (r <= 0.60)
            CA(:, j) = j;
        end
        if (r > 0.60 && r <= 0.80)
            CA(1, j) = j;
        end
        if (r > 0.80)
            CA(2, j) = j;
        end
    end

    [Rilp, ~] = ilp2015kC(CR, 2, p, 1, CA);
    if (size(Rilp, 1) > 0)
        route1 = Rilp(1,:);
        n1 = length(unique(route1));
        if (ismember(0, route1))
            n1 = n1 - 1;
        end
        C1 = costrouteinternal(CR, des2binN(route1, n, 1, p));
        C1p = C1 + 10*n1;
        route2 = Rilp(2,:);
        n2 = length(unique(route2));
        if (ismember(0, route2))
            n2 = n2 - 1;
        end
        C2 = costrouteinternal(CR, des2binN(route2, n, 1, p));
        C2p = C2 + 10*n2;
        [Milp, index] = max([C1p C2p]);

        M6ilp(n-6, i) = Milp;
        temp = [C1 C2];
        C6ilp(n-6, i) = temp(index);
    else
        M6ilp(n-6, i) = Inf;
        C6ilp(n-6, i) = Inf;
    end
    if (Milp <= 240)
        ilpAccepted(n-6, i) = 1;
    end

    [Rsiman, ~] = siman7k(2, p, CR, 1, 50, 0.95, 15, 0.001, CA);
    route3 = Rsiman(1,:);
    n3 = length(unique(route3));
    if (ismember(0, route3))
        n3 = n3 - 1;
    end
    C3 = costrouteinternal(CR, des2binN(route3, n, 1, p));
    C3p = C3 + 10*n3;
    route4 = Rsiman(2,:);
    n4 = length(unique(route4));
    if (ismember(0, route4))
        n4 = n4 - 1;
    end
    C4 = costrouteinternal(CR, des2binN(route4, n, 1, p));
    C4p = C4 + 10*n4;
    [Msiman, index] = max([C3p C4p]);
    temp = [C3 C4];
    M6siman(n-6, i) = Msiman;
    C6siman(n-6, i) = temp(index);
```

```
    end

    if (sum(ilpAccepted(n-6,:)) < 4)
        stop = 1;
    end
end

end
```