MAGISTERARBEIT / MASTER'S THESIS

Titel der Magisterarbeit / Title of the Master's Thesis

An Enumerative Approach to the Solution of a Multi-Period Orienteering Problem

verfasst von / submitted by

Alexander Ruth, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Magister der Sozial- und Wirtschaftswissenschaften (Mag. rer. soc. oec.)

Wien, 2017 / Vienna 2017

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 951
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Statistik
Betreut von / Supervisor:	Univ.-Prof. Mag. Dr. Walter Gutjahr

Erklärung

Hiermit versichere ich,

- dass ich die vorliegende Magisterarbeit selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubter Hilfe bedient habe,
- dass ich dieses Magisterarbeitsthema bisher weder im In- noch im Ausland in irgendeiner Form als Prüfungsarbeit vorgelegt habe
- und dass diese Arbeit mit der vom Begutachter beurteilten Arbeit vollständig übereinstimmt.

Klosterneuburg, am 29.10.2017

Unterschrift: [illegible]

Abstract in English

When dealing with orienteering problems, most people would pick heuristics as their choice. Especially, multi-period orienteering problems with the following characteristics are considered in this thesis. There are no vehicle capacity constraints. The demand grows linear over time, can be accommodated completely upon visit and decays exponentially if not visited. There is a fixed number of time steps and on each time step one tour will be scheduled. The tour length limit is constant without bonus for left over length. There are no time window constraints. Starting and end points are at the depot.
This thesis depicts the construction of a new algorithm which is able to solve multi-period orienteering problems exactly and in an efficient way. This algorithm consists of two parts, generating candidate tours and computing the optimal schedule, respectively. One advantage over standard complete enumeration is the reduction of the number of travelling salesman problems (TSP) that need to be solved. In a stepwise algorithm, many tours are discarded by using an easy-to-compute bound and the solving of linear assignment problems (LAP). Also, subtours of positive TSPs are discarded. This also leaves a lot smaller number to work with for the scheduling problem, which is the second advantage over complete enumeration.
For algorithm tuning multiple options such as the tradeoff between using and not using the bound or between LAPs and TSPs are considered and the best methods are determined by runtime analysis. It turned out that using the bound while using the maximum number of LAPs is the best combination. As a demonstration, a real-world application example with selected provincial capitals of Austria and Germany is profoundly analyzed.

Abstract in German

Üblicherweise werden Orienteering Probleme mithilfe von Heuristiken gelöst. Im Speziellen werden in dieser Magisterarbeit Multiperioden Orienteering Probleme mit nachfolgenden charakteristischen Eigenschaften behandelt. Es gibt keine Kapazitätseinschränkung der Fahrzeuge. Die Nachfrage wächst linear mit der Zeit, kann bei einem Besuch komplett gedeckt werden und sinkt exponentiell falls kein Besuch stattfindet. Es gibt eine vorab festgelegte Anzahl von Zeitpunkten und für jeden Zeitpunkt ist genau eine Tour geplant. Für alle Zeitpunkte ist die Längenbeschränkung der Tour konstant und es gibt keine Rückvergütung falls dieses Limit nicht ausgereizt wird. Es gibt keine Einschränkung der Besuchszeiten der Knoten. Alle Touren starten und enden beim Depot.

Diese Magisterarbeit stellt die Konstruktion eines neuen Algorithmus vor, der dieses spezielle Problem exakt und zeiteffizient lösen kann. Der Algorithmus besteht aus zwei Teilen, wobei zuerst candidate tours ermittelt werden und dann aus diesen der optimale Zeitplan berechnet wird. Ein Vorteil gegenüber vollständiger Enumeration ist die niedrigere Anzahl an Travelling Salesman Problemen (TSP) (deutsch auch "Problem des Handlungsreisenden"), die gelöst werden müssen, da schrittweise durch eine Schranke und das Lösen von Linear Assignment Problemen (LAP) (deutsch auch "Lineares Zuordnungsproblem") einige der zu testenden Touren ausgeschlossen werden. Außerdem werden Subtouren von Touren, die die Nebenbedingungen erfüllen, ausgeschlossen. Daraus ergibt sich die minimale Anzahl von relevanten Touren, was weiters die Laufzeit für die Berechnung des optimalen Zeitplans im Vergleich zur vollständigen Enumeration drastisch verkürzt.

Das Feintuning, welches daraus bestand zu vergleichen, ob es schneller ist die Schranke zu benutzen oder wegzulassen oder wegzulassen und zu entscheiden, ab welchem Zeitpunkt man von LAPs auf TSPs wechseln sollte, stellte sich heraus, dass die optimale Kombination die Verwendung von Schranke und maximaler Anzahl an LAPs ist. Als Demonstration des Algorithmus wird ein Anwendungsbeispiel bearbeitet, dass aus ausgewählten Landeshauptstädten von Österreich und Deutschland besteht, und analysiert.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The classic orienteering problem (OP) was inspired by the orienteering sport. Competitors have to find control points, marked on a map, in a given area and within a fixed amount of time. As most of the time it won't be able to visit all control points before time runs out, in addition to finding the shortest tour, a selection step is needed to determine the subset of control points, which will be visited. It is also possible to assign different positive values (weights) to the control points. In the end the set of visited control points determines the score and the participant with the highest score wins.

Since the first formulation of the orienteering problem (OP) by Tsiligirides (1984) [23], research has mainly focused on finding approximations for different kinds of OPs using methods such as variable neighborhood search (VNS) (Tricoire ea 2010) [22] or ant colony algorithms (ACS, ACO) (Montemanni ea 2009) [16]. In this thesis, an exact algorithm for solving a particular multi-period orienteering problem will be proposed.

Key features of this type of MPOP are

- Demand is a composition of a linear growth over time and an exponential decay for unfulfilled demand.
- There are no vehicular capacity constraints and complete demand can be satisfied upon visit.
- The tour length limit is constant for all time steps.
- All node distances are symmetric.
- Starting and end point of each tour is the depot.

The multi-period orienteering problem (MPOP) is an extension for situations that face OPs repeatedly. They occur in many different situations such as harvesters, field workers, delivery men, garbage truck drivers or tourists who want to optimize their tours. There is a fixed number of repetitions (rounds) of OPs but the repetitions are not identical. Node values decrease if the corresponding node has been visited last round and increase otherwise. While in the classic OP it is sufficient to look at a tour's score, here it is also necessary to evaluate its impact on the node values. Greedy algorithms cannot consider that and should therefore not be used (if the exact solution is desired).

The MPOP is (like all OPs) closely related to the travelling salesman problem (TSP). Every TSP can be formulated as an MPOP by setting the number of rounds to 1 and the constraint level to infinity or sufficiently large, which makes the TSP a special case of the MPOP. In this manner, the MPOP inherits the NP-hardness of the TSP.

To provide a reasonable runtime, the proposed algorithm includes multiple steps to reduce the number of TSPs and enumeration instances. In the first part candidate tours are identified by using statistics of row and column sums as well as solvers for linear assignment problems while eliminating subtours of candidate tours. Knowing the complete list of candidate tours results in a beneficial reduction of problem complexity. In the second part schedules are generated, filtered and then evaluated. The schedule with the highest score is then returned.

A real-world application example, including the nine provincial capitals of Austria and Munich as nodes, serves as a performance display to illustrate the algorithm. Pairwise distances were calculated by using the road distances of the fastest tour. The MPOP is solved for different constraint levels and the results are compared.

2 Problem Description

Before giving a mathematical formulation of the MPOP, this chapter will start with an informal introduction to the topic. Generally, the aim is to find a schedule of tours that is optimal for a given setting. As an illustration example, the problem setting is applied to an agricultural setting. In this case a company owns fields at different locations that grow vegetables. The aim is to optimize the tours of their truck driver, who visits the fields to collect the harvest and transport it to the depot for further processing. On the following pages the different elements of the MPOP and the application to the illustration example will be described.

The basis of the multi-period orienteering problem are the points of interest. In the context of MPOP they will be denoted as nodes. Typically, nodes are real locations but they can largely vary in size. For an international delivery service the nodes will be spread around the whole globe, whereas in the orienteering sport all nodes will be within a few kilometers. For the truck driver, the nodes are the locations of the fields of the company. These should be near enough so he can start from the depot, visit some of them and return to the depot within one working day.

One of these nodes has a special status and is called depot. The depot is the starting and end point of all tours. In the setting of the application example the depot shall be identified by the term depot. This is the place where all the picked-up vegetables are transported to. It is also the parking area for the truck before and after work.

Each pair of nodes is assigned a distance. This distance measures the cost it takes to travel from one node to another. There are several options on how to formulate distances. One possibility is to use length. This can either be the length of the bee line (often not travelable) or the length measured by an underlying infrastructure such as a road network, which is more realistic in most cases. Other options are to specify distances by the time or money it takes to travel from A to B. While time is self-explanatory, money costs can consist of driver wage, fuel, car toll or tariff, among others. For agriculture, there are multiple options, but assume the limiting quantity is length specified by street kilometers.

The total traveled distance is of course not unbounded. There is a tour length limit (also called constraint level later) which has the same unit as the distances. This restriction represents the action scope. People are limited to the number of hours within a working day, by the amount of money they are willing to spend or by the maximum distance one can travel within a day. For the truck driver with distances specified by street kilometers, the limit will probably be bounded by its fuel tank. Say, this truck has a fuel tank of 1200 liters and a gas mileage of 40 liters/km and refueling is done only at the depot, because there is an internal gas station with a low price. Thus, this truck can drive for 300 km before returning to the depot.

Different nodes may be of different importance. So, there is an additional variable that assigns importance coefficients to the nodes. These coefficients are called weights and exist for all nodes except for the depot. Visiting nodes with a high weight is more rewarding than visiting those with a lower weight. Depending on the application, these importance rankings can reflect demand for a product within a city, the number of parcels to deliver within a certain area or in the case of the illustration example the amount of harvest at the specific field. There are no capacity constraints for the truck driver, which means that the loading volume is expected to be sufficiently large to contain the whole harvest. Analogously, delivery services are assumed to be able to load all their parcels into their vehicle and product sellers have a production that covers the full demand. In most cases, the importance is based on demand, profit, reward or another quantity that is closely related to one of them. The amount of harvest, for instance, translates into profit for the agricultural company. Therefore, the measure of importance will from now on be denoted as demand, profit or reward, which will be used synonymously.

The total reward is the target quantity and the main preference that will shape the tour design. It is the benchmark to decide which tour is profitable and which is not. The truck driver prefers a tour that stops at big fields with a lot of harvest to pick up over a tour with stops at very small fields that do not offer a lot of harvest.

Until now, the problem described is only the classic orienteering problem. In addition to this the introduction of time steps leads to the multi-period orienteering problem. The length of one time step can vary from minutes to days or even longer time periods, but typically one time step reflects one day. Say collecting harvest is done from Monday to Friday with one tour per day, in that case there are five time steps of one day each. The solution to this problem will finally offer a schedule with one tour for each time step (day). There will be one tour scheduled for Monday, one for Tuesday, etc. and the next week will again start with the Monday tour. These tours can be different, but they can be identical as well. The specific schedule and its structure depends a lot on the input variables.

In many applications, there is a decrease in reward if the time before the visit is too long. The demand of a specific person for a product might disappear if a company waits to make an offer, because this person might buy at another company. Similarly, in regard to mathematical modeling, is the case of the vegetable transportation. The vegetables are cropped when they are ripe and waiting a few days causes a decay in quality. But waiting can also increase efficiency because driving to that field on the second day lets the truck driver collect the harvest of two days while having the chance to visit other fields on the first day instead. The status of a node tells how profitable the node is right now, in contrast to the weight of a node, which tells how profitable the node is in general. In the model, there are two components that influence the status of a node. The first one is a linear growth determined by the importance (weight) of the node and the second one is an exponential decay. The exponential decay is tunable by a decay factor. On the field, the linear growth would mean there is a constant harvest every day, say one ton of tomatoes. After two days, there are two tons of tomatoes, but half of it is not fresh, so the profit is smaller compared to two tons of fresh tomatoes. The decay factor can be used to declare the ratio of profit for fresh tomatoes and those of the day before.

The introduction of multiple time steps adds a lot of complexity to this problem. It is very easy to evaluate the total reward for a single tour if there are no time steps. However, in the multi-period setting also tours that do not offer a high reward can make sense, for example because they are complementary to a good tour. In the real-world application example in chapter 6 it turns out that using east and west tours alternatingly is optimal in many cases. This clearly is no general proof, but it indicates that tours that complement each other have good long-term properties. Greedy algorithms tend to find tours with good short-term properties and will therefore find good solutions for the classic orienteering problem, but not necessarily for the MPOP. Hence, an exact algorithm has to consider a higher number of tours instead of just the one that performs best in the classic OP. Furthermore, evaluating a schedule takes a lot longer in terms of runtime than evaluating a single tour. This leads to a problem with high runtime complexity, especially because the classic OP is already known to be NP-hard.

There are several applications for the MPOP such as harvest collection, already illustrated before, as well as planning tours for various delivery services ranging from letters or parcels to home heating fuel, garbage collection, traveling salesmen or tourists. These typically use heuristics to deal with this kind of problems, which generally find good but not necessarily optimal solutions. The aim of this thesis is to find a good tradeoff between runtime and solution quality by retaining the optimal solution and providing a runtime that is considerably lower than complete enumeration. The presented algorithm moreover provides a flexible structure with room for adaptations. It is possible to change the underlying model by introducing a minimum/-maximum number of nodes per tour or change the growth/decay model to practically any model with small adjustments in the programming code and without changing the structure of the algorithm.

3 Related Literature

Current research provides a great variety of solution approaches to the classic orienteering problem and various extensions. In the time from the first mentioning in 1984 by Tsiligirides [23] until now, many people developed algorithms to deal with this NP-hard problem. The complexity is very important in this regard because it allows exact procedures for a limited problem size only. Therefore, most approaches are heuristics and focus on finding good solutions within a short amount of time.

There are several related routing problems that are not OPs per se but should be mentioned in this context. Due to its young age compared to TSP and VRP (Vehicle Routing Problem), some algorithms that turned out to perform well for TSP and VRP, were adapted to the OP. A good example is the idea to partition the map of nodes into sectors and construct tours within sectors first before connecting them. This strategy was first used in the context of VRP by Wren ea (1972) [24] and was later applied to OP by Tsiligirides (1984) [23].

Table 1: Problem Abbreviations

OP	Orienteering Problem
TOP	Team Orienteering Problem
MPOP	Multi-Period Orienteering Problem
TSP	Traveling Salesman Problem
STSP	Selective Traveling Salesman Problem
TSSP+1	Traveling Salesman Subtour Problem with 1 additional constraint
VRP	Vehicle Routing Problem
SVRP	Selective Vehicle Routing Problem
VRPTW	Vehicle Routing Problem with Time Windows
SVRPTW	Selective Vehicle Routing Problem with Time Windows

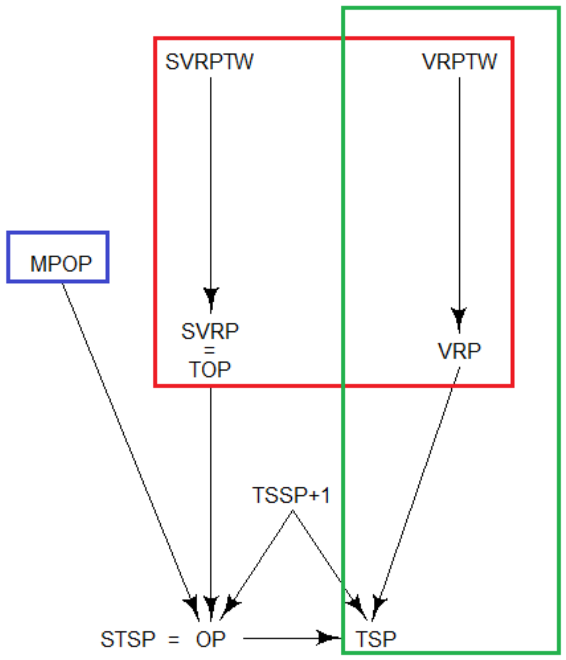

Figure 1: Overview Routing Problems

The arrows are pointing from the general problem to the special case.
The rectangles indicate common properties of the various problem types.

green: Every node is required to be visited exactly once.
red: Multiple tours are allowed but only one visit per node overall.
blue: Multiple tours are allowed and a node can be included in several of them.

Table 1 and Figure 1 give a brief overview of the different routing problems. The graph is not exhaustive but covers the most relevant types. Recall that the TSP minimizes tour length and requires all nodes to be visited and the OP maximizes the number of visited nodes (or the sum of weights) while restricting the total tour length. VRP and TOP are the straightforward extensions to multiple vehicles. It should be noted that the objective for the VRP can have several options, but usually is minimizing the travel cost, typically consisting of a linear combination of the total traveled distance and the number of vehicles used. The MPOP has some parallels with the TOP, both maximizing the total profit by scheduling multiple tours. The difference is that the MPOP allows nodes to appear in various tours, whereas the TOP limits each node to one visit in total at most. The idea behind this is that it can make sense to visit a node every day (MPOP), however it is unintuitive to visit a node on one day with several vehicles (TOP). The introduction of time windows (TW) sets time frames for every node during which it is available and can theoretically by applied to any routing problem. TSSP+1 is a class of problems, which can be generated by adding one constraint to the TSP and optionally removing the constraint to mandatory visiting all nodes. As used by Pillai (1992) [17], it is equivalent to the OP. Furthermore, there are variants for each type of routing problem with symmetric and asymmetric distances as well as start and end points coinciding or not, which are minor changes and do not need an extra notation. If not stated otherwise, the word *depot* will denote starting and end point at the same physical location.

The approaches for dealing with different kind of OPs mostly combine several algorithms in a specific order or in a loop to achieve good results. In the following sections, heuristics and exact algorithms will be considered separately. Nonetheless, heuristics are sometimes applied in exact algorithms to generate bounds or initial tours, which are then improved.

3.1 Heuristics

Cheapest insertion and *nearest neighbor* are two similar algorithms that can construct feasible tours very fast. They were publicized by Rosenkrantz ea (1977) [19] to find solutions for the TSP. Both start with a single node (the depot) and then add more nodes. Cheapest insertion views the current state as tour (as a cycle) and can add nodes anywhere. Nearest neighbor views the current state as a path (from A to B) and can only add points before the first node or after the last one. Which nodes are added is chosen by a greedy criterion, which originally consisted only of the additional length that the inclusion of that node would add to the total length (in the context of TSP). For the OP $\frac{Profit}{Add.\ Length}$ serves as criterion. Points are added until the inclusion of another point would violate the tour length limit. For nearest neighbor, the path has to be converted into a feasible tour by connecting the first and last node with the corresponding edge. This edge always has to be considered, when evaluating the feasibility of a state. According to Laporte ea (1990) [14] cheapest insertion can be done in $\mathcal{O}(n^2 \log n)$ and nearest neighbor in $\mathcal{O}(n^2)$ time.

Another heuristic is 2-*opt* proposed by Croes (1958) [5]. This algorithm can improve an existing tour by rearranging the order of visited nodes. Two nodes are selected and the nodes between them are reversed in order. For example the tour 01234567890 with selected nodes 3 and 8 will transform into 0123**7654**890. Depending on the size of the problem and available time all possible swaps or a random sample are tested. At this point, the depot needs to be protected from swaps. If one of the generated swaps turned out to decrease the total tour length, the corresponding tour is selected as the new benchmark. The algorithm stops, if the tour cannot be improved by any 2-opt swaps. The main idea of 2-opt is to repair paths the cross themselves. This approach can be generalized to k-opt, offering even more swap possibilities increasing both runtime and solution quality. Tsiligirides (1984) [23] Golden ea (1987) [7] and Ramesh ea (1991) [18] use 2-opt and/or 3-opt to improve tours in their OP heuristics.

Removal and insertion is another strategy to improve current tours. This is more of a general idea than a fixed algorithm. If a greedy algorithm such as cheapest insertion or nearest neighbor cannot add any more nodes, it can be beneficial to remove an existing node and try adding another one in return. A newly included point has to lead to a feasible solution. Typically, multiple points satisfy this constraint and the necessary selection can be done deterministically or randomly. Many criteria use scores with different components, which is often referred to as *desirability*. Natural components are its profit, additional distance for including, distance to its neighbors in the current tour or distance to depot. Golden ea (1988) [8] and Sokkappa (1990) [20] use distance to the *center of gravity* as one component and a learning factor, which stores information about previously removed points similar to tabu search. Keller (1989) [12] decided to allow remove/insert whole clusters of points instead of trading one for one.

Tabu search was invented by Glover (1986) [6], originally to improve integer linear programming (ILP). Conveniently, tabu search is a metaheuristic, meaning it can theoretically be applied to any optimization problem. This approach needs an initial solution, of which a neighborhood of similar solutions is computed. Of this neighborhood the best solution is selected (even if it is worse than the current solution). The difference of the old and new solution will be added to the tabu list, meaning that it cannot be reversed for some amount of time. For example, if the new solution was generated from the old one by removing node i, then node i cannot be added for the next few iterations. The process is iterated until a stopping criterion is met. The tabu list prevents local search algorithms from producing cycles like adding and removing the same node over and over again. Of course, the objective value of the solutions has to be tracked and the best solution until now has to be stored separately because even if the best solution is found (but not identified as such), it will be discarded for another solution in the next iteration and will not be met again soon due to the tabu criterion. If the algorithm is stopped, the solution with highest objective value is selected as the final solution. Tang ea (2005) [21] implemented tabu search for the TOP. He generates an initial set of tours by unspecified heuristics. The neighborhood of a given set of tours is computed by randomly adding a node to one of the tours or randomly selecting a random number of points and replace them by others. At this stage infeasible solutions are explicitly allowed, but penalized for exceeding the length limit. The distribution of the number of removed points changes during the tabu search producing small or large neighborhoods to allow inspecting the near neighborhood precisely or escaping a local optimum, respectively. The tabu search terminates, if the current best solution is not updated for a selectable amount of iterations.

Limited discrepancy search (LDS) was introduced by Harvey ea (1995) [11] to search for feasible solutions within decision trees in cases were feasible solutions might be rare in the total search space. The idea is that the solutions produced by heuristics often look similar to the solutions produced by an exact algorithm in terms of decision paths. Therefore, once a solution (possibly infeasible or suboptimal) is found by a heuristic, it makes sense to search similar decision paths. For two decision paths, each distinct decision counts as one discrepancy and all decision paths that have less than a fixed number of discrepancies is considered the neighborhood, which is then inspected for better solutions. In the context of routing problems, the decisions are usually binary and distinguish if a specific edge or node is included or excluded from the tour. This approach works well with methods that follow a tree structure e.g. branch and bound.

Chao ea (1996) [4] invented a powerful heuristic for the OP that was able to outperform six other common heuristics at that time. It was tested on 49 benchmark problems suggested by Tsiligirides (1984) [23] and Chao's algorithm produced total scores that were not beaten by any other algorithm on any of the problem sets but conversely beat every algorithm at least on one problem instance. The benchmark problems assumed starting and end point at different physical locations, which technically requires searching for a path instead of a tour, but that is only a small detail. First, paths are constructed by using cheapest insertion with additional length (without respecting profit) as insertion criterion until every node is visited by exactly one path. The path with the highest score is selected as the current solution. Then exchanges of one point each of the current path and an arbitrary other path are evaluated and the exchange the produces the highest feasible score for any path is selected from all exchanges. The insertion is again done with the cheapest insertion method. Should the other path become infeasible by adding the node, the node will create a new path instead. After each exchange the paths are evaluated and the one with highest score is selected as the current path. If no exchange results in an increase of the current score, small decreases in score are allowed too. The best feasible path that occurred in this process is called record. The next step checks for each node, if moving it feasibly from its present path to any other could raise the record score or lies at least slightly below it. The standard parametrization uses 90% of the record value as a threshold. In case it does, the move is performed, otherwise the node remains on its present path. Moreover, the current path is improved by running 2-opt method and subsequently trying to add points from other paths if possible (with respect to tour length limit). The current path is then removed the k least efficient nodes in terms of the $\frac{Profit}{Add.\ Length}$ criterion, which are added to other paths or creating new ones. The state of paths serves as a new initial solution and the steps are iterated with k increasing from 1 to K every iteration. On the 49 benchmark instances, this algorithm found a solution with the same score as Pillai's (1992) [17] exact algorithm, which was able to solve 47 problem instances within the 2 hours time limit (per instance). Another advantage of Chao's algorithm is its adaptability to the TOP with only small changes. In contrast to other algorithms, the present state of paths consists only of **disjoint** feasible paths which matches with the constraints of the TOP.

3.2 Exact Algorithms

Integer linear programming (ILP) is a popular technique to solve a special class of optimization problems. For this method it is important that the problem can be formulated by using a linear objective function, equalities and inequalities that are linear and integer constraints for some variables. All other constraints are not allowed. Sometimes constraints are not linear, but can be reformulated in a linear way. For example $x^2 < 1$ can be substituted by $x < 1$ and $-1 < x$, which leads to the same feasible set of values for x. If there are no integer constraints, ILP transforms into a linear programming (LP), which can be solved faster. In fact, ILP is NP-hard, while LP is not. LPs are usually tackled by using variants of the simplex algorithm such as the LP solver CPLEX. The process of removing integer constraints (or other constraints) is called relaxation. Solving the relaxed problem can yield an integral solution, which makes it the optimal solution to the original problem, or to a non-integral solution, which makes it an upper bound to the optimal objective value. In the first case the problem is solved, while in the second case the search space has to be separated to avoid the non-integer values. The classic strategy is using branch and bound or a variant of it (such as branch and cut) for this purpose. Branching approaches will be explained separately. The main advantage of ILPs is that LPs can be solved very fast and in case that all constraints have integer coefficients, there is a good chance that the ILP is solved after the first LP, see Boussier (2007) [3]. A disadvantage is that the linear formulation of a problem tends to require a high number of variables and constraints. For this reason, further relaxations (like dismissing subtour constraints) or procedures to reduce the search space (similar to candidate tour identification in section 5.2) are regularly used to reduce problem size.

Branch and bound is a strategy to divide the search space subsequently into smaller elements. At first there is branching, meaning to split the search space into multiple (typically two) spaces forming a partition. In terms of routing, one split could be done creating one branch using node i in the tour and another branch excluding node i completely from the problem. Another way would be to split on using a specific edge. After branching, upper bounds (wlog assuming a maximization problem from now on) are computed for every branch. If there already exist feasible solutions, for example created by a heuristic, as a product of LP solving or because the branch only includes one solution, then it serves as a benchmark and lower bound for the optimal solution. Consequently, all branches with an upper bound, that lies below the benchmark value, are terminated. All remaining branches are again split into subbranches and bounds are computed. When branching continues, the search spaces get smaller, the bounds tend to get tighter and are hence likely to be either terminated or serve as a new benchmark value. Table 2 illustrates the process of combining ILP solving with branch and bound. It can be seen, that even though the relaxation is a simplification of the original problem, the split replaces the integer constraint for the corresponding edge. Noticeable, the edge or node on which to split is not determined randomly, but strategically. In the case of LP, the split is done on an edge or node, that is traveled or visited by a fractional number of vehicles in the current solution. So the variable that violates the integer constraint is chosen to be the branching variable.

Table 2: Branch and Bound ILP Relaxation

Original Problem (ILP)		$X_{ij} \in \{0,1\}$	
Relaxed Problem (LP)		$0 \leq X_{ij} \leq 1$	
Split on X_{ij}	$X_{ij} = 0$		$X_{ij} = 1$

For explanation of the X_{ij} see section 4.2.

Branch and cut is an extension of branch and bound. In the first step, the relaxed LP is solved, but instead of branching immediately, the LP is enhanced by a cutting-plane method. The aim is to find a cutting plane (represented by a linear inequality), which separates the non-integral solution of the LP, from all integer feasible solutions. In case such cutting planes exist, they are included in the relaxed LP as new constraints and the LP is solved again. Otherwise, a branching step is performed. All occurring non-integral solutions serve as upper bounds for their branch and all occurring integral solutions serve as lower bounds to cut off branches of the tree. Branch and bound and cutting planes combined usually outperforms both individual algorithms due to a good synergy. Cutting planes accelerates the branch and bound tree by offering tighter bounds on the LPs, which allows to cut off suboptimal branches earlier. Conversely, branch and bound divides the search space, which is beneficial for finding more cutting planes. This method is applied in Pillai's (1992) [17] algorithm.

Another modification of branch and bound is *branch and price*. Like branch and cut it is specifically designed to solve ILPs. In some optimization problems, especially routing problems, ILPs can contain a lot of columns, but typically many of them are *non-basic*, meaning they have a value of zero and do not contribute to the objective function. Hence, many columns are not necessary in the ILP design, but it is unknown which columns are important and which are not, straight away. In this approach, the strategy is to start with a very small set of column and add new columns gradually, if and only if they have the potential to contribute to the objective function. The optimization task is divided into two parts. The master problem denotes the original problem formulated as ILP and includes all columns. After removing all but some elementary columns, it is called restricted master problem. The second part is called subproblem or *pricing* and consists of finding additional columns for the restricted master problem. Identifying potentially useful columns can be done by evaluating a simple expression containing the dual variables of the current LP solution and the coefficient of the objective function. The algorithm starts with the small-sized restricted master problem and solves it. All LP solves are done for the relaxed problem discarding integer constraints. The subproblem adds columns based on the LP solution and solves the LP again until no additional columns are found. If the solution at this stage is integral, the optimum is found. Otherwise, a branching step is executed to enforce integrality of one of the non-integer variables and provide a bound. A branch with an upper bound lower than a feasible solution anywhere in the tree is terminated. The process of column generation, branch and bound is iterated until all branches are terminated or provide a feasible solution.

Boussier ea (2007) [3] apply the concept of branch and price to treat TOP and SVRPTW the following way. Their work is an enhancement to Gueguen (1999) [9]. The master problem consists of an ILP with one column corresponding to one tour and one row corresponding to high level constraints such as the number of vehicles. The restricted master problem consists only of the elemental columns, which are tours that visit only one node (besides depot). The subproblem searches for profitable tours and checks feasibility of the potential columns. Feasibility requirements at this level include the tour length limit, subtour constraints and optionally time windows as well as the very basic constraints of visiting the depot and not visiting a node more than once per tour. Checking feasibility is done by a dynamic programming approach based on the Bellman-Ford algorithm. The Bellman-Ford algorithm (Bellman (1958) [2]) is an exact method that tries out different paths and updates the shortest path to each node every time it finds a better path or

subpath using a labeling system. The solution includes the shortest paths from one starting node (depot) to every single node. If a tour turns out to be feasible, it is added as a column to the restricted master problem. The branching to avoid non-integer solutions distinguishes multiple cases.
If a node exists, such that this node is visited by a fractional number of vehicles, then one branch excludes this node completely, whereas the other branch enforces the node to be visited. These new node constraints are added to the restricted master problems within the corresponding branches and result in column deletions in case of excluded nodes.
If all nodes are integral, but some edges are not, then one non-integral edge is selected as a splitting criterion. If one of the adjacent nodes is enforced (by a previous branch split on a node), then new split generates two branches, one enforcing the edge and one excluding it.
If all nodes are integral, some edges are not and the adjacent nodes of the chosen edge are not enforced, three branches are generated. The first branch excludes the starting node of the edge (resulting in excluding the edge as well). The second branch enforces the starting node and the edge. The third branch enforces the starting node, but excludes the edge. Contrary to node constraints, edge constraints are added to the subproblem level.

Dynamic programming approaches try to reduce computation time by storing previous results and recalling them instead of solving a subproblem repeatedly. For suitable problems, dynamic programming outperforms other algorithms, if the provided memory space is sufficiently large. For instance, finding the 100th Fibonacci number is hard, if no previous numbers except for the first two are known. However it is very easy, if the 98th and 99th Fibonacci number are already stored in the memory. The possibilities for dynamic programming are numerous and often diverse so they cannot be described as a standard algorithm. Dynamic programming approaches do have in common that they memorize previous results, have some kind of loop or hierarchical structure and try to use previous results to solve the current problem.

The algorithm proposed in this thesis is, as mentioned before, an exact method specifically designed for the MPOP. It applies a separation of the main problem (finding the optimal schedule) and a subproblem (identifying candidate tours), which is very important when dealing with multiple time steps or vehicles. Unlike many known approaches it does not use the structure of ILP in combination with a branch and bound variant for the main problem. At the level of solving the subproblem it performs a top-down dynamic programming approach that calls external procedures. Tours are treated in the order of the number of contained nodes and each tour runs the stages of a bound calculated from the distance matrix, a linear assignment solver and finally the TSP solver Concorde (Applegate 2002 [1]), which is based on branch and cut. The advantages of this approach are that once the subproblem process is completed it needs not to be called again and many tours do not even have to be considered once a tour containing it as a subtour was found to be feasible. Furthermore, the number of candidate tours is very small compared to the number of all theoretically possible tours. This process is very similar to column generation (pricing) in the branch and price algorithm. Optimizing the schedule is done using classic enumeration with an additional technique to reduce the number of possibilities that have to be checked. While some subroutines are similar to other approaches, the specific combination of bound, LAP and TSP solving in combination with subtour discarding is new and turns out to be efficient. The enumeration process is probably not competitive but could be replaced by linear programming.

4 The Multi-Period Orienteering Problem

4.1 Notation and Mathematical Problem Formulation

The basic problem in the MPOP is to find a T-tuple of tours that maximizes the total collected profit, while satisfying some constraints. To define the MPOP consider a set of nodes $\{0, 1, ..., N\}$, where 0 is the depot, which is the starting and end point for all tours. Depending on the application these nodes represent cities, fields, houses or other places of interest. The depot is the origin in this setting which can be home loaction, office or parking area. Furthermore, there is a finite number of time steps $t = 1, ..., T$. These time steps can be days, hours, minutes, etc. If a person travels from Monday to Friday with one tour per day, then T would be equal to 5. Let w_i denote the weight of node i measuring the profitability of a node. Nodes with a higher weight are more rewarding. In a network of cities, a bigger city might have a higher weight to outline a higher number of potential customers. For a farmer, a high-weight node would represent a field with rich harvest. Weight is not the only variable with influence on the value of a node at a certain time step. z_i^t is a non-negative, time-dependent variable describing the value (profit upon visit) of node i and changing its value baased on the time since the last visit scaled by the decay factor q. The base model is a linear growth over time. This growth is proportional to the weight of the node. Additionally, there is an exponential decay. Here, q is the ratio of demand of time step t and $t + 1$ before applying linear growth for nodes that are not visited at time step t. The decay factor q can take values in $[0, 1]$, where $q = 1$ would mean that there is no decay and the growth is exactly linear and $q = 0$ would mean that everything unvisited decays instantly. The latter case will transform the MPOP into a classic OP since it the decay will reset all node values to the starting values and lead to the same problem in every time step. In the application example $q = 0.9$ will be used, so 90% of the uncollected value can still be collected at the next time step. Each pair of nodes has a fixed, symmetric distance $d_{ij} >= 0$. (Note, that distances are not required to satisfy the triangle inequality.) In every time step, the total tour length is restricted to be shorter than its constraint level C. The variables d_{ij} and C always share the same dimensional unit, which can be length, time or money for instance. There is no refund for unused resources and they cannot be carried over to the next time step.

The variables X_{ij}^t are indicators being 1 if node j follows directly after i in time step t and 0 otherwise. These variables do not play a major role in the algorithm development but they are key to precise problem formulation as they are the link between nodes and distances on the one side and actual tours on the other side. For simplification there are the additional variables y_i^t indicating if node i is visited in time step t. These help to evaluate the total gained profit of a tour. For subtour elimination the auxiliary variables u_i^t will be used. They specify the positions of the corresponding nodes within the tour for nodes included and give no information about (and no constraints to) nodes not included in the tour.

$$X_{ij}^t = \begin{cases} 1 & \text{if j follows immediately after i in time step t} \\ 0 & \text{otherwise} \end{cases} \qquad (1)$$
$$\text{for} \quad i = 0, ..., N;\ j = 0, ..., N;\ t = 1, ..., T$$

The variables X_{ij}^t exist for all pairs of nodes (including the depot) for all time steps. Every segment (travel from one node to the consecutive one) of any possible tour at any time step is represented by one y_i^t. They are the link from nodes to tours and their properties such as length, connectivity and avoidance of cycles.

$$y_i^t = \begin{cases} 1 & \text{i is visited in time step t} \\ 0 & \text{otherwise} \end{cases} \qquad (2)$$
$$\text{for} \quad i = 1, ..., N;\ t = 1, ..., T$$

All information of the y_i^t is already included in the X_{ij}^t. Still the introduction of these new variables helps to construct a more intuitive formulation of the objective function and the variables z_i^t. Unlike the X_{ij}^t, the y_i^t are not defined for the depot (node 0), since it is included in every tour and does not contribute to the objective function.

$$\sum_{t=1}^{T}\sum_{i=1}^{N} w_i \cdot z_i^t \cdot y_i^t \rightarrow max \qquad (3)$$

The objective is to maximize the total reward. Depending on the application this can be profit/demand, harvest or collected/delivered goods. For each time step the tour value equals the product of node weight and current state of growth and decay. In the example of a farmer the node weight represents the amount of fresh harvest per day and growth and decay measures the accumulation of harvest over the days since the last visit and the average quality (freshness). Visiting the field after 5 days will be more rewarding then on the first day (because the amount of harvest is bigger), but it will be less rewarding then visiting everyday (because 4/5 of the harvest is not fresh). The exact values heavily depend on the choice of q. A small value for q (near 0) will mean a fast decay and a strong incentive to visit important nodes very frequently, whereas a high value for q (near 1) will mean a slow decay with an incentive to visit many different nodes. The total reward amounts to the sum of the tour values over all time steps. There is no bonus for remaining resources such as making full use out of the tour length constraints.

$$z_i^t = \begin{cases} 1 + q \cdot z_i^{t-1} & y_i^{t-1} = 0 \\ 1 & y_i^{t-1} = 1 \quad \text{or} \quad t = 1 \end{cases} \qquad (4)$$
$$\text{for} \quad i = 1, ..., N; \; t = 1, ..., T$$

As mentioned before, the variables z_i^t serve as a measure to assess the current state of a node. It evaluates the accumulation of reward and its decay and can be interpreted as a multiplier of how much more reward the node offers now compared to the first time step. The advantage of a growth/decay process over a basic linear accumulation process is that it allows a more realistic modeling for the typical routing problems. Some examples for decay are harvest losing quality when not picked up at the right time or potential customers not buying your product or buying from other competitors if there is no offer in time. Mathematically the z_i^t follow a geometric series (in t) and therefore are monotonically increasing and bounded by $\frac{1}{1-q}$ as long as the corresponding node is not visited given that $0 < q < 1$. Upon visit the node is reset to its starting value (1).

$$\sum_{j=0}^{N} X_{0j}^t = \sum_{i=0}^{N} X_{i0}^t = 1 \quad \text{for} \quad t = 1, ..., T \qquad (5)$$

Equation (5) makes sure that the tours all start and end at the depot (node 0). The first part of the equation represents the number of nodes visited immediately after the depot. The second part represents the number of nodes visited immediately before the depot. For every tour at every time step these have to be equal to 1, otherwise tours excluding the depot were feasible. Theoretically, it would be sufficient to satisfy ≥ 1 in (5) because visiting the depot more than once would not be profitable. In this case = 1 is definitely the better choice to make the equation more intuitive and clear, but for linear programming it is typically faster to use inequalities instead of equalities.

$$y_k^t = \sum_{i=0}^{N} X_{ik}^t \overset{(6a)}{=} \sum_{j=0}^{N} X_{kj}^t \overset{(6b)}{\leq} 1 \quad \text{for} \quad k = 1, ..., N;\ t = 1, ..., T \tag{6}$$

Unlike the depot, other nodes do not necessarily have to be included in all tours, but it is important to restrict visits to one per node per time step. The first equality sign is simply another form of rewriting the definition of the y_i^t. (6a) ensures that the path is connected and there are no dead ends. When viewing the X_{ij}^t as edges on the graph of all nodes, there has to be an outgoing edge if and only if there is an incoming edge. (6b) prohibits multiple visits of the same node within the same time step.

$$\sum_{i=0}^{N} \sum_{j=0}^{N} d_{ij} \cdot X_{ij}^t \leq C \quad \text{for} \quad t = 1, ..., T \tag{7}$$

(7) secures that the tours do not exceed the prescribed length limit. For every time step this sum runs over all distances, efficitively suming up only the ones that are actually travelled ($X_{ij}^t = 1$) and ignoring the other ones ($X_{ij}^t = 0$). (7) also determines that the tour length limits for different time steps are independent, especially the unused distance resources cannot be carried over to the next time step.

$$1 \leq u_i^t \leq N \quad \text{for} \quad i = 1, ..., N;\ t = 1, ..., T \tag{8}$$

$$u_i^t - u_j^t + 1 \leq N(1 - X_{ij}^t) \quad \text{for} \quad i = 1, ..., N;\ j = 1, ..., N;\ t = 1, ..., T \tag{9}$$

The combination of (8) and (9) eliminates subtours as proven by Miller et al. [15]. The variables u_i^t determine the position of the corresponding nodes within the tour in time step t. The positions range from 1 to a maximum of N as declared by (8). For a node i not included in the tour all $X_{ij}^t = 0$ so (9) simplifies to $u_i^t - u_j^t \leq N - 1$, which is trivially satisfied for all j if (8) holds true. For a node i' indeed included in the tour there exists one j' such that $X^t_{i'j'} = 1$. In this case (9) simplifies to $u^t_{i'} \leq u^t_{j'} - 1$. So, if one part of the tour travels from i' to j' then it follows that the position of i' within the tour is at least 1 point smaller than the position of j'. If i' should be the last node of the tour before visiting the depot, then it is necessary to swap the roles of i' and j'. This works unless i' is the only node visited on this tour, but in this case subtours cannot exist. The values of the variables u^t_i are not of particular interest, they are only used in (8) and (9) to prevent subtours.

(3)-(9) can be seen as a generalization of the problem formulation by Gunawan et al. [10] to multiple time steps.

The occurring variables are taking values in different sets.

$$\begin{aligned}
&X^t_{ij} \in \{0,1\} \quad \text{for } \ i = 0, ..., N; j = 0, ..., N; t = 1, ..., T \\
&y^t_i \in \{0,1\} \quad \text{for } \ i = 1, ..., N; t = 1, ..., T \\
&z^t_i \in \left\{1, 1+q, 1+q+q^2, \dots, \sum_{k=0}^{T-1} q^k\right\} \text{ for } \ i = 1, ..., N; t = 1, ..., T \\
&w_i \in \mathbb{R}^+ \quad \text{for } \ i = 1, ..., N \\
&u^t_i \in [1; N] \cap \mathbb{R} \quad \text{for } \ i = 1, ..., N; t = 1, ..., T
\end{aligned} \tag{10}$$

4.2 Tour Length Distribution

When dealing with problems with tour length restrictions, it is always a good idea to have a look at the actual distribution of the tour length. For the sake of simplicity, assume that problem instances are generated by N iid uniformly distributed nodes in a square and the Euclidean distance serves as a distance measure. The interesting quantity is the distribution of the shortest tour (TSP solution) that visits all nodes and satisfies (5)-(9) for $T = 1$. While these distributions will have a rather complicated form, a simple example will be used first.

Consider a unit square with the depot being located at the center of the square. The simplest case is $N = 1$, which means there is one additional node. This node is assumed to be uniformly distributed on the unit square. The total tour length of the tour $(0, 1, 0)$ is given by $d_{01} + d_{10}$ being equal to twice the Euclidean distance of the nodes. Points with same Euclidean distance to a reference point always depict a circle, but for radii greater than $1/2$ since parts of their circle lines lie outside of the unit square. This behavior is illustrated in Figure 2.

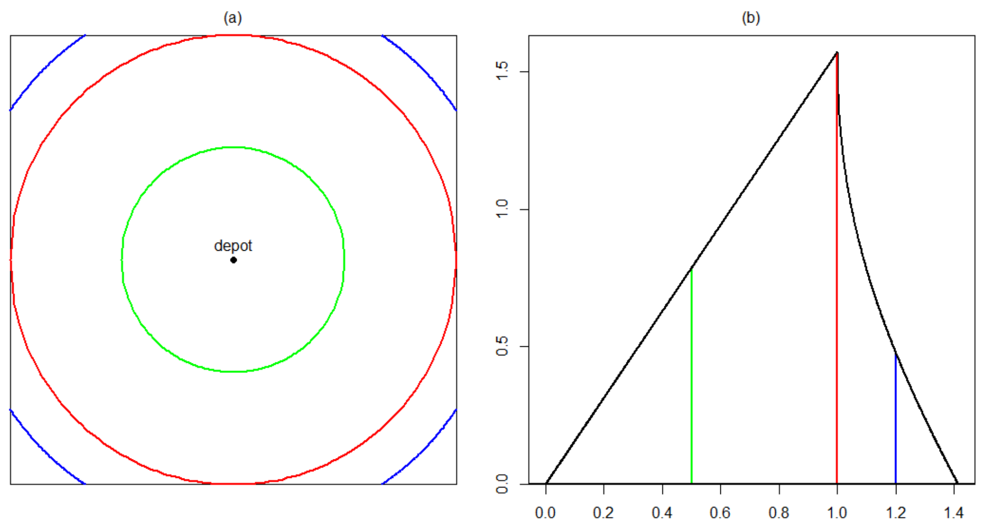

Figure 2: Tour length distribution for N=1

(a): Isolines of radii 0.25 (green), 0.5 (red), 0.6 (blue)
(b): Density at corresponding tour length values 0.5 (green), 1 (red) and 1.2 (blue)

The key to finding a closed-form expression for the density is calculating the ratio of the circle line that lies within the unit square. After using the Pythagorean theorem and the definition of cosine the formula can be observed. In the following formula, L denotes the tour length.

$$f(L) = \begin{cases} L \cdot \frac{\pi}{2} & \text{for } 0 \leq L \leq 1 \\ L \cdot \arccos\left(\frac{2\sqrt{L^2-1}}{L^2}\right) & \text{for } 1 < L \leq \sqrt{2} \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

For $N > 1$ the problem gets very complicated and therefore densities are estimated by using `Concorde` algorithm (TSP solver) and kernel densities on simulated data. The results are visualized in Figure 3.

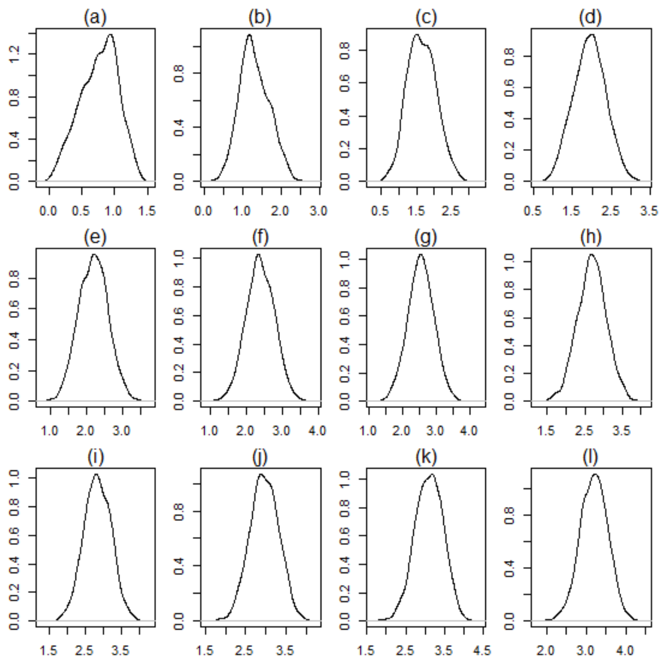

Figure 3: Simulated tour length distributions for $N = 1, ..., 12$

y-axis: Density
x-axis: Tour Length
(a)-(l) correspond to N=1,...,12

As seen in (11) the distribution of tour length for a centered base with a single node is already quite complicated and even more complicated for a higher number of nodes N. Luckily, these distributions seem to converge to a normal distribution very fast. That makes approximating the distributions, even without knowing their exact formulas, easy. Tour length distributions can be used to classify different constraint levels into groups for fixed assumption on number and distribution of nodes using the quantiles of the distribution.

5 The Algorithm

5.1 Structure

Roughly summarized, the algorithm uses the steps described in the following sections to identify candidate tours first in order to reduce the runtime complexity. It then provides the optimal schedule which is a T-tuple of individual (feasible) tours for each time step. As input it receives the distance matrix D, tour length limit C, the number of time steps T, a weight vector to reflect the importance of the respective nodes and the decay factor q.

To be more precise, the task is finding the T-tuple of tours that maximizes our objective function (e.g. gaining the most profit from visiting nodes) under the constraint that each of the T tours must be feasible. In the subsection "Finding Candidate Tours", which describes part one of the algorithm, the feasible tours get separated from the infeasible tours and within this set of feasible tours only the Pareto-optimal tours (in this thesis called candidate tours) are taken into consideration. This results in a list of (candidate) tours. In part two, which is explained in the subsection "Determine Optimal Schedule", all possible combinations of drawing T tours from this list with replacement are generated. Some of them can be discarded due to their structure that will prevent them from being optimal. The remaining T-tuples of tours are evaluated and the best one is chosen.

The following pseudo code describes the structure of the algorithm while being unspecific in when to switch from one sub procedure to the next one. This will depend on the choice of strategy. In section 5.2.3 these strategies and will be explained in detail including their specific pseudo code variant.

Algorithm 1 Structure of Algorithm

```
 1: procedure FINDING CANDIDATE TOURS  (Part 1)
 2:     Input: Distance Matrix D, Tour Length Limit C
 3:     Compute Bound from subsection 5.2.1
 4:     $NV_{max} \leftarrow computed_{bound}$
 5:     $k \leftarrow \mathrm{NV}_{\mathrm{max}}$
 6:     $candidate.tours \leftarrow$ empty list
 7:     $potential.candidate.tours \leftarrow$ empty list
 8:
 9: LAP Solving
10:     Generate all k-tuples of nodes, check the LAP solution and add to
        $potential.candidate.tours$ if it satisfies $LAP_{sol} \leq C$.
11:     $k \leftarrow k - 1$
12:
13: Evaluate Potential Candidates
14:     Solve TSP for each tour of $potential.candidate.tours$ and add to
        $candidate.tours$ if it satisfies $TSP_{sol} \leq C$.
15:
16: TSP Solving
17:     Generate all k-tuples of nodes, discard all subsets of tours of
        $candidate.tours$ check the TSP solution and add to $candidate.tours$ if
        it satisfies $TSP_{sol} \leq C$.
18:     $k \leftarrow k - 1$
19:
20: procedure FINDING OPTIMAL SCHEDULE  (Part 2)
21:     Input: Vector of Weights w, Number of Time Points T,
        $candidate.tours$
22: Enumeration
23:     Generate all T-tuples of tours by drawing with replacement from
        $candidate.tours$.
24:     Discard unpromising tours.
25: Evaluation
26:     Evaluate all remaining T-tuples of tours and return the best one.
```

5.2 Finding Candidate Tours

In order to identify candidate tours two properties will be used:

1. A candidate tour must be feasible and optimal (TSP solution) for a given set of visited nodes.

2. Subtours of candidate tours cannot be candidate tours.

In this context, a subtour is a tour such that the set of visited nodes is a subset of the set of visited nodes of the original tour. Furthermore, the set of visited nodes of a candidate tour will be called candidate set. To illustrate the second condition, consider a subtour of a candidate tour. This tour would in any case be outperformed by its supertour. The supertour is a candidate tour and therefore feasible. By this result the subtour should never be used and can be discarded. These requirements reduce the set of candidate tours to the smallest size possible without losing tours that could be part of the optimal solution. Nevertheless, the number of candidate tours can get very large.

The theoretical maximum of candidate sets is

$$n_{max} = \begin{cases} \binom{N}{N/2} & \text{if N is even} \\ \binom{N}{(N-1)/2} & \text{if N is odd} \end{cases} \tag{12}$$

To show that this number is identical to the theoretical maximum of candidate tours, it is necessary to find a bijective function that maps the set of candidate tours onto the set of candidate sets. Suppose every candidate tour is mapped onto its own candidate set. This function is clearly surjective since every candidate set has at least one candidate tour that maps onto it. If the TSP solution for all candidate sets is unique, then the mapping is also injective. For candidate sets with more than one optimal tour, the candidate tour can be chosen arbitrarily from the set of TSP solutions for this candidate set. Therefore, (12) also describes the theoretical maximum of candidate tours.

The bound mentioned in (12) holds for arbitrary distance measures. This bound is tight for problem instances that use the following discrete distance measure with $C \overset{!}{=} {}^{N}\!/_{2}$ as tour length limit.

$$d_{ij} = \begin{cases} 1 & \text{for } i \neq j \\ 0 & \text{for } i = j \end{cases} \tag{13}$$

It is not clear, if this bound is tight for all N in the setting of section 4.2 with nodes in a plane and Euclidean distance. There probably exists a narrower bound, because examples like the distance matrix generated by (13) cannot be constructed for Euclidean distances. At least not for finite dimensional Euclidean spaces with N sufficiently large.

Figure 4 and Table 3 are depicting results of simulations to approximate the distribution candidate tours for given N (from 1 to 12) and C (10 equidistant constraint levels, chosen to be representative depending on N). Data points were sampled from a uniform distribution over a unit square and different constraint levels. Each combination of N and C was sampled 10 times while counting the number of candidate tours for every iteration. The total number of analyzed datasets is $1200 = 12 \cdot 10 \cdot 10$.

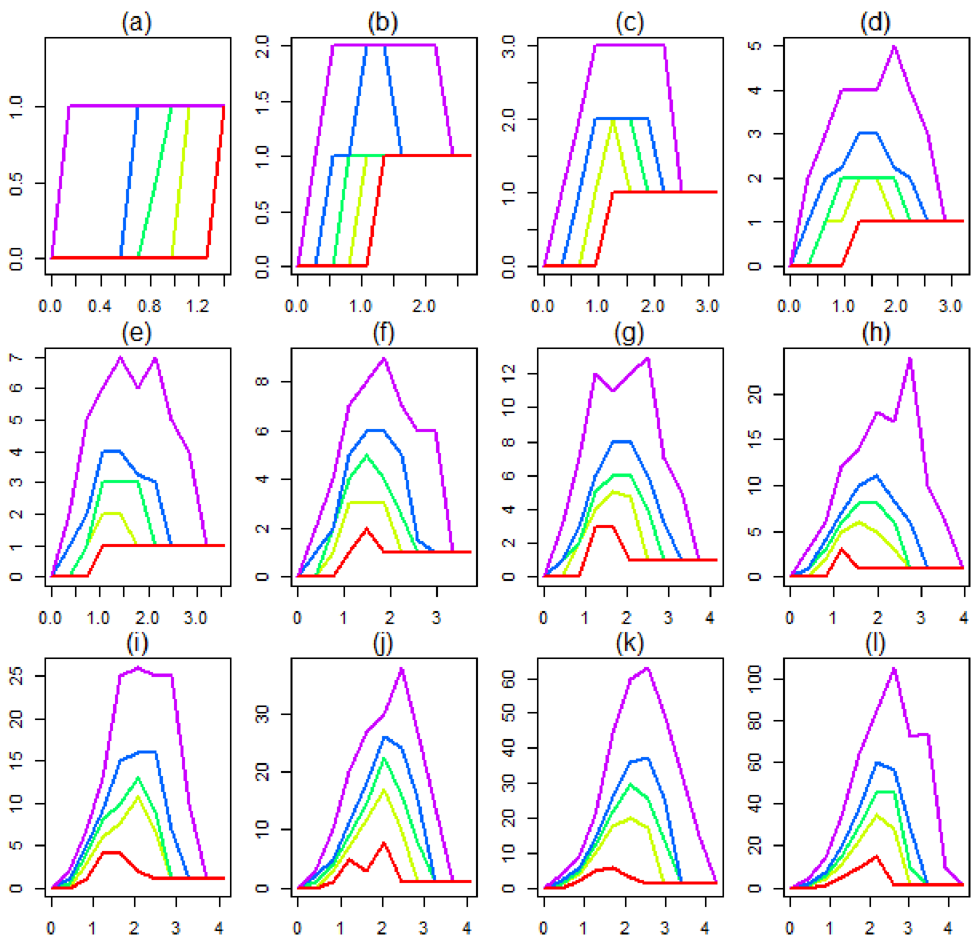

Figure 4: Simulated Distribution of Number of Candidate Tours

y-axis: Number of Candidate Tours
x-axis: Constraint Level
(a)-(l) correspond to N=1,...,12
colors: observed minimum (red), 0.25 quantile (yellow), median (turquoise), 0.75 quantile (blue) and observed maximum (purple)

Figure 4 shows that the isolines (for fixed percentile) tend to look like a normal distribution for increasing N. These distributions are relevant for predictions of runtime since the number of candidate tours is the most important variable in this regard. A good way to use this, could be designing a simulation study and determine a reasonable number of iterations.

Table 3: Number of Candidate Tours - Observed and Theoretical Maximum

N	observed	theoretical
1	1	1
2	2	2
3	3	3
4	5	6
5	7	10
6	9	20
7	13	35
8	24	70
9	26	126
10	38	252
11	63	462
12	105	924

As Table 3 shows, there is an increasing gap between the observed and theoretical maximum at least for higher values of N. In this case, it is advantageous to obtain a gap, since it reduces the number of candidate tours and in further consequence the runtime. It might be worth mentioning again that these results come from the two-dimensional Euclidean space and might have completely different distributions when using different distance metrics.

5.2.1 Upper Bounds for Number of Nodes

Based on the entries of the distance matrix it is possible to derive bounds for the number of node visits for a given tour length limit. For this purpose, the depot is not counted as a node visit. To formulate the boundary condition it is useful to introduce some new variables.

Let m_i denote the distance from node i to the nearest distinct node (excluding depot) and denote their order statistics by $m_{1,N}$ to $m_{N,N}$. Furthermore, let $d^0_{i,N}$ denote the i-th order statistic of the depot distances. The second subscript (N) indicates the size of the sample.

$$m_i = \min_{\substack{j \in \{1,\ldots,N\} \\ j \neq i}} d_{ij} \qquad \text{for} \quad i = 1, ..., N \tag{14}$$

$$m_{1,N} \leq ... \leq m_{N,N} \text{ are the order statistics of } \{m_1 \,, \ldots, \, m_N\} \tag{15}$$

$$d^0_{1,N} \leq ... \leq d^0_{N,N} \text{ are the order statistics of } \{d_{0,1} \,, \ldots, \, d_{0,N}\} \tag{16}$$

The boundary condition for number of node visits (NV) is defined as follows:

$$B(k) := \begin{cases} d^0_{1,N} + d^0_{2,N} + \sum_{i=2}^{k} m_{i,N} & \text{for } 2 \leq k \leq N \\ 2 \cdot d^0_{1,N} & \text{for } k = 1 \\ 0 & \text{for } k = 0 \end{cases} \tag{17}$$

$$NV_{max} = \max_{\substack{k \in \{0,1,\ldots,N\} \\ B(k) \leq C}} k \tag{18}$$

Proposition 1.

NV_{max} as defined in (17) and (18) is a valid upper bound for the number of nodes of tours satisfying (5)-(9).

In (17) the right side can be viewed as the highest number of node visits that can still lead to a feasible tour. To prove this, consider an arbitrary tour satisfying conditions (5)-(9). Therefore $\exists! \ k \in \{0, 1, ..., N\}$ such that $NV = k$. Without loss of generality, the visited nodes shall be $\{1, ..., k\}$ in this specific order. This can be achieved be renaming of the nodes. Since for $k \in \{0, 1\}$ the proof is trivial, from now on $k \in \{2, ..., N\}$ will be assumed. The length of this tour can now be written as follows.

$$\begin{aligned} L(\{1, ..., k\}) &= d_{0,1} + d_{1,2} + ... + d_{k-1,k} + d_{k,0} \\ &= d_{0,1} + d_{0,k} \quad + \quad d_{1,2} + ... + d_{k-1,k} \end{aligned} \tag{19}$$

By definition, the following inequalities hold.

$$m_i \le d_{i,i+1} \qquad \text{for} \quad i = 1, ..., N-1 \tag{20}$$

$$m_{i+1} \le d_{i,i+1} \qquad \text{for} \quad i = 1, ..., N-1 \tag{21}$$

In the next step for an arbitrary $j \in \{1, ..., k\}$ (19) will be used for $i = 1, ..., j-1$ and (20) will be used for $i = j, ..., k-1$.

$$\begin{aligned} d_{1,2} + ... + d_{k-1,k} \ge m_1 + ... + m_{j-1} \ + \ m_{j+1} + ... + m_k &= \\ = \left(\sum_{i=1}^{k} m_i \right) - m_j \quad \text{for} \quad j = 1, ..., k \end{aligned} \tag{22}$$

Taking the maximum on both sides over this band of inequalities yields the following.

$$\begin{aligned} d_{1,2} + ... + d_{k-1,k} &\ge \max_{j \in \{1,...,k\}} \left\{ \left(\sum_{i=1}^{k} m_i \right) - m_j \right\} \ge \\ &= \left(\sum_{i=1}^{k} m_{i,k} \right) - \underbrace{\min_{j \in \{1,...,k\}} m_j}_{=m_{1,k}} = \\ &= \sum_{i=2}^{k} m_{i,k} \ge \sum_{i=2}^{k} m_{i,N} \end{aligned} \tag{23}$$

$$d_{0,1} + d_{0,k} \ge d^0_{1,N} + d^0_{2,N} \qquad \text{for} \quad k = 2, ..., N \tag{24}$$

Plugging (22) and (23) into (18) finally yields the desired result and proves Proposition 1.

$$L(\{1, ..., k\}) \ge d^0_{1,N} + d^0_{2,N} + \sum_{i=2}^{k} m_{i,N} \qquad \text{for} \quad k = 2, ..., N \quad \square \tag{25}$$

(24) proves that $B(k)$ as defined before is a valid lower bound for the tour length of tours with k node visits. This bound will be tight for very small values of k but not for higher values. As the problem is NP-hard one cannot expect to find an easy-to-compute tight bound. Still this bound helps a lot since it has a negligible runtime and discards all tours with $NV > NV_{max}$. This is convenient because the tours with high NV correspond to the hardest TSPs.

To estimate the effect of this bound it is useful to have a look at some simulations. These are visualized in Figure 5 and 6. The key features of the following simulations are uniformly randomly distributed points in a 10 by 10 square, Euclidean distance measure, 9 different tour length limit levels $(1, 2, 3, 5, 10, 15, 20, 25, 30)$ on 5 different problem instances. For now, no LAP solving is included and candidate tours are identified by TSP solving only. This is equivalent to Strategy 1 in section 5.2.3 .

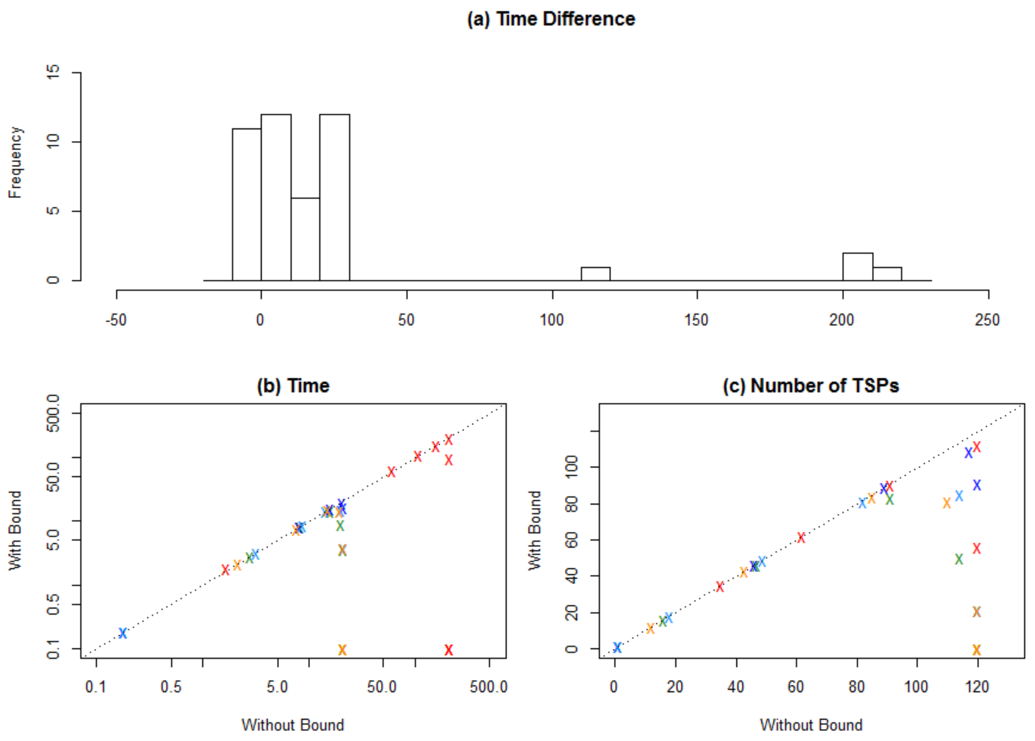

Figure 5: Runtime - Bound vs No Bound 1

Fig.5a
x-axis: $Runtime_{NoBound} - Runtime_{Bound}$ [sec]
Fig.5b
x-axis: Runtime [sec] **without** bound (on log scale)
y-axis: Runtime [sec] **with** bound (on log scale)
Fig.5c
x-axis: Number of TSPs **without** bound
y-axis: Number of TSPs **with** bound

On the dotted line (diagonal) both algorithms have the same runtime (b) or number of TSPs (c), respectively.
Five different colors correspond to five different problem instances.

Figure 5a gives a nice overview on time differences caused by using the bound from Prop. 1. Roughly one half of the simulations only have a small difference ($<$ 10 sec), whereas the other half shows an improvement between 10 and 220 seconds in favor of using the bound. Having no outlier instances, in which the bound approach would be heavily outperformed, is important here, since it is desirable to have a reliable strategy with some benefits rather than a high-risk, high-reward approach.

This behavior can also be seen in Figure 5b. The datapoints near the dotted line have a similar performance, while datapoints in the lower right corner reflect instances that can be solved more efficiently by using the bound approach. Due to the skewed distribution with a few outliers with high runtimes, the time is log scaled on both axis.

Closely related to runtime is the number of TSPs that need to be solved in order to identify candidate tours. TSPs are the main source for runtime in the first part of the algorithm, but the raw number might not be a good estimate. TSPs are solved hierarchical with the largest TSPs (the ones with the most nodes) in the beginning and are consecutively getting smaller. The effort it takes to solve TSPs grows fast when increasing the number of nodes, so leaving out the first few TSPs can conserve runtime considerably, while only slightly reducing the raw number of TSPs. This can be seen when comparing Figure 5c with Figure 5b. Figure 5c shows one special yellow point in the lower right corner. In this problem instance the bound reduces the number of TSPs from 120 to 0. That effect also cuts down the runtime from roughly 20 seconds to almost zero.

To confirm the suspected effect, a test was conducted. 10000 bootstrap iterations were generated and lead to an empirical distribution of the average runtime reduction. A 95% confidence interval will then decide whether the

nullhypothesis $(mean(Runtime_{NoBound} - Runtime_{Bound}) \leq 0)$

can be rejected in favor of the

alternative hypothesis $(mean(Runtime_{NoBound} - Runtime_{Bound}) > 0)$.

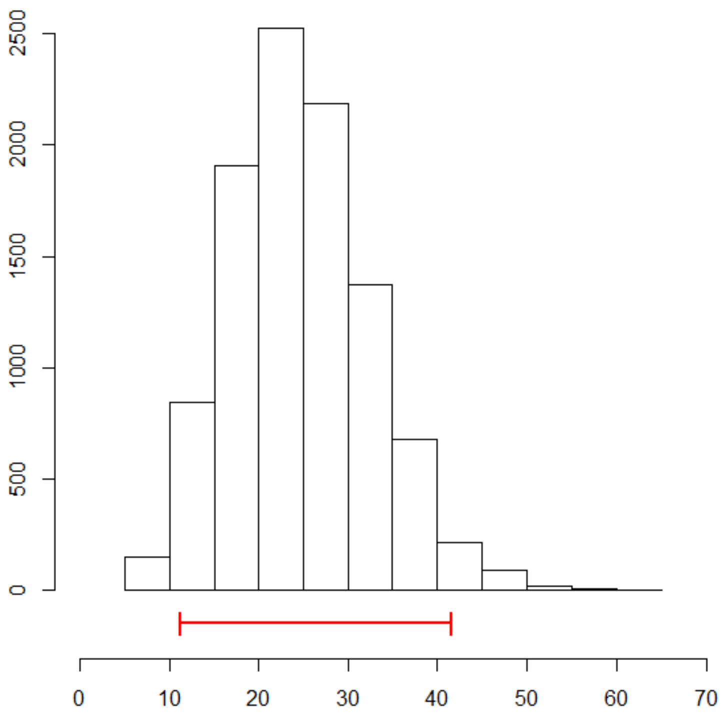

Figure 6: Bootstrap - Bound vs No Bound 1

y-axis: Bootstrap Frequencies
x-axis: Bootstrapped Mean [sec]
The red line is a 95% confidence interval for the average time reduction by using the bound from Prop. 1.

The positive effect, already seen in the previous graphs in Figure 5, of applying the bound is affirmed by the bootstrap results. Not even a single one of 10000 bootstrap simulations found a negative change in average runtime. In this regard, it is advantageous to obtain a confidence interval with respectful distance to zero even for a small sample size.

After withstanding this first test, there is another important comparison for this bound. As 5.2.3 and 5.2.4 explain in detail, also solving of Linear Assignment Problems (LAPs) is an option to reduce the number of TSPs. Strategy 2 and Strategy 3 use this approach and as Strategy 3 turns out to be best it also be evaluated with and without the bound. For this reason, the latest experiment will be repeated but Strategy 3 (includes LAPs) will be used instead of Strategy 1 (no LAPs). The results are displayed in Figure 7 and Figure 8.

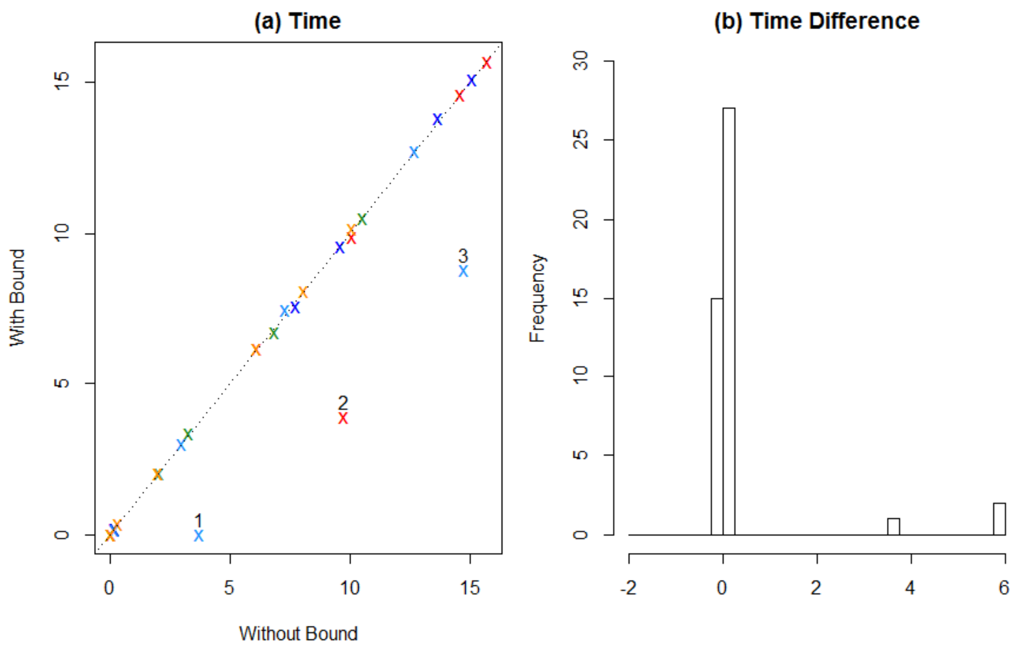

Figure 7: Runtime - Bound vs No Bound 2

Fig.7a
x-axis: Runtime [sec] **without** bound
y-axis: Runtime [sec] **with** bound
Fig.7b
x-axis: $Runtime_{NoBound} - Runtime_{Bound}$ [sec]
On the dotted line (diagonal) in Fig.7a both algorithms have the same runtime.
Five different colors correspond to five different problem instances.

In Figure 7a a large number of datapoints very near to the diagonal and 3 points, which show an improved performance by using the bound, can be seen. The reason for this is that Strategy 3 uses LAPs to reduce the number of TSPs and LAPs are a lot easier and faster to solve. So, there are several cases were the unbounded approach closes the gap in runtime by discarding TSPs that are otherwise discarded by the bound. In most cases TSPs discarded by the bound can also be discarded by LAP solving. The 3 marked datapoints are cases were this is not the case and the bound cannot be compensated. Figure 7b shows the time difference which in 42 is cases very close to zero with 3 datapoints being outliers on the far right. Again, the bound proves to be a reliable tool because even in the worst case scenario the bound approach performs very similar to the unbounded approach.

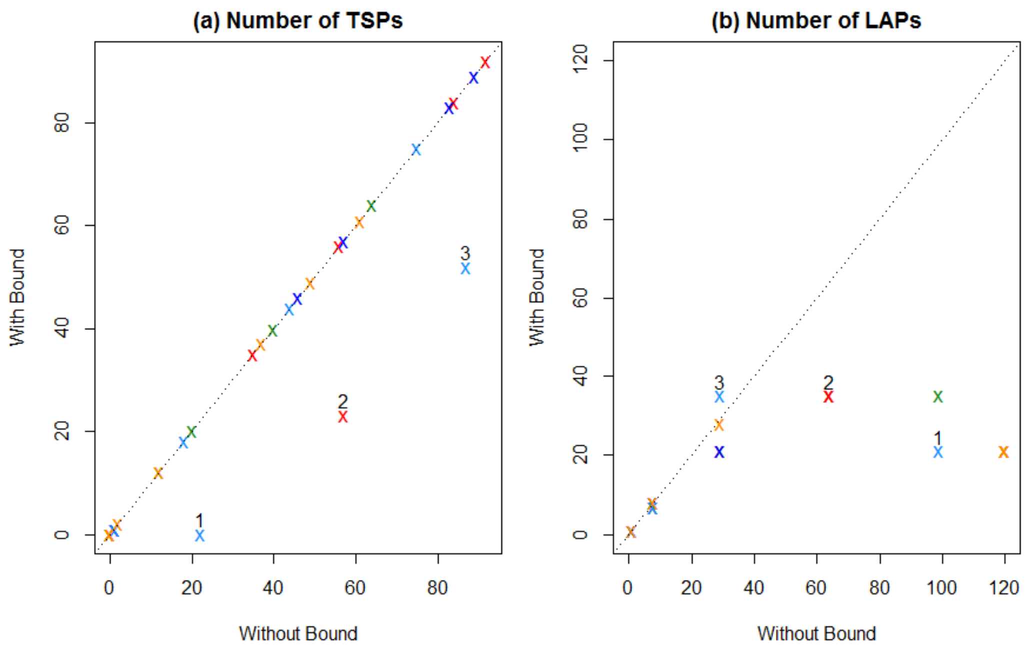

Figure 8: Number of TSPs and LAPs - Bound vs No Bound 2

Fig.8a
x-axis: Number of TSPs **without** bound
y-axis: Number of TSPs **with** bound
Fig.8b
x-axis: Number of LAPs **without** bound
y-axis: Number of LAPs **with** bound

On the dotted line (diagonal) both algorithms have the same number of TSPs (a) or LAPs (b), respectively.
Five different colors correspond to five different problem instances.

Figure 8a appears to be a mirror image of Figure 7a with transformed axis. This outlines the high impact and correlation of number of TSPs to solve and total runtime. So, reducing the number of TSPs will result an instant decrease in runtime. The datapoints 1,2 and 3 visualize this effect very clearly. Unlike Figure 8a, Figure 8b does not show much of a connection to Figure 7a (runtime). Solving LAPs takes a small amount of time compared to TSPs.

When comparing runtime for bound computation and LAP solving, it is important to consider that LAPs are known to be NP-hard and the bound, which consists of order statistics of row sums, can be computed in $\mathcal{O}(N^2)$. This may cause the gap between bounded and unbounded approach to widen for higher N.

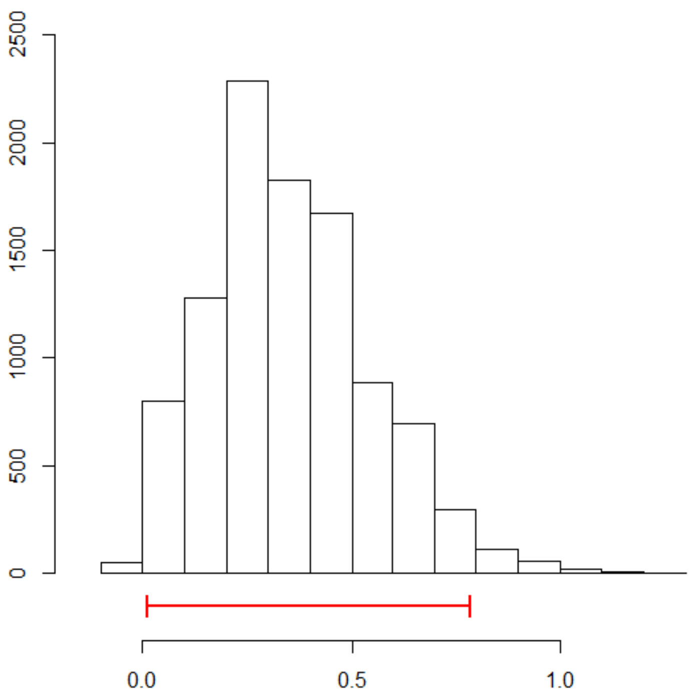

Figure 9: Bootstrap - Bound vs No Bound 2

y-axis: Bootstrap Frequencies
x-axis: Bootstrapped Mean [sec]
The red line is a 95% confidence interval for the average time reduction by using the bound from Prop. 1.

Like before in Figure 6, 10000 bootstrap resamples were simulated. Figure 9 shows that the resulting 95% confidence interval extends only over the positive section of the time axis, so it can be concluded that the bound approach also improves performance when using an LAP-involving strategy (Strategy 3). All conclusions in this section are of course limited to the circumstances under which the simulations were conducted.

5.2.2 Evaluation and Elimination

The main part of the algorithm follows a top-down approach. The starting point is determined by the criterion NV_{max} of the last section. Then all sets of nodes with size NV_{max} are generated and evaluated. Evaluation is done with the Linear Assignment solver from the R-package `adagio` and the `Concorde` TSP solver from the R-package `TSP`. Concorde is a very efficient exact solver for symmetric TSPs, which is a small limitation to the data that can be dealt with.

Sets that have a TSP tour with length smaller than or equal to the tour length limit C will be stored as candidate set along with the respective candidate tour. When all sets of size NV_{max} are processed, all sets of size $NV_{max}-1$ will be generated. After that there is an elimination step. All sets that are subsets of a previously found candidate set are discarded. For this purpose, the candidate sets will be used, since within a candidate set the elements (nodes) are increasingly sorted by their node number by construction. For each candidate set all subsets of size $NV_{max} - 1$ are generated. All of these subsets are eliminated from the current list. Fortunately, the list of all sets of size $NV_{max}-1$ as well as the subsets of candidate sets are ordered increasingly by construction, so there is no need to check different permutations.

From now on evaluation and elimination alternate until either all sets of some size are eliminated or the set size reaches zero. At this point all candidate tours are identified.

5.2.3 Linear Assignment

Another possibility to reduce the number of TSPs is using the formulation as Linear Assignment Problem (LAP). Both are very similar minimization problems, but the TSP has subtour elimination constraints whereas the LAP has none. It follows that the set of feasible tours of TSP is a subset of the set of feasible solutions of the LAP. Also, every LAP solution can be interpreted as a tour. Thus, it holds that for every set of nodes the optimal TSP solution is greater than or equal to the LAP solution.

The question of interest is, if there exists a tour to visit a specific set of nodes within a tour length smaller than or equal to C. Should the LAP solution be greater than C, then the question can clearly be answered with “no”. Otherwise the TSP has to be computed. Considering that LAPs are a lot easier to solve, this is a big advantage compared to solving the TSP for all cases. The next question that appears is when to stop computing LAPs.

There are three strategies that will be compared.

- The first strategy (S1) does not compute LAPs at all and only uses TSPs instead.
- The second strategy (S2) starts with solving LAPs until one set cannot be decided. It goes on with LAP solving until the current evaluation step is over and stops then.
- The third strategy (S3) starts like the second strategy but lasts one more additional evaluation step before stopping.

These strategies can be described by pseudo codes as follows.

Algorithm 2 Strategy 1

```
procedure
    $k \leftarrow \mathrm{NV}_{\max}$
    $candidate.tours \leftarrow$ empty list

loop:
    if $k = 0$ then return $candidate.tours$
    $subsets \leftarrow$ all subsets of n nodes of size k

    for i in $subsets$ do
        solve TSP for i
        if $TSP\ length \leq C$ then
            add TSP tour to $candidate.tours$

    $k \leftarrow k - 1$
    repeat $loop$
```

Algorithm 3 Strategy 2

```
procedure
    $k \leftarrow \mathrm{NV}_{\max}$
    $candidate.tours \leftarrow$ empty list
    $potential.candidate.tours \leftarrow$ empty list
    $use.LAPs \leftarrow$ true

loop 1:
    if $k = 0$ then go to LAP evaluation
    $subsets \leftarrow$ all subsets of n nodes of size k

    for i in subsets do
        solve LAP for i
        if LAP objective value $\leq C$ then
            add i to potential.candidate.tours
            $use.LAPs \leftarrow$ false

    $k \leftarrow k - 1$
    if $use.LAPs =$ true then repeat loop1
    else go to LAP evaluation

LAP evaluation:
    for i in potential.candidate.tours do
        solve TSP for i
        if TSP length $\leq C$ then
            add TSP tour to candidate.tours
    go to loop 2

loop 2:
    if $k = 0$ then return candidate.tours
    $subsets \leftarrow$ all subsets of n nodes of size k

    for i in subsets do
        solve TSP for i
        if TSP length $\leq C$ then
            add TSP tour to candidate.tours

    $k \leftarrow k - 1$
    repeat loop 2
```

Algorithm 4 Strategy 3

```
procedure
    $k \leftarrow \mathrm{NV}_{\max}$
    $candidate.tours \leftarrow$ empty list
    $potential.candidate.tours \leftarrow$ empty list
    $use.LAPs \leftarrow$ true
    $extra.round \leftarrow$ true

loop 1:
    if $k = 0$ then go to LAP evaluation
    $subsets \leftarrow$ all subsets of n nodes of size k

    for i in $subsets$ do
        solve LAP for i
        if LAP objective value $\leq C$ then
            add i to $potential.candidate.tours$
            $use.LAPs \leftarrow$ false

    $k \leftarrow k - 1$
    if $use.LAPs =$ true then repeat loop1
    else
        if $extra.round =$ true then
            $extra.round \leftarrow$ false
            repeat loop1
        else go to LAP evaluation

LAP evaluation:
    for i in $potential.candidate.tours$ do
        solve TSP for i
        if TSP length $\leq C$ then
            add TSP tour to $candidate.tours$
    go to loop 2

loop 2:
    if $k = 0$ then return $candidate.tours$
    $subsets \leftarrow$ all subsets of n nodes of size k

    for i in $subsets$ do
        solve TSP for i
        if TSP length $\leq C$ then
            add TSP tour to $candidate.tours$

    $k \leftarrow k - 1$
    repeat loop 2
```

5.2.4 Runtime Analysis

Since all strategies have the same output quality, the strategies are assessed by looking at their runtime. The results of the first exploratory simulations are illustrated in Figure 10 and Table 4. These also use a variety of constraint levels. For each number of nodes 3000 simulations as described in section 4.2 were conducted, TSP solutions computed and the maximum was stored. The constraint levels used in Figure 10 and Table 4 are 10%, 30%, 50% and 70% of these individual maxima. For a better visualization, these constraint levels have a horizontal displacement. The observed times for N number of nodes with (10, 30, 50, 70) % of the particular maximum are plotted at (N-0.15, N-0.05, N+0.05, N+0.15) respectively.

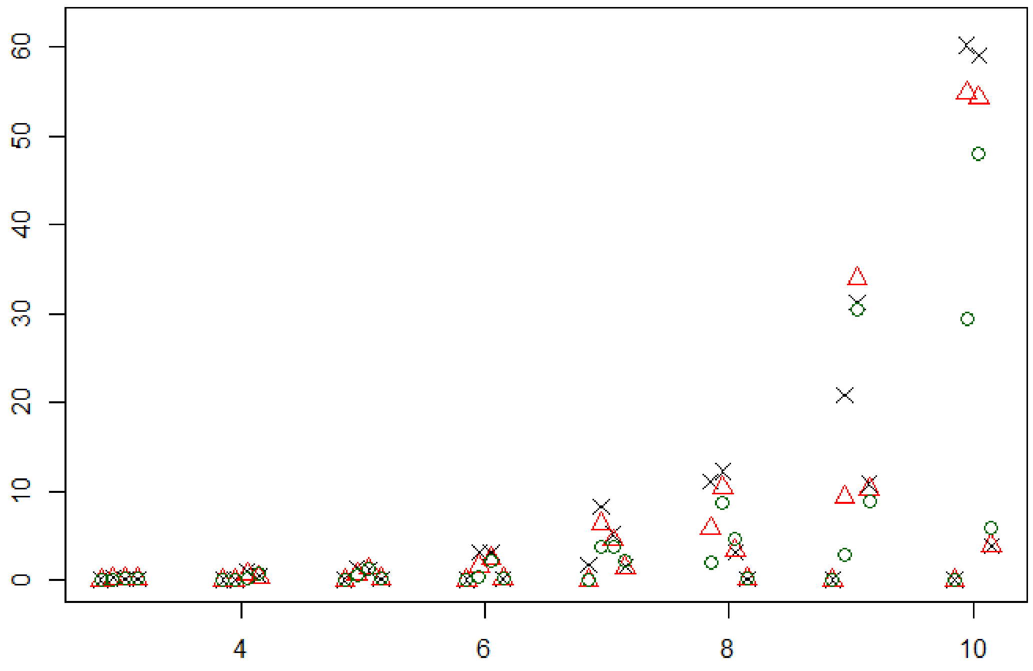

Figure 10: Runtime Comparison for 3 Strategies 1

y-axis: Runtime in seconds
x-axis: Number of nodes with small displacement for different constraint levels
Strategies 1 (black x), 2 (red triangle) and 3 (green circle)

Table 4: Mean Runtime for 3 Strategies

S1	S2	S3
7.49s	6.44s	4.91s

It turns out that there is no dominant strategy that performs best in all cases but S3 has the best overall performance in this first survey. To evaluate this hypothesis, it is tested on a new simulation dataset.

The test is designed as follows

- For each possible combination of $N \in \{3, 4, 5, 6, 7, 8, 9, 10\}$ and constraint levels of $(10, 30, 50, 70)\%$ of the corresponding observed maximum TSP tour 7 problem instances are generated and solved using all 3 strategies (analogously to pilot survey).
- Sample size equals 224 (per strategy).

  (8 numbers of nodes x 4 constraint levels x 7 iterations)
- Nullhypotheses are $mean(S1_{time} - S3_{time}) <= 0$ and

  $mean(S2_{time} - S3_{time}) <= 0$.
- Alternative Hypotheses are $mean(S1_{time} - S3_{time}) > 0$ and

  $mean(S2_{time} - S3_{time}) > 0$, which translates to S3 being superior to S1 or S2, respectively.
- Testing on the aggregated data $S1_{time} - S3_{time}$ and $S2_{time} - S3_{time}$ results in a design with independent observations.
- After evaluating assumptions for classic test procedures such as normality or symmetry, it turned out they were not satisfied. Additionally, the data is not identically distributed, so a robust and non-parametric method was chosen, namely a bootstrap confidence interval.
- Level of significance is set to 0.05.

Figure 11 and Figure 12 are visualizing the testing data. In Figure 11 the raw time distribution can be seen, while Figure 12a presents the time differences of S1 and S3 and Figure 12b presents the time differences of S2 and S3.

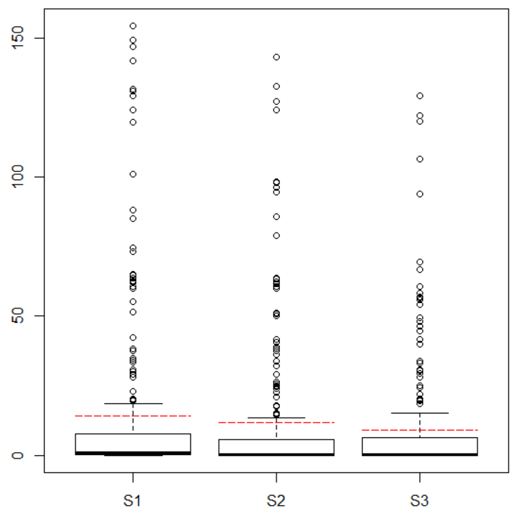

Figure 11: Runtime Comparison for 3 Strategies 2

y-axis: runtime in seconds
red line: mean computation time

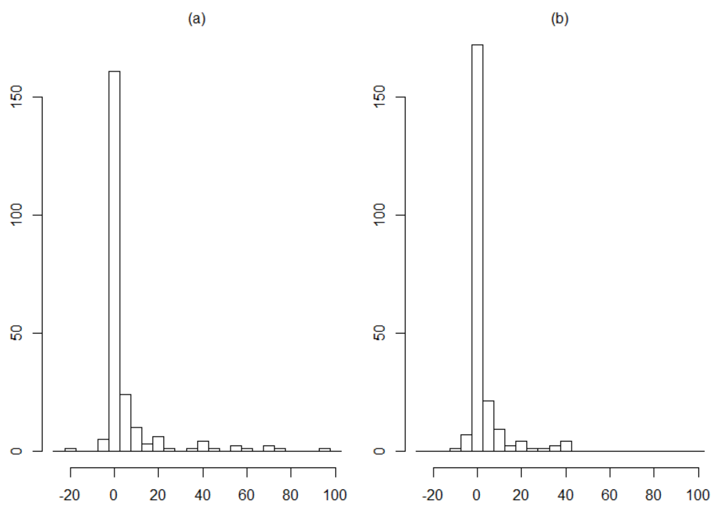

Figure 12: Runtime Differences

y-axis: Runtime in seconds
12a: $S1_{time} - S3_{time}$
12b: $S2_{time} - S3_{time}$

Table 5: Selected Statistics of Testing Data

statistic	S1	S2	S3	S1-S3	S2-S3
mean	14.22s	11.69s	9.26s	4.96s	2.43s
median	1.01s	0.19s	0.36s	0.00s	0.00s
sd	31.31s	26.89s	21.58s	14.09s	7.60s
min	0.00s	0.00s	0.00s	-19.54s	-11.83s
max	154.27s	143.04s	129.08s	97.24s	42.45s
p-value of KS test*	$< 10^{-10}$	$< 10^{-10}$	$< 10^{-10}$	$< 10^{-10}$	$< 10^{-10}$

* Kolmogorov-Smirnov test to assess if data is normally distributed

As mentioned before, the data violates assumptions of standard test procedures. The time measures are not identically distributed because measures on problems with more nodes have higher a higher expected value and a higher variance. Nevertheless, it is necessary to use a variety of node numbers to get representative and general results. It was expectable that the computation times of the pure strategies do not follow a symmetric distribution, since they are naturally truncated at zero. Figure 11 displays this right-skewed distribution. Figure 12 also visualizes a naturally right-skewed distribution. By design of S3 it can only be outperformed by the duration of solving a few LAPs, which can quickly be done. On the contrary S3 can have a lower number of TSPs, which are a lot harder to solve, so there is a lot of potential for S3 to outperform S1 and S2.

None of the data vectors follows a normal distribution. This is of course closely linked to the non-fulfilled symmetry property. The corresponding test results are listed in the last row of Table 5.

In Figure 13 the results of 10000 bootstrap iterations each are shown in histograms with added 95 % confidence intervals.

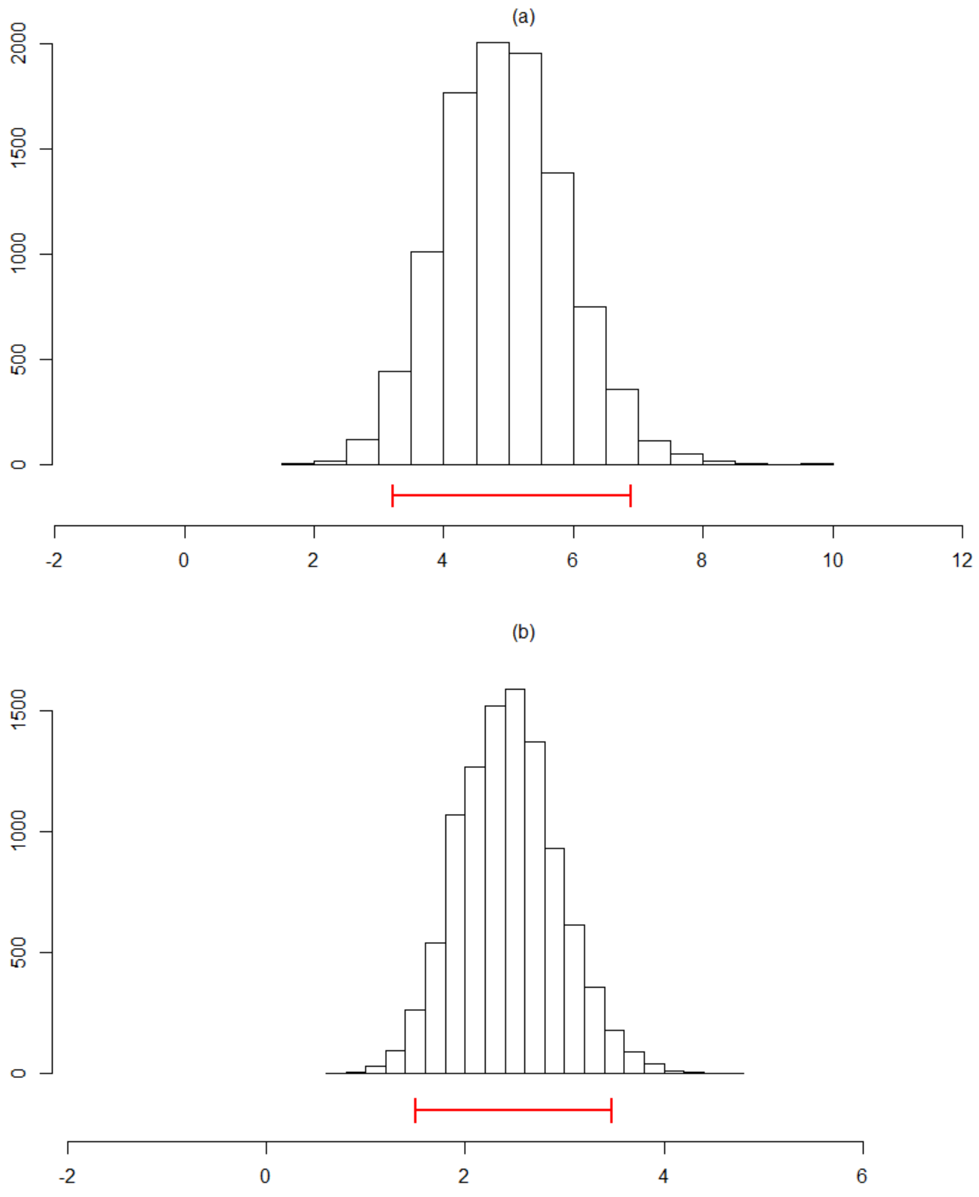

Figure 13: Bootstrap Results

y-axis: Frequencies
x-axis: Means of bootstrap simulations
red: 95 % confidence interval for the true mean
13a: $S1_{time} - S3_{time}$
13b: $S2_{time} - S3_{time}$

The construction of the confidence intervals was done by simply taking the 0.025 and 0.975 quantiles of the resampled means. This is a natural way to obtain confidence intervals from bootstrap methods. Now these confidence intervals are used for hypothesis testing. Both confidence intervals are strictly > 0, so it is not plausible, that the mean time difference of the strategies is ≤ 0. Therefore, the null hypotheses

- $mean(S1_{time} - S3_{time}) \leq 0$ and
  $mean(S2_{time} - S3_{time}) \leq 0$

  can be rejected in favor of the alternative hypotheses

- $mean(S1_{time} - S3_{time}) > 0$ and
  $mean(S2_{time} - S3_{time}) > 0$.

It follows that S3 is superior to both S1 and S2 and will therefore be used in the algorithm.

5.3 Determine Optimal Schedule

In the first part of the algorithm important variables in particularly the number of time steps T, the vector of weights w and the decay factor q were not used since they are independent of the set of candidate tours. In this section, these variables loom large, whereas distance matrix and tour length limit are not required anymore.

As mentioned before, determining the optimal schedule is quite difficult. Small changes for weights, decay factor or the set of candidate tours can lead to completely different solutions. Additionally, it is hard to evaluate parts of the algorithm without knowing the exact schedule, which would be done in a branch and bound approach. While collecting a high value in one time step is profitable, it deteriorates the outlook for the next one. Still the total uncollected value is not a sufficient measure because its distribution is also of importance.

However, a rule can be established. In order to do so, the following two cases based on the set of candidate tours are distinguished.

(i) There are multiple best tours.

(ii) There is a unique best tour.

A candidate tour is considered a best tour if it performs better than or equal to all other candidate tours in the classic orienteering problem ($T = 1$). Let n from now on denote the number of candidate tours. Case (i) is the easier one for enumeration as in the optimal schedule consecutive tours are always distinct. Should a schedule contain two consecutive identical tours, then one can always construct a similar schedule that outperforms it. The reason for this property is the structure of the z_i's and that recently visited nodes decrease in value, while others increase.

Case (ii) contains a special case namely $n = 1$. If there is only one candidate tour, then of course the optimal schedule will consist of T copies of that one tour. Even for $n > 1$ and $T > 1$ the optimal schedule can consist only of copies of one tour or at least have some consecutive identical tours. This occurs in problems that have one very dominant candidate tour and a few other less viable ones that visit outlier nodes or in problems that have a high decay factor. Still these problems have in common, that if there appear consecutive identical tours in the optimal schedule, then these must be copies of the unique best tour.

Theoretically, there is a third case in which the set of candidate tours is the empty set but hence the problem is trivial.

It is important to check if this rule does not violate the claim to find the exact solution. Therefore, superior schedules are constructed for the omitted ones. As mentioned in chapter 2, the schedule score consists of two components, namely the basic score of the included tours and a penalty, which punishes large overlaps of the tours. There are two intuitive approaches to improve the score of a schedule with consecutive identical tours (also called pairs). The first move consists of swapping the first tour of the pair with the tour right before the pair. This does not change the basic score, because the same nodes (with respective multiplicities) are included, but it improves the overlap structure because the pair nodes are now separated. This move can always be performed unless the pair consists of the first two nodes of the schedule. In this case, the second move is performed. It replaces the first node of a pair with the best tour. The basic score increases, because a not-best tour is replace by the best tour, and the overlap structure cannot worsen because it is locally already at the lowest level. This moves can be iterated and eventually lead to schedules without pairs of not-best tours. After the first iteration, one has to exclude the best tours (X) from swapping and replacing. Table 6 and 7 list the omitted schedules and provides superior schedules. Due to the increasing number of possibilities, only schedules for $T \leq 5$ are covered.

To verify the dominance, the comparison was formulated as an LP. Therefore, the tours have to be divided into $2^{\#\text{ of tours}}$ pairwise disjoint sets that represent all possible intersections of tours. The objective function is the difference of the score of the discarded schedule and the superior schedule. A block of equality constraints ensures the subsets form a proper partition of the node space. For instance, all sets formed by intersections with tour A have to sum up to A. Inequality constraints are employed to express the superiority of the best tour. In this setting a negative objective value means the pair-avoiding tour has a higher score, while a positive objective value means the pair-including tour has a higher score. The LP maximizes over all possible overlaps between all tours. If the optimal solution is ≤ 0, then there exists no configuration of nodes such that the tour with consecutive identical tours is superior. All LPs had an optimal solution ≤ 0 and many even had all coefficients of the objective function ≤ 0.

In case (ii) (multiple best tours) the process is analogously, but in case of multiple insertions of X, different best tours are used to produce even better scores and consequently omit more schedules.

Table 6: Omitted Schedules T=2,3,4

Discarded Schedule	Superior Schedule
T=2	
AA	XA
T=3	
AAB	XAB
ABB	BAB
AAA	XXA
T=4	
AABC	XABC
ABBC	BABC
ABCC	ACBC
AABB	ABAB
ABBA	BABA
AAAB	XXAB
AABA	XABA
ABAA	XABA
BAAA	XABA
AAAA	XXXA

A,B,C and D denote tours that are pairwise distinct and can be identical to X unless they are part of a pair in the left column. X denotes the best tour.

Table 7: Omitted Schedules T=5

Discarded Schedule	Superior Schedule
T=5	
AABCD	XABCD
ABBCD	BABCD
ABCCD	ACBCD
ABCDD	ABDCD
AABBC	ABABC
AABCB	ABACB
ABACC	ABCAC
ABBAC	BABAC
ABBCA	BABCA
ABBCC	BACBC
ABCCA	ACBCA
ABCCB	ACBCB
AAABC	XXABC
AABAC	XABAC
AABCA	XABCA
ABAAC	XABAC
ABCAA	ABACA
ABBBC	XBABC
ABBCB	BABCB
ABCBB	BABCB
ABCCC	CACBC
AAAAB	XXXAB
AAABA	XXABA
AABAA	XXABA
ABAAA	XXABA
ABBBB	XXBAB
AAAAA	XXXXA

The algorithm generates all possible combinations for schedules that consist only of candidate tours and then omits instances that include consecutive identical tours unless they are copies of a unique best tour. Subsequently, the remaining schedules are evaluated and the maximizer (not necessarily unique) is returned.

Clearly, the runtime of the second part heavily depends on n and T. The number of possibilities can be computed by the following formulas and is illustrated by Figure 14.

$$\text{Total (unfiltered)} = n^T \quad \text{for } T \geq 1;\ n \geq 1 \tag{26}$$

$$\begin{aligned}&\text{Case (i) (filtered)} = \\ &n(n-1)^{T-1} + n^{T-2} + (T-2)(n-1)^{T-2} \\ &\text{for } 2 \leq T \leq 5;\ 1 \leq n \leq 15\end{aligned} \tag{27}$$

$$\text{Case (ii) (filtered)} = n(n-1)^{T-1} \quad \text{for } T \geq 2;\ n \geq 2 \tag{28}$$

Unlike (26) and (28), equation (27) has no straightforward combinatorial interpretation. However, the formula was tested on all points *within the given range* and returned the right number of schedules for all input values.

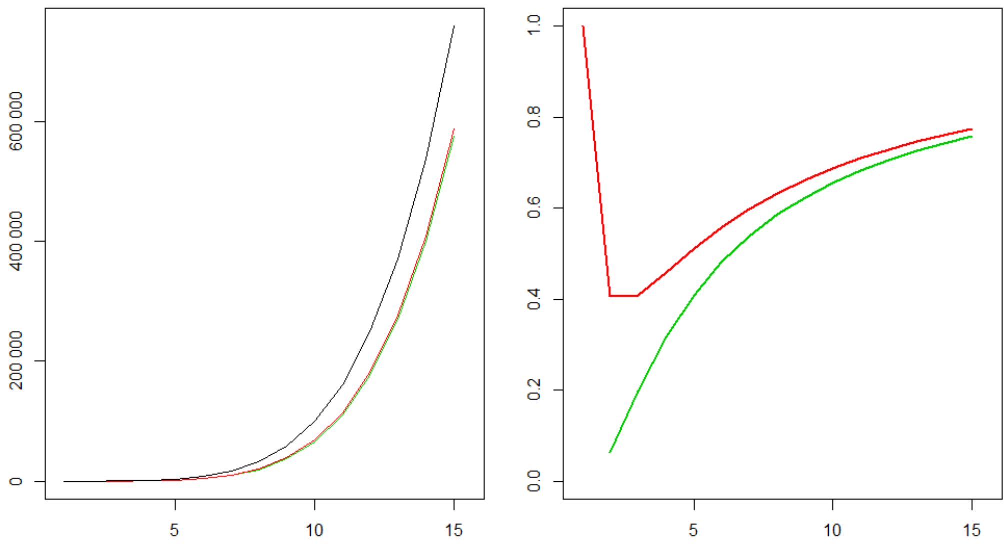

Figure 14: Number of Possibilities for Schedule Optimization (T=5)

x-axis: number of candidate tours
(a) Total number of possibilities unfiltered (black), case (i) (red) and case (ii) (green)
(b) Relative share of possibilities: case (i) (red) and case (ii) (green), both divided by unfiltered

6 Application Example

6.1 Data Description

To exemplify the results of the algorithm, consider the following problem setting. A company has potential customers in all Austrian provincial capitals and Munich. Due to its central position Salzburg serves as depot. The expected revenue per visited city is computed by a degressive proportional function (see (29)) based on the number of inhabitants [25]. Distances from one city to another were computed by calculating the road distances of the fastest tour with data of Google Maps [26] and then rounded to multiples of full kilometers. Different tour length limits ranging from 500 to 1000 km provide a comprehensive view of how solutions look like and change in dependence on the parameter. $T = 5$ represents a five-day workweek. Inhabitants are measured in multiples of 1000 and weights are rounded to whole numbers.

For a first view on the problem, the decay factor q is set to 0.9 which is a rather slow decay and makes a diverse schedule more attractive. In this application example, q can be interpreted as probability that the demand of a potential purchase is still available after one day. In other words everyday there is a chance of $1 - q$ for a potential customer to lose interest in the purchase or buy at another competitor. The z_i^t are the time-dependent multipliers that state how much more revenue one can expect from this node compared to the starting value of the node. $z_i^1 = 1$ for all i and will transform to $z_i^2 = 1 + q$ if node i is not visited at time step $t = 1$. Consequently, there is the "1" for the fresh demand of the second day and "q" which is the rest of the demand of the first day. Conversely, if node i does get visited at time step 1, then there is no left over demand from day 1 and the demand is reset to $z_i^2 = 1$, which is the fresh demand of day 2. Should city i get no visit for the first 4 days, then $z_i^5 = 1 + q + q^2 + q^3 \overset{q=0.9}{=} 4.0951$ so there will be demand roughly equal to about 4 consecutive visits. Hence, there is an incentive to plan diverse tours to meet the accumulating demand. For $q' = 0.3$ the multiplier would only be $z_i^5 = 1 + q' + q'^2 + q'^3 \overset{q'=0.3}{=} 1.4251$ so there is only very little reward for including unvisited nodes. This would lead to tours that visit nodes with a high weight every day, whereas nodes with small weight and far away from depot would get no visits at all. The choice of q is of course important and users of this algorithm should do some kind of pilot study to estimate q. In the section Results a variety of values for q are compared and differences are highlighted.

The growth/decay model will simplify to a linear growth in z_i^t ($q = 1$) or to constant z_i^t ($q = 0$) for appropriate values of q which gives adaptability to this model to describe the underlying process as good as possible.

$$\text{weight(City)} = \text{Round}(\sqrt{\text{Inhabitants}} + 10) \tag{29}$$

Table 8: City Details

City	**Abbreviation**	**Inhabitants**	**Weight**
Salzburg (Depot)	S	151	22*
Bregenz	B	29	15
Eisenstadt	E	14	14
Graz	G	280	27
Innsbruck	I	131	21
Klagenfurt	K	99	20
Linz	L	201	24
Muenchen	M	1450	48
St.Poelten	P	53	17
Vienna	V	1840	53

* Salzburg's weight is only computed for visualization reasons. For problem solving Salzburg has no weight at all in its role as depot.

6.2 Network

The problem is visualized in different levels of abstraction. To begin with, the cities' locations are determined by their geographic coordinates [27] [28]. Edges are only drawn if they match the real road network and are part of at least one tour. This can be seen in Figure 15.

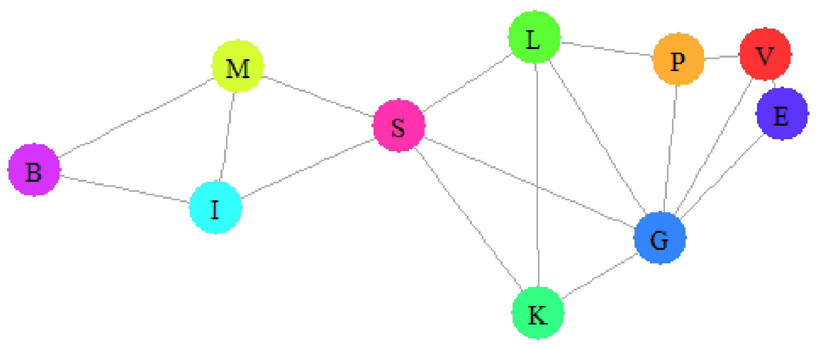

Figure 15: Geographical Problem Representation

Figure 15 represents the geographic arrangement with the edge lengths depicting the airline. For the problem as defined it is more intuitive to plot the actual road distance instead. This leads to an optimization problem called multidimensional scaling (see Kruskal 1964 [13]), which can be solved by using the R-function cmdscale. The exact solution for this problem typically has the dimension of the number of nodes − 1, so the two-dimensional solution should be seen as an approximation. To account for population size, the cities' radii correspond to the weights defined in (28). Figure 16 therefore gives a less geographic but more problem-oriented point of view.

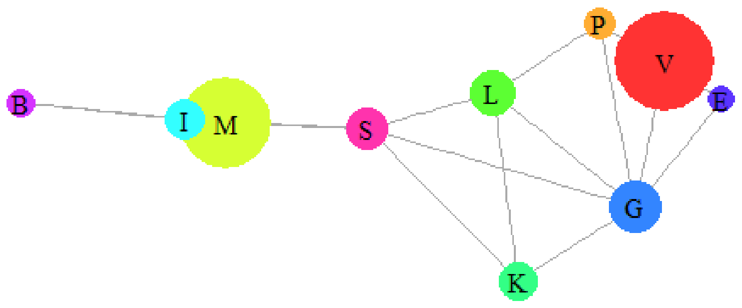

Figure 16: Problem-Oriented Representation

When comparing Figure 16 with Figure 15, one can clearly see that Vienna and Linz are moving towards the center and on the other hand St. Pölten is moving outwards meaning that Vienna and Linz have a good transport connection.

6.3 Results

In Table 9, optimal schedules (Day 1-Day 5) and final objective values (Obj.val) for the different constraint levels as well as some algorithm details. The first part of the algorithm consists of identifying the set of candidate tours, which is measured by time1. This quantity is mainly influenced by the number of solved LAPs (# Assign) and the number of solved TSPs (# TSP). The second part enumerates and evaluates possible schedules and is measured by time2. Its runtime has a strong relation to the number of candidate tours (n) and the number of time steps ($T = 5$). Figure 17 and 18 illustrate an example schedule.

Table 9: Application Example Results Summary (q=0.9)

	500km	600km	700km	800km	900km	1000km
Day 1	LP	K	BM	IBM	KGVPL	LPVEGK
Day 2	IM	IM	EV	LPVE	IBM	LIBM
Day 3	K	G	GK	LGK	LPVEG	LPVEGK
Day 4	LP	PV	IM	IBM	IBM	LIBM
Day 5	IM	LM	LPV	LPVG	KGVPL	LPVEGK
Obj.val	524.4	693.4	831.5	954.8	1026.7	1068
Best Tour	IM	LM	LPV	LPVG	KGVPL	LPVEGK
time1 [sec]	1.1	6.2	13	13.9	22	29.4
time2 [sec]	0.1	1.6	29.7	6.9	675	118.4
# Assign	120	210	252	252	210	120
# TSP	8	54	131	143	234	303
n	3	6	10	8	15	12

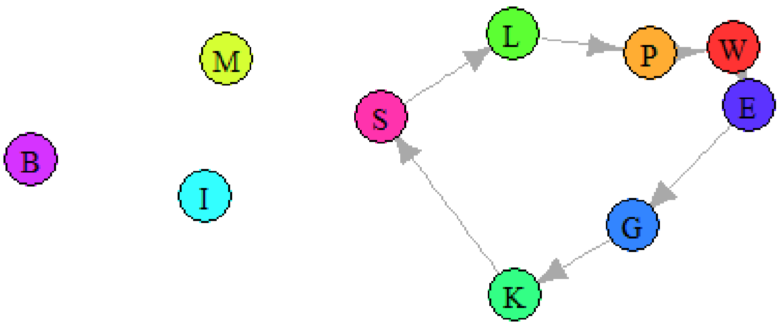

Figure 17: Example Tour 1

This tour is scheduled for Day 1, 3 and 5 for C=1000km.
For C=1000km this is the unique best tour.

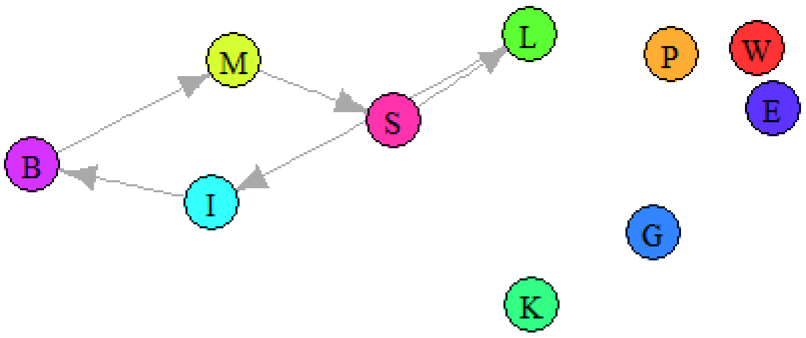

Figure 18: Example Tour 2

This tour is scheduled for Day 2 and 4 for C=1000km.

In the schedules with lower constraint levels Munich is very present, while for the higher constraint levels Vienna is visited most. The reason for it is that both cities have high weights due to their large number of inhabitants and Munich profits from its closeness to Salzburg, whereas Vienna profits from the fact that area east of Salzburg has more cities overall.

Interestingly, all schedules finish with the unique best tour. This seems reasonable, since after the last time step all left-over values expire. Still, it could happen that the unique best tour does not even appear in the optimal schedule. Best tours are determined by their short-term properties and do not necessarily have good long-term properties. An example for a problem instance, in which the best tour does not appear in the optimal schedule, is mentioned in chapter 7.

When comparing the time measures with the most influential factors, one can see a perfect rank correlation between time1 and # TSP and between time2 and n. Time spent on using LAPs is negligible and should mainly be seen as a way to decrease # TSP.

6.3.1 Parameter Sensitivity

The results from Table 9 assumed a static decay factor $q = 0.9$. This section will deal with multiple values for q and provides a possibility to bound the optimal objective value even if q is unknown. The tour length limit C, which also impacts the outcome a lot, will take the same values as before.

Table 10 compares the optimal solutions for all combinations of $C \in \{500, 600, 700, 800, 900, 1000\}$ and $q \in \{0.1, 0.3, 0.9, 0.99\}$. 0.99 was chosen instead of 1 because for $q = 1$ an anomaly occurs which typically produces a high number of solutions with identical optimal objective value. For $q = 0.99$ the optimal schedule is unique in most cases and is also included in the set of optimal schedules for $q = 1$. So, using $q = 0.99$ instead of $q = 1$ gives similar results to $q = 1$, ranking all solutions by gradient and choosing the flattest. The anomaly that produces the ties is that for $q = 1$ there is no decay so the collected reward for a specific node only depends on the last visit, where all accumulated profit can be picked up without discount. All visits of the same node before this last visit can be permutated or even skipped. Unlike Table 9, it omits some lines because the best tour, # Assign, # TSP and n are completely identical to the values in Table 9 and time1 and time2 have the same distribution as before but of course have slightly different values for repeated measurements.

Table 10: Application Example Results for different values of q

	500km	600km	700km	800km	900km	1000km
			q=0.1			
Day 1	IM	LM	LPV	LPVG	KGVPL	LPVEGK
Day 2	IM	PV	LPV	LPVG	KGVPL	LPVEGK
Day 3	IM	LM	LPV	LPVG	KGVPL	LPVEGK
Day 4	IM	PV	LPV	LPVG	KGVPL	LPVEGK
Day 5	IM	LM	LPV	LPVG	KGVPL	LPVEGK
Obj.val	345	384.4	470	605	705	775
			q=0.3			
Day 1	IM	LM	LPV	LPVG	KGVPL	LPVEGK
Day 2	LP	PV	IM	IBM	MLV	MLVP
Day 3	IM	IM	LPV	LPVG	KGVPL	LPVEGK
Day 4	LP	PV	IM	IBM	MLV	LIBM
Day 5	IM	LM	LPV	LPVG	KGVPL	LPVEGK
Obj.val	355	442	517.8	654	740.2	816.4
			q=0.9			
Day 1	LP	K	BM	IBM	KGVPL	LPVEGK
Day 2	IM	IM	EV	LPVE	IBM	LIBM
Day 3	K	G	GK	LGK	LPVEG	LPVEGK
Day 4	LP	PV	IM	IBM	IBM	LIBM
Day 5	IM	LM	LPV	LPVG	KGVPL	LPVEGK
Obj.val	524.4	693.4	831.5	954.8	1026.7	1068
			q=0.99			
Day 1	LP	K	EV	IBM	KGVPL	LPVEGK
Day 2	IM	IM	BM	LPVE	IBM	LIBM
Day 3	K	G	GK	LGK	LPVEG	LPVEGK
Day 4	LP	PV	IM	IBM	IBM	LIBM
Day 5	IM	LM	LPV	LPVG	KGVPL	LPVEGK
Obj.val	564.4	773.5	920	1021.4	1077.3	1106.7

As a reminder, q is the fraction of the current value of a node that is carried over to the next time step if this node is not visited during the current time step. Consequently, a high value of q describes a slow decay and a small value of q describes a fast decay. It makes sense that for $q = 0.1$ the schedules tend to repeat one tour throughout the whole time span because the fast decay works similar to a reset of the nodes. Conversely, the schedules for $q = 0.99$ tend to use a broad variety of nodes (and consequently tours) because the slow decay makes ignoring nodes for the whole time span very unappealing.

A property that all tested parameter combinations have in common, is that they all have the unique best tour at the last time step in the optimal schedule. Intuitively, it makes sense to use a very rewarding tour at the end when the reward piled up. However, this is a tendency and not a general property. As later mentioned in chapter 7, there exist problem instances that do not use the unique best tour in the optimal schedule. So fixing the last time step to the best tour could be a good approach for heuristics but it might discard the optimal schedule in rare cases.

At first glance it is not obvious if optimal schedules found for one value of q will also perform good for others. Figure 19 compares all schedules found with each other for the whole range of $q \in [0, 1]$. In this image the differences on the left side of the plots are a lot smaller than the differences on the right side of the plots. This suggests that it is safer to overestimate q when searching for a good schedule because the solutions found for large q also perform relatively well for small q but on the other side optimizing for small q will lead to results that perform poorly for large q. The latter case holds true especially for the constant lines which are schedules that repeat one tour throughout the whole time span and therefore typically ignore some nodes completely.

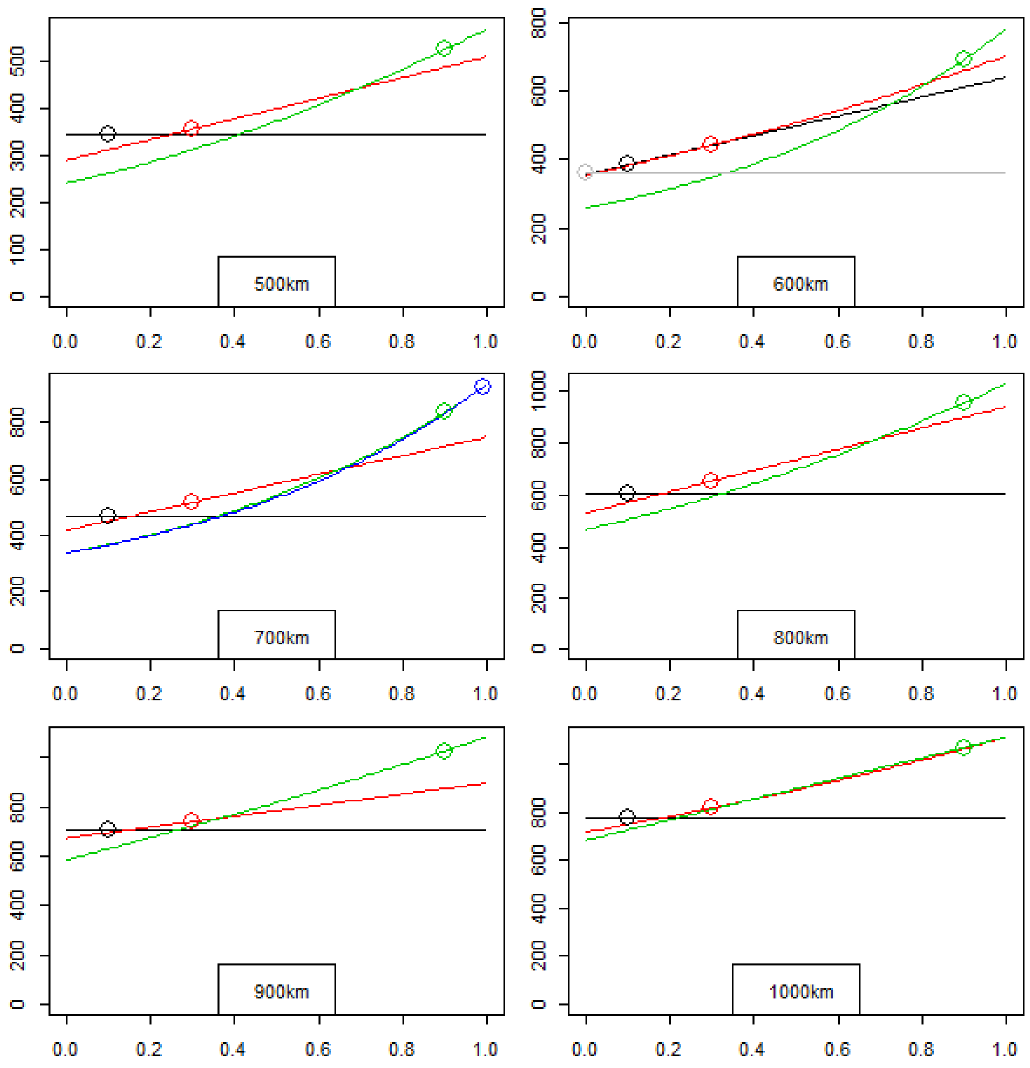

Figure 19: Objective Value in Dependence of C and q 1

y-axis: Objective Value
x-axis: q
In each plot the colors represent the optimal solutions for given C and q with the color code being gray (q=0), black (q=0.1), red (q=0.3), green (q=0.9), blue (q=0.99). Gray and blue were omitted if the corresponding solutions were identical to black or green, respectively. The circles also denote the value of q for which the solution is optimal.
The box at the bottom indicates the constraint level C.

Figure 20 combines the optimal schedules of the grid points for q to use the most rewarding schedule for given q, where one line represents one constraint level. Of course, intervals between the grid points could theoretically be optimized by a schedule that is not optimal for any of the grid points and hence not used in this graph. If such a schedule should exist, it can only surpass the objective value of the plotted line by a small amount. This will be discussed at Figure 21. The lines in Figure 20 are lower bounds for the objective value, which are tight at the grid points. The graph also includes the total minimum (0) and maximum (1195) objective value which are attained for appropriate C but are independent of q. They are attained by visiting all cities every day or no cities at all, respectively.

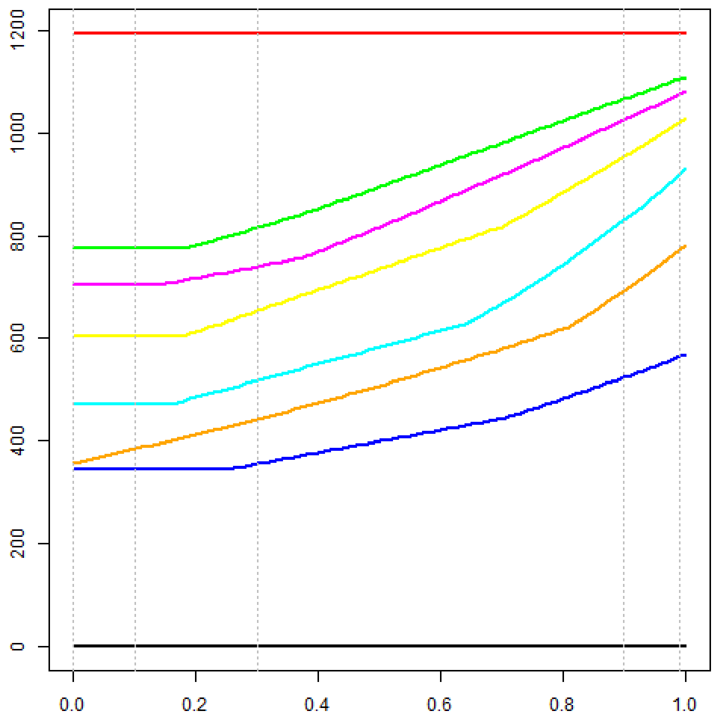

Figure 20: Objective Value in Dependence of C and q 2

y-axis: Objective Value
x-axis: q
This image depicts the optimal combination of all found solutions at $q \in \{0, 0.1, 0.3, 0.9, 0.99, 1\}$, which are also highlighted by the vertical lines. The color code is black (C<62), blue (C=500), orange (C=600), turquoise (C=700), yellow (C=800), magenta (C=900), green (C=1000), red (C≥1611).

For the next calculations, it is necessary to recall the definition of the objective function and variables occurring from chapter 4. They are listed in (30). These will make it possible to derive some mathematical properties to create upper bounds for the optimal objective value between the grid points.

$$
\begin{aligned}
&\text{objective function: } \sum_{t=1}^{T}\sum_{i=1}^{N} w_i \cdot z_i^t \cdot y_i^t \to max\\
&y_i^t = \begin{cases} 1 & \text{i is visited in time step t} \\ 0 & \text{otherwise} \end{cases}\\
&\qquad \text{for} \quad i = 1, ..., N; \; t = 1, ..., T\\
&z_i^t = \begin{cases} 1 + q \cdot z_i^{t-1} & y_i^{t-1} = 0 \\ 1 & y_i^{t-1} = 1 \quad \text{or} \quad t = 1 \end{cases}\\
&\qquad \text{for} \quad i = 1, ..., N; \; t = 1, ..., T\\
&z_i^t \in \left\{1, 1+q, 1+q+q^2, \ldots, \sum_{k=0}^{T-1} q^k\right\} \text{ for } \; i = 1, ..., N; t = 1, ..., T\\
&w_i \in \mathbb{R}^+ \quad \text{for} \;\; i = 1, ..., N
\end{aligned}
\tag{30}
$$

Let $O_C(q)$ denote the optimality function which for fixed C assigns the optimal objective value to an input q. This function is the pointwise maximum of all possible schedules' objective values.

$$
\begin{aligned}
O_C(q) &= \max_{(4)-(9) \text{ hold}} \sum_{t=1}^{T}\sum_{i=1}^{N} w_i \cdot z_i^t(q) \cdot y_i^t \\
&= \max_{s \,\in\, \text{feasible schedules}} \sum_{t=1}^{T}\sum_{i=1}^{N} w_i \cdot z_i^t(q, s) \cdot y_i^t(s)
\end{aligned}
\tag{31}
$$

As (31) outlines, the value of q influences the z_i^t directly and the y_i^t indirectly by making different schedules optimal. For a fixed schedule s, w_i and y_i^t are non-negative constants and the z_i^t are monotone and continuous in q. Finite sums are also monotone and continuous functions so for a fixed schedule the objective value is monotone and continuous in q. Furthermore, $z_i^t(q)$ is twice continuously differentiable in q. $z_i^t(q)$ is a polynomial in q with only positive coefficients so the second derivative is ≥ 0. This makes $z_i^t(q)$ a *convex* function. Multiplying convex functions with non-negative constants yields convex functions, as well as summing up a finite number of convex functions. Thus, the objective value for a fixed schedule is a monotonous, continuous and convex function. None of these properties is lost when taking the maximum over a finite set, hence $O_C(q)$ is also a monotonous, continuous and convex function.

In an application example, where distances d_{ij}, weights w_i and constraint level C are known but q is unknown, $O_C(q)$ is unknown, but can be evaluated for every q by solving the MPOP for this specific q. If $O_C(q)$ is known at a small number $G \geq 1$ of grid points $g_1 \leq ... \leq g_G$ then the resulting schedule(s) as well as their pointwise maximum are all lower bounds for $O_C(q)$. For the lower bounds, no additional properties are required.

$$\begin{aligned} O_C(q) &= \max_{s \,\in\, \text{feasible schedules}} \sum_{t=1}^{T} \sum_{i=1}^{N} w_i \cdot z_i^t(q, s) \cdot y_i^t(s) \\ &\geq \max_{\substack{s \,\in\, \text{feasible schedules} \\ \text{s optimal at a grid point}}} \sum_{t=1}^{T} \sum_{i=1}^{N} w_i \cdot z_i^t(q, s) \cdot y_i^t(s) \end{aligned} \tag{32}$$

For the upper bounds, there are three different cases depending on the location of the value of q at which $O_C(q)$ needs to be bounded.

(i) q is smaller than the smallest grid point g_1.

(ii) q lies between two consecutive grid points.

(iii) q is larger than the largest grid point g_G.

In case (i) it is possible to create a bound using monotonicity. $O_C(q)$ is monotonously nondecreasing, so $q < g_1$ directly implies $O_C(q) \leq O_C(g_1)$.

Case (ii) is the main case. Say, the two consecutive grid points are g_i and g_{i+1} and because q lies between them it holds that $g_i < q < g_{i+1}$. By convexity of $O_C(q)$ it follows that $O_C(q) \leq \alpha \cdot O_C(g_i) + (1 - \alpha) \cdot O_C(g_{i+1})$ for appropriate α such that $q = \alpha \cdot g_i + (1 - \alpha) \cdot g_{i+1}$.

If q lies in case (iii) then constructing a tight upper bound is complicated. One possibility is to compare it to the perfect schedule of visiting all nodes at all time points (red line in Figure 20). Another option is to formulate a bound on the highest possible slope of $O_C(q)$ by bounding the derivative of $z_i^t(q)$. Both methods will typically lead to very loose upper bounds and are not helpful. A better approach is choosing the largest grid point equal to $g_G = 1-\epsilon$ for ϵ very small. If ϵ is sufficiently small, then the schedule optimizing $O_C(1-\epsilon)$ will be identical to one of the set of schedules optimizing $O_C(1)$ because for a fixed schedule the objective value in dependence of q is continuous. Having one grid point near 1 also has an advantage for the quality of the lower bounds. Should q be equal to one of the grid points, creating bounds is of course obsolete because the exact value of the function is already known.

To illustrate the application of bounds, both lower and upper bounds are computed and plotted for $C = 500$ km in Figure 21.

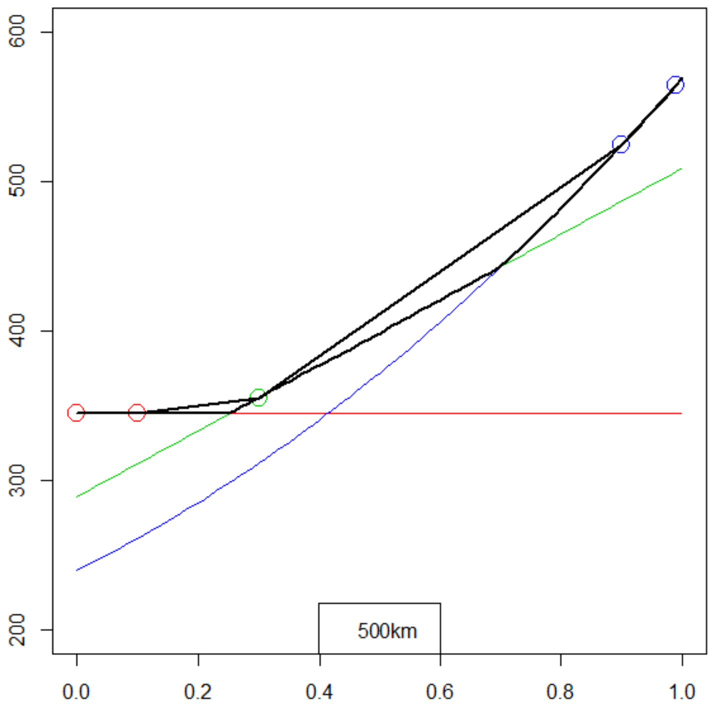

Figure 21: Bounds for Optimal Objective Value using Convexity

y-axis: Objective Value
x-axis: q
Optimal solutions for q=0 (red), q=0.1 (red), q=0.3 (green), q=0.9 (blue) and q=0.99 (blue).
The black lines are the bounds for the optimal objective value.

Figure 21 displays that even for a small number of grid points (G=5) narrow bounds can be computed. The performance of the bounds is best between grid points that are optimized by the same schedule. If additionally the objective value of this schedule is linear in q, then the lower and upper bound are equal in this interval. Should the spread between the bounds be too large at some points or intervals, it can easily be reduced by adding grid points in this specific area. In general, the optimal choice of grid points depends on the application. It works well to choose $g_1 = 0$ and $g_G = 1-\epsilon$ with some grid points in between with focus on the area of interest or equidistant.

6.3.2 Runtimes for more Time Steps

In the application example the number of time steps was set to $T = 5$ to represent the five days of a workweek. For this setting the computation time was not a big issue. But the number of possible schedules and therefore runtime increases very fast with the number of time steps. To estimate runtimes for a higher T, the emphasis was put on the time2 variable, which measures the time to enumerate and compare all schedules. time1 (the time to identify candidate tours) does not depend on T and can be seen as a constant in this regard. Section 5.3 already stated how the relation of number of candidate tours, number of time steps and the number of iterations works, so it is possible to extrapolate the computation times for other possible values of T. The predictions are summarized in Table 11.

Table 11: Runtime Predictions

	T	500km	600km	700km	800km	900km	1000km
time1	any	1s	6s	13s	14s	22s	29s
n	any	3	6	10	8	15	12
time2	5	<1s	2s	30s	7s	11m	2m
time2	6	<1s	25s	11m	3m	3h	37m
time2	7	1s	2m	2h	19m	35h	7h
time2	8	2s	11m	16h	2h	500h	75h

time code: seconds (s), minutes (m), hours (h)

If the number of time steps would increase from $T = 5$ to $T = 6$ this would still be possible to compute within reasonable time. But already for $T = 7$ one MPOP needs a runtime of more than one day. For a higher number of nodes (currently $N = 9$, excluding depot), which will typically result in a higher number of candidate tours, also $T = 6$ can take a long time. The predicted runtimes of Table 11 should not be seen as exact numbers but more as rough estimates. They still give a good intuition on which problem instances are manageable. Of course, runtime can be cut down by running the algorithm on a supercomputer.

7 Conclusions

The algorithm proposed in this thesis is designed to solve the MPOP exactly and in a time-efficient way. The MPOP is based on a set of nodes that are of different importance and have distances between each other. The aim is to find the optimal T-tuple of tours such that the total profit is maximized after T time steps. These tours must also satisfy a tour length limit constraint as well as start and end at the depot node. Furthermore, they have to fulfill the typical tour constraints known from the TSP. The path has to be connected, avoid subtours and each node must not be visited more than once per tour. Up to this point goes the general description of the MPOP.

In particular, the MPOP considered here has no vehicle capacity constraints, no time windows, a special growth/decay model and the full reward can be collected upon visit. The reward for visiting a node grows linear over time, while not visiting results in an exponential decay. The algorithm is very flexible with respect to this model. It can be exchanged easily in the code as long as the new model does not lead to negative node values. Theoretically, it is even possible to adjust the algorithm to allow different tour length limits for different time steps. This could represent shorter workdays that some companies have on Friday or Saturday. Of course, there would be an increase in runtime. On the other hand, it is not possible to implement the option to carry over unused resources to the next time step without a complete overhaul. The same holds true for time window constraints, which would require a new design of candidate tour identification with an exact TSP solver that can deal with time windows. Moreover, the algorithm cannot deal with objective functions with different objective variables such as minimizing the number of time steps ($\sim$ number of vehicles) such that a given profit is exceeded, which would be the case in some variations of the (Selective) Vehicle Routing Problem. In the classic setting, the depot has no reward attached to it. If there is the need to add an importance weight to the depot, it can be handled by adding a new node with the desired weight and distance 0 to the depot.

For the particular MPOP treated in this thesis, the runtime results were obtained and the runtimes for further problems were extrapolated. Since the time for the candidate tour identification only plays a minor part for the total runtime, is hard to predict and independent on T, only the runtime of the enumeration step is factored in. Table 12 outlines the relation of problem size and runtime.

Table 12: Problems to be solved in under 8 hours

n	3	4	5	6	7	8-9	10-12	13-18	19-31	32-72	73-297
T	17	13	11	10	9	8	7	6	5	4	3

n: number of candidate tours
T: number of time steps

For each number of candidate tours, the maximum number of time steps that can be covered within 8 hours of runtime is computed and stated in Table 12. Certainly, the runtime can be reduced by a decent factor by using modern hardware with a high-performance computing. The adduced problem sizes are approximately of same runtime complexity and point out the tradeoff between candidate tours and time steps. There are some limits to how fast the exact solution can be found because the problem is known to be NP-hard.

The division of the algorithm into two steps is a very important property. Fortunately, the problem design offers the opportunity to separate the input variables and solve one big problem with all variables by solving two smaller problems (identifying candidate tours and evaluating schedules) with less variables. Thereby, the enumeration complexity is drastically decreased. This is a benefit on the one hand, but restricts the implementation of new enhancements to the algorithm. The distances and length restrictions in the current algorithm have to be identical for every time step. This can be extended to have different conditions for different time steps as long as the conditions are independent of each other (e.g. no carry-over of unused resources to the next time step). Another restriction is the use of the symmetric TSP solver Concorde. It requires the distances to be symmetric, which could be a problem in cities with many oneway roads. Typically, TSP solvers that focus only on symmetric TSPs are faster and most real traffic networks can be approximated well enough by a symmetric distance matrix.

In some cases, it is necessary to visit each node at least once per period. If the optimal schedule does not satisfy this condition, then there are two possibilities to force its fulfillment. One approach is to increase the period length. For example, the tours of the garbage collection typically have greater period lengths in areas with a low population density to avoid ignoring some households. Another option is to add a penalty to the objective function for every unvisited node. Both approaches can also be combined if necessary.

The main use of the algorithm is the computation of the best schedule for given input parameters. Conversely, it can also be used the other way round. It can help planning the infrastructure by evaluating different variations of input values and find the optimal conditions for a profitable setting. In the example of harvest collection this can be used to find a good place to build the depot, determine the optimal number of truck drivers or buy/sell a field.

To start the algorithm, the input parameters have to be chosen. Finding the appropriate period length is usually not a problem. Different period lengths (weekly vs. biweekly) can also be compared if the setting is flexible in this regard. Choosing distances and maximum tour length is typically easy if it is a single unit such as time, length or money. It can be more difficult if the limit is expressed by an abstract score that combines multiple units. Assigning importance weights to the nodes will not be an issue normally. Tuning the decay factor can be a challenge. Two possible options, in case a suitable value is unknown, are to conduct a pilot study or to use multiple values for the decay factor as explained in section 6.3.1.

The topic in general and the algorithm in particular provide a lot of room for further research. Specifically, performance comparison with other algorithms are of interest such as comparing the solution quality with fast heuristics and the runtime with other exact algorithms or new variants such as an adaption of this algorithm which solves the enumeration part with (I)LP solver CPLEX. Conveniently, the problem formulation of chapter 4 already satisfies the conditions of integer linear programming, but starting with LP solving after candidate tour identification will reduce the amount of necessary variables drastically. Relaxing integer constraints and combine with branch and bound to ensure optimality should be used as well.

While there is a variety of possible enhancements to the algorithm that extend the range of problems it can be applied to, which were mentioned already, there are also adaptions that can potentially improve runtime. First there is the chance to make the number of LAP iterations flexible and dependent on input parameters. For example, a higher number of nodes leads to a higher number of LAP iterations. Second identifying potential patterns that cannot lead to optimal schedules could save a lot of enumeration steps and hence runtime. These patterns could include a criterion that penalizes a big overlap of tours in the same schedule or ranks schedules with the same tours based on their permutations.

One hypothesis for pattern utilization can be discarded because of a counter example. Like the results of section 6.3 suggested, scheduling the best tour on the last time step tends to produce very good solutions. While all optima in section 6.3 followed this pattern, it turned out to be non-binding. Even the weaker hypothesis, that the best tour is included in every optimal schedule (at an arbitrary time step), can be disproved. If these patterns were binding, omitting all schedules without best tours on the last/any time step would never result in a loss in solution quality while saving evaluation iterations. Albeit, this turned out to be a tendency and thus can be used for heuristics only. Figure 22, Table 13 and Table 14 depict the counter example that proves the hypotheses wrong.

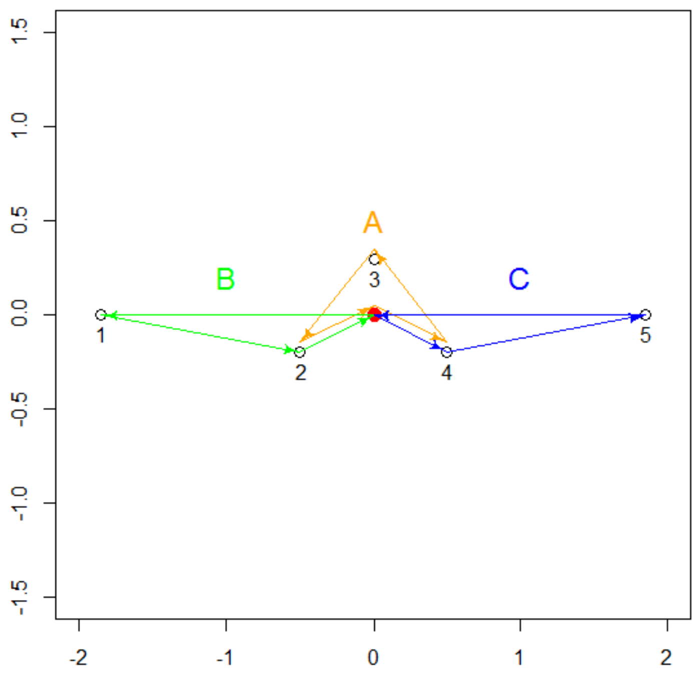

Figure 22: Counter Example - Best Tour

x-axis, y-axis: coordinates of datapoints
1-5: datapoints, red: depot
The candidate tours are mapped in orange, green and blue, respectively.

Table 13: Counter Example - Best Tour - Candidate Tours

Node	Weight	Tour A	Tour B	Tour C
0 (Depot)	NA	✓	✓	✓
1	1		✓	
2	1	✓	✓	
3	0.01	✓		
4	1	✓		✓
5	1			✓
Tour Length		2.49	3.75	3.75
Tour Profit		2.01	2.00	2.00
Best Tour		✓		

Tour Length Limit: C=4

Table 14: Counter Example - Best Tour - Optimal Schedules

Obj.Val	Day 1	Day 2	Day 3	Day 4	Day 5
17.20	C	B	C	B	C
17.20	B	C	B	C	B
17.12	A	B	C	B	C
17.12	A	C	B	C	B
17.05	B	C	A	B	C
17.05	C	B	A	C	B
17.04	B	A	C	B	C
17.04	C	A	B	C	B

Parameters used: T=5 time steps, decay factor q=0.9

Figure 22 shows the basic setting of the example. There are five nodes in addition to the depot, who are treated with the Euclidean distance measure. Furthermore, the candidate tours are already mapped because they play a major role in this counter example. As noted in Table 13, the maximum permissible tour length is C = 4. It summarizes which tours include which nodes and emphasizes the (intentional) strong imbalance of the weights. Tour A has a slightly higher score than tour B and tour C, so A is the best tour in this example. The tour value, which is equal to the profit of the tour in the setting of T = 1, does not factor in the overlap between the tours. In the light of overlap, tour B and tour C have a great synergy because they have disjoint sets of visited nodes (except for the depot), whereas tour A has an overlap of 1 node with both B and C. Due to the weight imbalance this accounts for roughly one half of its tour value and making it less efficient. For T = 5 time steps and n = 3 there are $3^5 = 243$ possible schedule and Table 14 shows the top eight ranked by their objective value. As can be seen there are pairs of schedules with same objective value and exchanged roles of tour B and tour C. The reason for that is that B and C have the same properties, especially same tour value and same overlap to other tours. The top pair schedule consists of alternating tour B and tour C, noticeably missing tour A. As has been noted, tour A is slightly ahead in terms of tour value but less efficient due to high overlap. Tour B and C complement each other in a convenient way. Given these points, it is not surprising that in this case tour A is not included in the optimal schedule. As a result, excluding schedules not involving the best tour cannot be used to accelerate an exact algorithm for the MPOP.

Appendix A

Table 15: Used Symbols and Variables A-S

#Assign	number of solved LAPs
#TSP	number of solved TSPs
ACO	Ant Colony Optimization
ACS	Ant Colony System
$B(k)$	boundary condition for k node visits
	lower bound for the tour length of shortest tour with k node visits
C	tour length limit, constraint level
D	matrix of distances
d_{ij}	distance of node i and node j
$d^0_{i,N}$	i-th order statistic of depot distances
G	number of grid points
g_i	grid points
LP, ILP	(Integer) Linear Programming
k	number of node visits
L	length of a tour
LAP	Linear Assignment Problem
m_i	distance from node i to nearest distinct node (excluding depot)
$m_{j,N}$	j-th order statistic of the m_i's
MPOP	Multi-Period Orienteering Problem
N	number of nodes
n	number of candidate tours
NV	number of node visits of a tour
NV_{max}	highest possible number of node visits
	for given distance matrix and tour length limit
$O_C(q)$	optimality function that for fixed C assigns optimal objective value to q
OP	Orienteering Problem
q	decay factor that controls the value process of unvisited nodes (see z_i^t)
S1, S2, S3	strategies to decide when to use LAP and TSP representation of the problem
STSP	Selective Traveling Salesman Problem
SVRP	Selective Vehicle Routing Problem
SVRPTW	Selective Vehicle Routing Problem with Time Windows

Table 16: Used Symbols and Variables T-Z

T	number of time steps
TOP	Team Orienteering Problem
TSP	Travelling Salesman Problem
TSSP+1	Traveling Salesman Subtour Problem with 1 additional constraint
u_i^t	auxiliary variable that is used for subtour elimination
	corresponds to position of node i in time step t
VRP	Vehicle Routing Problem
VRPTW	Vehicle Routing Problem with Time Windows
VNS	Variable Neighborhood Search
w_i	weight of node i, measures profitability of node i
X_{ij}^t	indicator that tells if node j follows immediately after node i in time step t
y_i^t	indicator that tells if node i is visited in time step t
z_i^t	value of node i in time step t

Appendix B

The following pages include the R code for the main functions regarding the proposed algorithm. The execution requires R and the installation of four extra packages (first four lines of code). The concorde software has to be separately downloaded from http://www.math.uwaterloo.ca/tsp/concorde/ . Additionally, non-Unix operating systems need a UNIX-emulating software such as Cygwin for Windows.

```r
require("adagio")        # LP solving
require("igraph")        # graph plotting
require("shape")         # graphics (including customizable arrows)
require("TSP")           # TSP solving

capture.output(concorde_path("your_concorde_installation_folder"), file='NUL')
            # file='NUL' only works on windows

####  Generate a set of points in a square by uniform distribution ####

get_datapoints <- function(n, lower=0, upper=1){
  datapoints <- round(matrix(runif(2*(n+1),min=lower,max=upper),n+1,2),2)
  datapoints[1,] <- rep((lower+upper)/2,2)
  rownames(datapoints) <- 0:n
  colnames(datapoints) <- c("x","y")
  return(datapoints)
}

#### Calculating optimal Tours (single assignment) ####

optimaltours <- function(distances, maxlength,prec=3){
  distances <- as.matrix(distances)
  n <- nrow(distances)-1
  rownames(distances) <- colnames(distances) <- 0:n
  nonzerodist <- distances[-1,-1]                          # distances between nodes, that are not the base
  zerodist <- distances[-1,1]                              # distances from zero to other nodes

  if(n>=2){
    mindist  <- sort(apply(nonzerodist,1,sort,partial=2)[2,],F)
                 # distance for every node to its nearest neighbour
    zeromins <- sort(zerodist,partial=2)[1:2]              # 2 smallest of these distances
    kmaximum <- function(maxlength,mindist,zeromins){      # computing maximum number of nodes
      if(sum(zeromins)>maxlength) return(0)
      k <- n
      cum.min <- c(0,cumsum(mindist[-1]))
      while(cum.min[k]+sum(zeromins)>maxlength){
        k <- k-1
      }
      return(k)
    }
  kmax <- max(kmaximum(maxlength,mindist,zeromins),2)
  } else {kmax <- 1}

  assign.costs <- floor(distances*10^prec)
  assign.costs <- assign.costs + diag(n+1)*4*max(assign.costs)

  opttours <- list()  # opttours includes the sets of vertices(unordered) (better for checking subroutes)
  opttours2 <- list() # opttours2 includes tours in right order (needed for execution of tours)
  k <- kmax
  count1<<-0
  count2<<-0
  use.assignment <- T

  opentours <- list()
  while(k>1&use.assignment==T){
    tours <- as.data.frame(combn(n,k))              # all possible routes with k vertices
    for(i in 1:ncol(tours)){
      ind <- 1+c(0,tours[,i])
      sol <- assignment(assign.costs[ind,ind])
      count1<<-count1+1
      if(sol$min*10^-prec<=maxlength) {use.assignment <- F; opentours[[length(opentours)+1]]<- tours[,i]}
                        # if assignment is possible, need to compute the TSP
    }
    k <- k-1
  }
  if(kmax>1&length(opentours)>0){
   tours <- as.data.frame(opentours)                      # check opentours
   for(i in 1:ncol(tours)){
      ind <- 1+c(0,tours[,i])
      suppressWarnings(sol <- solve_TSP(TSP(distances[ind,ind]),method="concorde",
                                                control=list(precision=prec,verbose=F)))
      count2<<-count2+1
      if(attr(sol,"tour_length")<=maxlength){
                  opttours[[length(opttours)+1]] <- tours[,i]
              opttours2[[length(opttours2)+1]] <- as.numeric(attr(sol,"names"))[-1]
      }
    }
  }

  while(k>1){
    tours <- as.data.frame(combn(n,k))             # all possible routes with k vertices
    if(length(opttours)>0){ partialtours <- NULL
      for(i in 1:length(opttours)){
        partialtours <- cbind(partialtours,combn(x=opttours[[i]],m=k))
      }
      partialtours <- as.data.frame(partialtours) # subroutes of known optimal routes
      tours <- tours[,is.na(match(tours,partialtours))]
                        # routes, that are not subroutes of known optimal routes
    }
    if(length(tours)==0) return(opttours2)        # stopping criterion
    if(is.vector(tours)) tours <- as.data.frame(tours)  # ncol does not work on vectors
    for(i in 1:ncol(tours)){
      ind <- 1+c(0,tours[,i])
      suppressWarnings(sol <- solve_TSP(TSP(distances[ind,ind]),method="concorde",
                                        control=list(precision=prec,verbose=F)))
      count2<<-count2+1
      if(attr(sol,"tour_length")<=maxlength){
                 opttours[[length(opttours)+1]] <- tours[,i]
             opttours2[[length(opttours2)+1]] <- as.numeric(attr(sol,"names"))[-1]
      }
    }
    k <- k-1
  }                                          # concorde accepts only TSPs with >=3 nodes

    tours <- as.data.frame(combn(n,1))
    if(length(opttours)>0){ partialtours <- NULL
      for(i in 1:length(opttours)){
        partialtours <- cbind(partialtours,combn(x=opttours[[i]],m=1))
      }
      partialtours <- as.data.frame(partialtours)  # subroutes of known optimal routes
      tours <- tours[,is.na(match(tours,partialtours))]
                                # routes, that are not subroutes of known optimal routes
    }
    if(length(tours)==0) return(opttours2)              # stopping criterion
    if(is.vector(tours)) tours <- as.data.frame(tours)
                        # tours will be a vector if only 1 column is selected
    for(i in 1:ncol(tours)){
      if(2*distances[1,1+tours[,i]]<=maxlength){ opttours[[length(opttours)+1]] <- tours[,i]
                                                opttours2[[length(opttours2)+1]] <- tours[,i]
      }
    }
  return(opttours2)
}

#### Calculating optimal Tours (double assignment) ####

optimaltours_2a <- function(distances, maxlength,prec=3){
  distances <- as.matrix(distances)
  n <- nrow(distances)-1
  rownames(distances) <- colnames(distances) <- 0:n
  nonzerodist <- distances[-1,-1]                      # distances between nodes, that are not the base
  zerodist <- distances[-1,1]                          # distances from zero to other nodes

  if(n>=2){
    mindist  <- sort(apply(nonzerodist,1,sort,partial=2)[2,],F)
                                # distance for every node to its nearest neighbour
    zeromins <- sort(zerodist,partial=2)[1:2]          # 2 smallest of these distances

    kmaximum <- function(maxlength,mindist,zeromins){  # computing maximum number of nodes
      if(sum(zeromins)>maxlength) return(0)
      k <- n
      cum.min <- c(0,cumsum(mindist[-1]))
      while(cum.min[k]+sum(zeromins)>maxlength){
        k <- k-1
      }
      return(k)
    }
kmax <- max(kmaximum(maxlength,mindist,zeromins),2)
} else {kmax <- 1}

assign.costs <- floor(distances*10^prec)
assign.costs <- assign.costs + diag(n+1)*4*max(assign.costs)

opttours <- list()       # opttours includes the stes of vertices (better for checking subroutes)
opttours2 <- list()      # opttours2 includes tours in right order (needed for execution)
k <- kmax
count1<<-0
count2<<-0
use.assignment <- T

opentours <- list()
while(k>1&use.assignment==T){
  tours <- as.data.frame(combn(n,k))                  # all possible routes with k vertices
  for(i in 1:ncol(tours)){
    ind <- 1+c(0,tours[,i])
    sol <- assignment(assign.costs[ind,ind])
    count1<<-count1+1
    if(sol$min*10^-prec<=maxlength) {
              use.assignment <- F; opentours[[length(opentours)+1]] <- tours[,i]
    }                                    # if assignment is possible, need to compute the TSP
  }
  k <- k-1
}

### second assignment start

opentours2 <- list()
if(k>1){
  tours <- as.data.frame(combn(n,k))       # all possible routes with k vertices
  for(i in 1:ncol(tours)){
    ind <- 1+c(0,tours[,i])
    sol <- assignment(assign.costs[ind,ind])
    count1<<-count1+1
    if(sol$min*10^-prec<=maxlength) {opentours2[[length(opentours2)+1]] <- tours[,i]}
              # if assignment is possible, need to compute the TSP
  }
  k <- k-1
}

### second assignment end

if(kmax>1&length(opentours)>0){
 tours <- as.data.frame(opentours)                                    # check opentours
 for(i in 1:ncol(tours)){
    ind <- 1+c(0,tours[,i])
    suppressWarnings(sol <- solve_TSP(TSP(distances[ind,ind]),method="concorde",
                                                  control=list(precision=prec,verbose=F)))
    count2<<-count2+1
    if(attr(sol,"tour_length")<=maxlength){
                         opttours[[length(opttours)+1]] <- tours[,i]
                         opttours2[[length(opttours2)+1]] <- as.numeric(attr(sol,"names"))[-1]
    }
  }
}

if(kmax>1&length(opentours2)>0){
 tours <- as.data.frame(opentours2)                         # check opentours2
 if(length(opttours)>0){ partialtours <- NULL
    for(i in 1:length(opttours)){
      partialtours <- cbind(partialtours,combn(x=opttours[[i]],m=k+1))
    }
    partialtours <- as.data.frame(partialtours)           # subroutes of known optimal routes
    tours <- tours[,is.na(match(tours,partialtours))]
              # routes, that are not subroutes of known optimal routes
 }
 if(is.vector(tours)) tours <- as.data.frame(tours)      # ncol doesn't work on vectors
 if(ncol(tours)>0){
  for(i in 1:ncol(tours)){
      ind <- 1+c(0,tours[,i])
      suppressWarnings(sol <- solve_TSP(TSP(distances[ind,ind]),method="concorde",
                         control=list(precision=prec,verbose=F)))
      count2<<-count2+1
      if(attr(sol,"tour_length")<=maxlength){
                         opttours[[length(opttours)+1]] <- tours[,i]
                         opttours2[[length(opttours2)+1]] <- as.numeric(attr(sol,"names"))[-1]
      }
    }
  }
}
```

```
  while(k>1){
    tours <- as.data.frame(combn(n,k))                  # all possible routes with k vertices
    if(length(opttours)>0){ partialtours <- NULL
      for(i in 1:length(opttours)){
        partialtours <- cbind(partialtours,combn(x=opttours[[i]],m=k))
      }
      partialtours <- as.data.frame(partialtours)    # subroutes of known optimal routes
      tours <- tours[,is.na(match(tours,partialtours))]
                # routes, that are not subroutes of known optimal routes
    }
    if(length(tours)==0) return(opttours2)            # stopping criterion
    if(is.vector(tours)) tours <- as.data.frame(tours)  # ncol does not work on vectors
    for(i in 1:ncol(tours)){
      ind <- 1+c(0,tours[,i])
      suppressWarnings(sol <- solve_TSP(TSP(distances[ind,ind]),method="concorde",
                                    control=list(precision=prec,verbose=F)))
      count2<<-count2+1
      if(attr(sol,"tour_length")<=maxlength){
                          opttours[[length(opttours)+1]] <- tours[,i]
                          opttours2[[length(opttours2)+1]] <- as.numeric(attr(sol,"names"))[-1]
      }
    }
    k <- k-1
  }                    # "concorde" accepts only TSPs with >=3 nodes

    tours <- as.data.frame(combn(n,1))
    if(length(opttours)>0){ partialtours <- NULL
      for(i in 1:length(opttours)){
        partialtours <- cbind(partialtours,combn(x=opttours[[i]],m=1))
      }
                partialtours <- as.data.frame(partialtours)      # subroutes of known optimal routes
                tours <- tours[,is.na(match(tours,partialtours))]
                # routes, that are not subroutes of known optimal routes
    }
    if(length(tours)==0) return(opttours2)                # stopping criterion
    if(is.vector(tours)) tours <- as.data.frame(tours)
                        # tours will be a vector if only 1 column is selected
    for(i in 1:ncol(tours)){
      if(2*distances[1,1+tours[,i]]<=maxlength){opttours[[length(opttours)+1]] <- tours[,i]
                                                opttours2[[length(opttours2)+1]] <- tours[,i]
      }
    }
  return(opttours2)
}

### Enumerate

enumerate <- function(poss.tours, n, rounds=5, weights=1, decay=0.9){
  nposs <- length(poss.tours)              # nposs number of possible tours
  l <- list()                              # list helps to create grid with all combinations
  for(i in 1:rounds){l[[i]] <- 1:nposs}
  combs <- expand.grid(l)                  # dataframe with all combinations
  colnames(combs)<- paste("Vx",1:ncol(combs),sep="")
  obj.val <- numeric(1)
  weights <- weights*rep(1,times=n)

  poss.mat <- matrix(NA,nrow=nposs,ncol=n)    # matrix for easier handling of possible tours
  for(i in 1:nposs){
    poss.mat[i,] <- table(factor(poss.tours[[i]],levels=1:n))
  }

  if(rounds==1){obj.val <- as.vector(poss.mat%*%weights)}

  if(rounds>1){
    if(length(poss.tours)>1){
      diffs <- t(apply(combs,1,diff))
      if(rounds==2) diffs <- t(diffs)
      best_ind <- which(poss.mat%*%weights==max(poss.mat%*%weights))
      if(length(best_ind)>1) { combs <- combs[apply(diffs^2,1,min)>0,]
       } else { combs <- combs[rowSums((diffs==0)*(combs[,-1]!=best_ind))==0,]
                        # =0, if there are no double tours or they are both the "best" tour
      }
    }

    for(j in 1:nrow(combs)){                 # evaluating each combination
      obj.val[j] <- evaluate(index=unlist(combs[j,]),poss.mat,weights,decay)
    }
  }

  return(cbind(obj.val,combs)[obj.val==max(obj.val),])
}
```

```
###  Evaluate

evaluate <- function(index, poss.mat, weights, decay=0.9){
  objperround <- numeric(length(index))
  program <- poss.mat[index,]
  status <- rep(1,times=length(weights))

  if(sd(weights)==0){
    for(i in 1:length(index)){
      objperround[i] <- program[i,]%*%status
      status[as.logical(program[i,])] <- 0
      status <- status*decay
      status <- status +1
  }}

  if(sd(weights)!=0){
    for(i in 1:length(index)){
      objperround[i] <- (program[i,]*weights)%*%status
      status[as.logical(program[i,])] <- 0
      status <- status*decay
      status <- status +1
  }}

  return(sum(objperround))
}

### Complete function solve_ori

solve_ori <- function(distances,maxlength,rounds=5,weights=1,decay=0.9,prec=6,input.tours=NULL){
  n <- nrow(as.matrix(distances))-1
  weights <- weights*rep(1,times=n)
  ifelse(is.list(input.tours),poss.tours <- input.tours,
                              poss.tours <- optimaltours_2a(distances,maxlength,prec)
  )
  if(length(poss.tours)==0) return(list("Objective Value"=0,"Schedule"=NULL,"Scheme"=NULL,
                                         "Maximum Length"=maxlength,"Weights"=weights,"Decay"=decay))

  sol <- enumerate(poss.tours, n, rounds, weights=weights, decay=decay)
  schedule <- as.matrix(sol[,-1])
  rownames(schedule) <- 1:nrow(schedule)
  colnames(schedule) <- paste("Day",1:rounds)
  scheme <- schedule

  for(i in 1:length(poss.tours)){
    schedule[schedule==i] <- paste(poss.tours[[i]],sep=" ",collapse=" ")
  }

  res <- list("Objective Value"=sol[1,1],"Schedule"=schedule,"Scheme"=scheme,
                                         "Maximum Length"=maxlength,"Weights"=weights,"Decay"=decay)

  return(res)
}


#### Plot the solution ####

plot_sol <- function(datapoints, tour=NA, xlim=c(-0.05,1.05), ylim=c(-0.05,1.05),main=""){
  par(mar=c(2,2,2,1))
  n <- nrow(datapoints)-1
  if(!is.null(xlim)) plot(matrix(datapoints[-1,],ncol=2), xlim=xlim, ylim=ylim,main=main,cex=1.3)
  if(is.null(xlim))  plot(matrix(datapoints[-1,],ncol=2),cex=1.3,main=main)
  points(t(datapoints[1,]),col="red",pch=16,cex=1.5)
  text(datapoints, labels=rownames(datapoints), adj=c(0.5, -0.5))

  if(!is.na(tour[1])){
    if(is.character(tour)) tour <- as.numeric(strsplit(tour,split=" ")[[1]])
    ind <- c(0,tour,0)+1
    ifelse(n==1,code<-3,code<-2)

    for(i in 1:(n+1)){
      Arrows(x0=datapoints[ind[i],1],y0=datapoints[ind[i],2],
              x1=datapoints[ind[i+1],1],y1=datapoints[ind[i+1],2],
              lwd=1,arr.length=0.25,arr.adj=1,code=code)
    }
  }
}

plot_sol2 <- function(datapoints, tour=NA, names=NULL, asp=1){
  n <- nrow(datapoints)-1
  mat <- matrix(0,nrow=n+1,ncol=n+1)
  colnames(mat) <- names
  if(!is.na(tour[1])){
    if(is.character(tour)) tour <- as.numeric(strsplit(tour,split=" ")[[1]])
    ind <- c(0,tour,0)+1

    for(i in 1:length(ind)){
      mat[ind[i],ind[i+1]] <- 1
    }
  }
  graph <- graph.adjacency(mat,mode="directed",weighted=NULL)
  plot(graph,layout=geocoords,asp=asp,vertex.label.color="black",vertex.color=rainbow(10,s=0.8)[10:1])
}


#### Calculate Tour length ####

tourlength <- function(tour, distances){
  if(is.character(tour)) tour <- as.numeric(strsplit(tour,split=" ")[[1]])
  distances <- as.matrix(distances)
  tourlength <- distances[1,tour[1]+1]+distances[1,rev(tour)[1]+1]  # first + last edge

  for(i in 1:(length(tour)-1) ){
    tourlength <- tourlength + distances[tour[i]+1,tour[i+1]+1]     # adding remaining edges
  }
  return(tourlength)
}

#### Renaming tour results with initial vertex names (city names) ####

vertexnames <- function(tour,vertexnames){
    if(!is.list(tour)&is.character(tour)) res <- strsplit(tour,split=" ")
    if(!is.list(tour)&!is.character(tour)) res <- tour <- list(tour)
    if(is.list(tour)) res <- tour
    for(i in 1:length(tour)){
      res[[i]] <- as.numeric(res[[i]])
      for(j in 1:length(vertexnames)){
        res[[i]][res[[i]]==j] <- vertexnames[j]
      }
    }
    return(res)
}
```

References

[1] Applegate, D., Cook, W., Dash, S., Rohe, A. (2002)
"Solution of a min-max vehicle routing problem"
INFORMS Journal on Computing, Vol. 14 Issue 2, 132-143

[2] Bellman, R. (1958)
"On A Routing Problem"
Quarterly of Applied Mathematics (Brown University)
Volume 16, No. 1, 87-90

[3] Boussier, S., Feillet, D. & Gendreau (2007)
"An exact algorithm for team orienteering problems"
4OR, Vol. 5, Issue 3, 211-230

[4] Chao, I., Golden, B.L., Wasil, E.A. (1996)
"A fast and effective heuristic for the orienteering problem"
European Journal of Operational Research, Vol. 88, 475-489

[5] Croes, G.A. (1958)
"A method for solving traveling salesman problems"
American Journal of Operations Research, Vol. 6 , 791-812

[6] Glover, F. (1986)
"Future paths for integer programming and links to artificial intelligence"
Computers & Operations Research, Vol. 13, Issue 5, 533-549

[7] Golden, B.L., Levy, L., Vohra, R. (1987)
"The orienteering problem"
Naval Research Logistics, Vol. 34, 307-318

[8] Golden, B.L., Wang, Q., Lyu, L. (1988)
"A multifaceted heuristic for the orienteering problem"
Naval Research Logistics, Vol. 35, 359-366

[9] Gueguen, C., (1999)
"Methodes de résolution exacte pour problémes de tournées de véhicules"
Doctoral Dissertation, Ecole Centrale Paris

[10] Gunawan, A., Lau, H. C.,Vansteenwegen, P. (2016)
"Orienteering Problem: A survey of recent variants,
solution approaches and applications"
European Journal of Operational Research
Vol. 255, Issue 2, 315-332

[11] Harvey, W., Ginsberg, M., (1995)
"Limited discrepancy search"
Proceedings of the 14th international joint conference on
artificial intelligence, Montreal, Vol. 1, 607-615

[12] Keller, C.P. (1989)
"Algorithms to solve the orienteering problem: A comparison"
European Journal of Operational Research, Vol. 41, 224-231

[13] Kruskal, J.B. (1964)
"Multidimensional scaling by optimizing goodness of fit to
a nonmetric hypothesis"
Psychometrika Vol. 29, 1-27

[14] Laporte, G., Martello, S. (1990)
"The Selective Traveling Salesman Problem"
Discrete Applied Mathematics, Vol. 26, 193-207

[15] Miller, C. E., Tucker, A. W., Zemlin, R. A. (1960)
"Integer programming formulations and travelling salesman problems"
Journal of the Association for Computing Machinery Vol. 7, 326-329

[16] Montemanni, R., Gambardella, L. M. (2009)
"Ant Colony System for Team Orienteering Problems with Time Windows"
Foundations of Computing and Decision Sciences, Vol. 34, 287-306

[17] Pillai, R.S. (1992)
"The traveling salesman subset-tour problem with one additional
constraint (TSSP+ 1)"
Doctoral Dissertation, The University of Tennessee (Knoxville)

[18] Ramesh, R., Brown, K.M. (1991)
"An efficient four-phase heuristic for the generalized
orienteering problem"
Computers & Operations Research, Vol. 18, Issue 2, 151-165

[19] Rosenkrantz, D. J., Stearns, R. E., Lewis, P. M., (1977)
"An analysis of several heuristics for the
traveling salesman problem"
SIAM Journal on Computing Vol. 6, Issue 3, 563-581

[20] Sokkappa, P.R. (1990)
"The cost-constrained traveling salesman problem"
Doctoral Dissertation, The University of California (Livermore)

[21] Tang, H., Miller-Hooks, E., (2005)
"A TABU search heuristic for the team orienteering problem"
Computers and Operations Research, Vol. 32, Issue 6, 1379-1407

[22] Tricoire, F., Romauch, M., Doerner, K. F., Hartl, R. F. (2010)
"Heuristics for the multi-period orienteering problem
with multiple time windows"
Computers & Operations Research, Vol. 37, 351-367

[23] Tsiligirides, T. (1984)
"Heuristic Methods Applied to Orienteering"
Journal of the Operational Research Society, Vol. 35, No. 9, 797-809

[24] Wren, A., Holliday, A. (1972)
"Computer scheduling of vehicles for one or more depots
to a number of delivery points"
Operational Research Quarterly, Vol. 23, 333-344

Online Sources

[25] City distances (Jan 21st 2017 12:00)
https://www.google.at/maps/dir///@47.7038659,11.0605785,7z

[26] Geographic coordinates for Austrian cities (Mar 1st 2017 13:00)
http://www.city-coordinates.net/home/1_oesterreich

[27] Geographic coordinates for Munich (Mar 1st 2017 13:00)
http://www.gpskoordinaten.de/

[28] Number of inhabitants (Mar 1st 2017 13:00)
https://www.citypopulation.de/Oesterreich-Cities_d.html
https://www.citypopulation.de/Deutschland-Cities_d.html