A Dynamic Programming Approach to the Orienteering Problem with Deterministic-Stochastic Weights

Marijn T. Waltman
Erasmus University Rotterdam

June 30, 2013

Abstract

In this paper a new variant of the Orienteering Problem with Stochastic Weights (OPSW) is introduced: the Orienteering Problem with Deterministic-Stochastic Weights (OPDSW), where the weights on the arcs are a mix of a deterministic weight plus a non-negative stochastic weight. The problem is solved exactly using a Dynamic Programming (DP) approach and multiple state pruning algorithms are proposed with the goal of making the DP algorithm more efficient. Specific combinations of these algorithms are tested in a case study with a dataset of 30 nodes. The pruning algorithms have a substantial positive effect on the runtime of the DP algorithm, while the optimal tour remains mostly unchanged. The combination of using the End-Of-Tour method plus the relaxed Completion Bound method (a pruning method that is unique for the OPDSW) results in the biggest reduction of the runtime. This reduction increases even further if an initial solution is used and if the relaxed Completion Bound algorithm is only applied on specific states, which are determined by some decision rule.

Keywords: Orienteering, Stochastic weights, Dynamic Programming, Pruning, Completion Bound

Contents

1 Introduction **2**

2 Dynamic Programming formulation **4**
2.1 OPDSW notation 4
2.2 Dynamic Programming notation 5
2.2.1 States, stages and decision sets 5
2.2.2 Pulling and reaching methods 6
2.2.3 Dominant states 7
2.3 Dynamic Programming algorithm 9

3 Pruning algorithms **12**
3.1 End-Of-Tour method 12
3.2 Completion Bound method 13
3.2.1 The exact Completion Bound algorithm 13
3.2.2 The relaxed Completion Bound algorithm 14

4 Case study **17**
4.1 Overview of algorithms 17
4.2 Implementation details 17
4.3 Results of the case study 18
4.3.1 Analysis of the results 19
4.3.2 Higher deadlines 21

5 Future research **24**

Chapter 1

Introduction

The Orienteering Problem (OP) is an interesting routing problem with many applications to real-world economic decision problems. It is similar to the well known Traveling Salesman Problem (TSP) and is also commonly referred to as the Selective Traveling Salesman Problem (STSP) or the Time-Constrained Traveling Salesman Problem (TCTSP). A complete directed graph is given along with a fixed start and end node. Each node has a certain reward for visiting that node and each arc has a certain weight. The goal of the problem is to find the path with the highest total profit that satisfies a certain capacity constraint on the total weights of the arcs in the path. In the OP we assume that each node can not be visited more than once, which means that the path must be elementary. This aspect of the OP makes it similar to the TSP, but instead of having to visit all nodes in the TSP we can choose which nodes we visit.

An application of the OP can be seen in multiple real world problems. Take for example a traveler who starts at home and has several destinations that he can choose from. Each destination (node) has a specific attractiveness (reward) and the roads between two destinations (arcs) have a certain travel time (weight), but the traveler only has a limited time T before he has to return back home.

In the original OP the weights on the arcs are considered to be deterministic. However this is not always the case in reality, where the weights are often uncertain; think for example of unforeseen events like traffic jams that could increase the travel time. A more accurate representation of real-world problems could be made by using stochastic weights. This variant of the OP is classified as the Orienteering Problem with Stochastic Weights (OPSW) [6]. In this paper I will explore a variant of the OPSW that uses a sum of a deterministic weight, which denotes the travel time in the best-case scenario, and a non-negative stochastic weight, which denotes the delay caused by random events. Because of the added uncertainty we can not simply state whether the sum of the weights of the arcs in a path is lower than the weight limit; instead we know the *probability* that this occurs. A path is therefore defined as feasible if this probability is greater than a certain required probability α. There is very little literature

on this specific variant of the OPSW. By lack of a previously defined name, the problem in this paper will be referred to as the Orienteering Problem with Deterministic-Stochastic Weights (OPDSW).

The focus in this paper is on finding the exact solution of the problem: the feasible path with the highest possible total profit. To this end I will apply a Dynamic Programming (DP) approach which is based on the DP approach for solving the TSP [11] and the OPSW [3, pp. 65-66]. The dynamic programming approach of the OPDSW will be formulated in detail in Chapter 2.

In dynamic programming, a pruning algorithm [1] is an algorithm that uses a certain pruning rule to determine at a certain decision node (called a state) whether another decision node should be visited or not. If the latter is the case, then that decision node and its subtree will be pruned. In Chapter 3 I will introduce two pruning rules that apply to the OP in general and three pruning algorithms that are based on these rules and are adapted to the OPDSW. In Chapter 4 I will test the efficiency of the dynamic programming approach by means of a case study and discuss how the aforementioned pruning algorithms affect the run time of the dynamic programming algorithm. Finally, Chapter 5 concludes the paper and suggests future research on this subject.

Chapter 2

Dynamic Programming formulation

In this chapter I will provide a formulation of the dynamic programming method. First I will discuss the notation of the OPDSW in detail. Then I will explain the concepts of states, stages and decision sets in dynamic programming terminology, I will explain what pulling and reaching methods are and how they relate to this dynamic program, and I will show that some states dominate other states. Finally I will present the DP algorithm and explain it in detail.

2.1 OPDSW notation

The notation of the OPDSW in this paper is for the most part consistent with the notation of the OPSW in Evers et al. [5, 6]. Let N be a set of n nodes. In this paper I will consider the case where the start and end node are the same node, which is called the depot and is denoted by node $0 \notin N$. In this case a path is formally called a tour. The case where the start and end nodes can be different will also be reviewed in Section 3.2.1. Let $G = (N^+, A)$ be a complete directed graph, where $N^+ = N \cup \{0\}$ is the set of all nodes including the depot and A is the set of all arcs. Each node $i \in N$ has an associated reward r_i. Each arc $(i, j) \in A$ has an associated weight f_{ij}:

$$f_{ij} = d_{ij} + \gamma_{ij}, \tag{2.1}$$

where d_{ij} is the deterministic part and $\gamma_{ij} \sim \Gamma(k_{ij}, \theta)$ is the stochastic part, which is a Gamma distributed random variable. I will assume that the scale parameter θ is the same for each arc and only focus on the shape parameter k_{ij}. Note that if we set $d_{ij} = 0$ the problem becomes the OPSW, and that if we remove γ_{ij} from (2.1) the problem becomes the deterministic OP. Lastly, assume that the matrices $\mathcal{D} = [d_{ij}]$ and $\mathcal{K} = [k_{ij}]$ and the vector $\mathcal{R} = [r_i]$ are all given a priori.

Next, consider a path $P = n_1 \rightarrow ... \rightarrow n_p$, which is a sequence of p distinct, consecutive nodes $n_1, ..., n_p$. Let the deterministic distance of a path P be

$$d(P) \equiv \sum_{i=1}^{p-1} d_{n_i n_{i+1}}, \tag{2.2}$$

or in other words: the sum of d_{ij} for all $p-1$ arcs (i,j) in P. Similarly let the stochastic distance of a path P be

$$k(P) \equiv \sum_{i=1}^{p-1} k_{n_i n_{i+1}}. \tag{2.3}$$

Note that the sum of the weights f_{ij} of the arcs in P is equal to $d(P)+\Gamma(k(P),\theta)$, since the sum of two Gamma distributed random variables $\gamma_1 \sim \Gamma(k_1,\theta)$ and $\gamma_2 \sim \Gamma(k_2,\theta)$ with the same scale parameter is $\Gamma(k_1+k_2,\theta)$ distributed.

2.2 Dynamic Programming notation

2.2.1 States, stages and decision sets

In this section I will introduce the dynamic programming approach and discuss its notation and terminology. The following notation is similar to the one used in Campbell et al. [3]. Let P_s be the path that has been traveled so far. Let i be the last node visited and let K be the lexicographically ordered set of all nodes in P_s (i.e. order is based on the index of each node). Also let $d \equiv d(P_s)$ and $k \equiv k(P_s)$ and lastly denote r as the sum of the rewards of all previously visited nodes. Then the state $s = (i, K, d, k, r)$ describes the current state of the dynamic program. These parameters will henceforth also be denoted as s_i, s_K, s_d, s_k, and s_r respectively.

Next I will describe the concepts of stages and decision sets in the dynamic program as is described in Bradley et al. [2, pp. 320-349]. For a given state $s = (i, K, d, k, r)$ the set of decisions D_s is the subset of all states that can be visited directly after this state. If the current node is i and the set of all previous nodes is K, then $N \backslash K$ is the set of all nodes that can still be visited. Thus all $|N \backslash K|$ decisions of s are of the form $(j, K \cup j, d + d_{ij}, k + k_{ij}, r + r_j)$, where $j \in N \backslash K$. Also, the set D'_s denotes the set of all predecessor states of s, or in other words: all states s^* where $s \in D_{s^*}$.

A stage is defined as follows: the dynamic program starts at an initial stage and each decision from a state in the current stage must put the dynamic program into the next stage, and so on. Therefore, let the stage be determined by the length of the path of the current state. This then satisfies the definition of a stage, since each decision increases the length of the path of the current state by 1. To initialize the dynamic program the initial stage is defined as 0 and has only one state: $s_0 = (0, \emptyset, 0, 0, 0)$. See Figure 2.1 for a graphical representation of states, stages and decision sets.

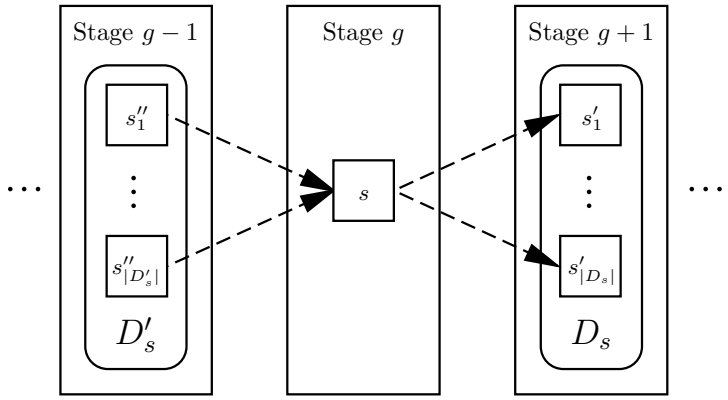

Figure 2.1: Graphical representation of states, stages and decision sets. A dashed arrow from one state to another indicates that the former is a predecessor state of the latter.

2.2.2 Pulling and reaching methods

Dynamic programming algorithms are often based on so called pulling methods [8] such as the following method, which uses forward recursion. Start at the initial stage and repeatedly determine for each state s in the current stage the optimal way of getting to it from a predecessor state $s'' \in D'_s$ from the previous stage, until the path can no longer be increased due to the time deadline. For every state s we memoize the functional value $f(s)$, which is defined by the recursive equation

$$f(s) = \max_{s'' \in D'_s} \{f(s'') + R(s'', s)\}, \tag{2.4}$$

where $R(s'', s)$ denotes the profit of going from state s'' to s. The optimal predecessor state $p(s)$ is then defined as

$$p(s) = \arg\max_{s'' \in D'_s} \{f(s'') + R(s'', s)\}. \tag{2.5}$$

Pulling methods like the one described above require that for a given state s we must know D'_s. However in our problem D_s is easily computed, but computing D'_s requires more work. For the OPDSW, however, we can make use of the so called *reaching* method [8]. This method is based on the concept of forward recursion and works the same way as the previously described method. However we can omit computing D'_s and simply only visit the states s' from D_s under the assumption that all predecessor states of s' have been considered in determining the functional value and that s is the optimal predecessor state. Because each state s has $|s_K| - 1$ predecessor states (since s_i can be visited from any node in $s_K \backslash s_i$) and because every one of those predecessors *will* be considered at some point in the algorithm, the previously mentioned assumption can safely be made.

Now this will be applied to the OPDSW. If we choose to visit j from (i, K, d, k, r), then this results in the expected profit

$$R(i, d, k, j) = r_j(F(T - d - d_{ij}; k + k_{ij}, \theta) - \alpha), \tag{2.6}$$

where $F(x; k, \theta)$ is the cdf of the Gamma distribution. Also, (2.4) translates to the following functional equation:

$$f(j, K', d', k', r') = \max_{(i,K,d,k,r):K=K'\setminus j, d'=d+d_{ij}, k'=k+k_{ij}, r'=r+r_j} \{f(i, K, d, k, r) + R(i, d, k, j)\}. \tag{2.7}$$

In (2.6), $F(T - d - d_{ij}; k + k_{ij}, \theta)$ denotes the probability that the path is feasible after adding the arc (i, j). We then subtract α, which means (2.6) is positive if and only if the feasibility probability is bigger than α. This is then multiplied by a factor r_j, which indicates that visiting nodes with higher profits implies a higher functional value than those with lower profits. Furthermore, if $R(i, d, k, j) \leq 0$, then node j can not be visited before the deadline, and we will therefore never travel to this node from the current state.

2.2.3 Dominant states

In this section I will explain the relationships between states and show that certain states dominate other states. This is an important result as it will substantially reduce the number of states that have to be visited. If there are two states s and s' such that $s_i = s'_i$ and $s_K = s'_K$, then the set of all paths[1] to complete the tour are the same. This is clear since the last nodes of the paths are the same and since both states have the same set of unvisited nodes. Now consider a path τ from this set. Then if

$$P(A_j \leq T) \geq P(A'_j \leq T) \tag{2.8}$$

for every $j \in \tau$, A_j and A'_j (where A_j and A'_j are random variables representing the arrival time to node j from s and s' respectively), it was proven in Campbell et al. [3, p. 66] that if $f(s) \geq f(s')$ then s dominates s'. This then implies that s' can be pruned.

In Campbell et al. [3, p. 66] it was shown that (2.8) is true for the OPSW if the stochastic distance parameter of s is smaller than or equal to that of s', so $s_k \leq s'_k$. This is true because

1. if $s_k \leq s'_k$, then $F(T; s_k, \theta) \geq F(T; s'_k, \theta)$, and
2. this remains true for every continuation of the path with cumulative stochastic distance $k' > 0$, since $F(T; s_k + k', \theta) \geq F(T; s'_k + k', \theta)$.

So in order to show that (2.8) is true for the OPDSW, the following two questions must be answered.

[1] both feasible and infeasible

<table>
<tr><td colspan="2"></td><td>$s_k < s'_k$</td><td>$s_k = s'_k$</td><td>$s_k > s'_k$</td></tr>
<tr><td rowspan="2">$s_d < s'_d$</td><td>$s_d < x \leq s'_d$</td><td rowspan="2">True</td><td rowspan="2">True</td><td>True</td></tr>
<tr><td>$s_d < s'_d < x$</td><td>False</td></tr>
<tr><td colspan="2">$s_d = s'_d$</td><td>True</td><td>True</td><td>False</td></tr>
<tr><td rowspan="2">$s_d > s'_d$</td><td>$s_d > s'_d \geq x$</td><td>True</td><td rowspan="2">False</td><td rowspan="2">False</td></tr>
<tr><td>$s_d > x > s'_d$</td><td>False</td></tr>
</table>

Figure 2.2: The outcomes of (2.9) for all combinations of s_d, s'_d, s_k and s'_k.

1. Which combinations of s_d, s'_d, s_k and s'_k satisfy

$$F(T - s_d; s_k, \theta) \geq F(T - s'_d; s'_k, \theta)? \tag{2.9}$$

2. Does (2.9) remain true after every continuation of the path with cumulative deterministic distance $d' > 0$ and cumulative stochastic distance $k' > 0$? In other words, is

$$F(T - s_d - d'; s_k + k', \theta) \geq F(T - s'_d - d'; s'_k + k', \theta) \tag{2.10}$$

   true for every combination of s_d, s'_d, s_k and s'_k that satisfies (2.9)?

To answer the first question, all possible combinations are enumerated in Figure 2.2, where *True* means that (2.9) is true for those values and obviously *False* means that this is false. Here x is the solution to

$$F(T - s_d; s_k, \theta) = F(T - x; s'_k, \theta), \tag{2.11}$$

where we note the following results:

$$\begin{aligned} s_k > s'_k &\iff s_d < x, \\ s_k = s'_k &\iff s_d = x, \\ s_k < s'_k &\iff s_d > x. \end{aligned} \tag{2.12}$$

Also, if we apply the identity in (2.11) to the left-hand side of (2.9), then we get

$$F(T - x; s'_k, \theta) \geq F(T - s'_d; s'_k, \theta), \tag{2.13}$$

which clearly shows that (2.9) is true if and only if $s'_d \geq x$ and false otherwise.

Lastly, the second question is shown to be true by the following proof.

Theorem 1. *Equation (2.10) is true for all combinations of s_d, s'_d, s_k and s'_k where (2.9) is true and false otherwise.*

Proof. We can easily deduce that $s_k \times s'_k \iff s_k + k' \times s'_k + k'$ (where $\times$ represents the operator $<$, $=$ or $>$), and similarly $s_d \times s'_d \iff s_d + d' \times s'_d + d'$. This covers almost all combinations; we only need to show that the outcomes of the two cases in the top-right corner and the two cases in the bottom-left corner of Figure 2.2 remain the same. Let x' be the solution to (2.11) with

$s_d \leftarrow s_d + d'$, $s_k \leftarrow s_k + k'$ and $s'_k \leftarrow s'_k + k'$. Then the results in (2.12) will still be true. Now we only need to verify that (2.10) is true if and only if $s'_d + d' \geq x'$ and false otherwise. Note that

$$F(T - (s_d + d'); s_k + k', \theta) \overset{(2.11)}{=} F(T - x'; s'_k + k', \theta), \tag{2.14}$$

so (2.10) is equivalent to $F(T - x'; s'_k + k', \theta) \geq F(T - (s'_d + d'); s'_k + k', \theta)$. So (2.10) is true if and only if (2.13) is true for $x \leftarrow x'$, $s'_d \leftarrow s'_d + d'$, $s_k \leftarrow s_k + k'$ and $s'_k \leftarrow s'_k + k'$. This implies that (2.10) is true if and only if $s'_d + d' \geq x'$ (and false otherwise), which is what needed to be shown. □

To summarize: if there are two states s and s' where $s_i = s'_i$ and $s_K = s'_K$, then the set of all paths to complete the tour are equal. Also, if the probability that s is feasible is greater than or equal to the probability that s' is feasible, then this remains true for every path to complete the tour. Therefore s' can never become more feasible than s and so if $f(s) \geq f(s')$, then this will remain true for all continuations of the tour. So s dominates s' and so s' can be pruned.

2.3 Dynamic Programming algorithm

Now that the notation has been discussed I will explain the basic dynamic programming algorithm. A general outline of this algorithm can be found in Algorithm 1. The algorithm itself is divided into 3 phases: the initialization phase (lines 1-3), the reaching phase (lines 4-17) and the tour retrieval phase (lines 18-22).

1. *The initialization phase:* The algorithm starts by defining the initial state s_0. We let s_{max} denote the state with the current highest functional value, but s_{max} is required to end at the depot. This implies that we are only interested in tours. Initially we set s_{max} to s_0. Lastly, let S be the set of states in the current stage (which is informally initialized at stage 0) and let S' be the set of feasible states in the *next* stage.

2. *The reaching phase:* The first while-loop signals that the reaching phase will stop if there are no more feasible states in the current stage. If this is false, then for all states s in the current stage we will visit all states s' from the decision set of s, but only if the expected profit of going to that state (denoted as ρ) is strictly positive. $s' = (j, s_K \cup j, *, *, *)$ is defined in line 8 and we let $f(s')$ and $p(s')$ be defined in terms of the current state s. This does not mean, however, that $f(s')$ and $p(s')$ are defined correctly, because we have not yet visited every single predecessor state of s' so we do not know if s is the optimal predecessor state or not. So before we decide to add s' to S', we note that if the state $(j, s_K \cup j, *, *, *)$ has already been visited, then there must now be another version of this state, say s^*, in S', but $f(s^*)$ and $f(s')$ may differ. Then by the result in Section 2.2.3 we know that if $f(s^*) \geq f(s')$, then s^* dominates s' and thus

s' can be pruned (i.e. it will not be added to S'). If instead $f(s') > f(s^*)$, then s' dominates s^* and now s^* can be pruned (i.e. it will be removed from S'). If however no state s^* exists in S', then $(j, s_K \cup j, *, *, *)$ has not yet been visited and we therefore add s' to S'. Further, in lines 15-16 we check if a return to the depot from this state is feasible and, if so, we check whether this would increase the current highest functional value $f(s_{max})$. Finally, in line 17 the stage of the dynamic program is increased by 1, by setting S to S'.

3. *The tour retrieval phase:* At this point we have visited all feasible states and we have successfully determined s_{max}. Then we can simply retrieve the optimal tour by starting at $s_{max,i}$, which is the last node of the tour before the return to the depot. Then we repeatedly insert the node of the optimal predecessor state of the current state before the first node in the tour, until we are back at s_0. That is, the optimal tour t is

$$t = 0 \rightarrow ... \rightarrow p(p(s_{max}))_i \rightarrow p(s_{max})_i \rightarrow s_{max,i} \rightarrow 0. \tag{2.15}$$

Further note that $s_{max,r}$ denotes the reward associated with this tour.

Algorithm 1: Dynamic Programming algorithm

Data: Time deadline T; required probability α; set of nodes N^+ and the associated data matrices $\mathcal{D}$, $\mathcal{K}$, and $\mathcal{R}$

Output: Optimal tour t

$s_0 \leftarrow (0, \emptyset, 0, 0, 0), f(s_0) \leftarrow 0, p(s_0) \leftarrow s_0$
$s_{max} \leftarrow s_0$
$S \leftarrow \{s_0\}$, $S' \leftarrow \emptyset$
while $S \neq \emptyset$ **do**
    for $s \in S$ **do** // $i \equiv s_i$
        for $j \in N \backslash s_K$ **do**
            $\rho \leftarrow R(i, s_d, s_k, j)$
            $\rho' \leftarrow R(j, s_d + d_{ij}, s_k + k_{ij}, 0)$
            $s' \leftarrow (j, s_K \cup j, s_d + d_{ij}, s_k + k_{ij}, s_r + r_j)$, $f(s') \leftarrow f(s) + \rho, p(s') \leftarrow s$
            if $\rho > 0$ **then**
                if $\exists s^* \in S' : s^*_i = s'_i \wedge s^*_K = s'_K$ **then**
                    if $f(s') > f(s^*)$ **then**
                        remove s^* and add s' to S'
                else add s' to S'
                if $\rho' > 0$ **and** $f(s') + \rho' > f(s_{max})$ **then**
                    $s_{max} \leftarrow s', f(s_{max}) \leftarrow f(s') + \rho'$
    $S \leftarrow S', S' \leftarrow \emptyset$
$s \leftarrow s_{max}, t \leftarrow$ empty tour
repeat
    insert s_i at beginning of t
    $s \leftarrow p(s)$
until $s = s_0$

Chapter 3

Pruning algorithms

In this chapter I will discuss several state pruning algorithms that apply to the OPDSW. In state pruning (see Bailey et al. [1]), states are adaptively removed (pruned) from the dynamic programming network when they are deemed not to lie on an optimal path by a certain pruning rule. The use of pruning methods for dynamic programs that make use of the reaching method has been shown to greatly reduce the number of states that are visited in Denardo and Fox [4], which in effect speeds up the DP algorithm. Since Algorithm 1 uses the reaching method, it is therefore interesting to see if using certain pruning algorithms might speed up the DP algorithm for the OPDSW. I will propose the following two pruning rules in which we can assume that the current state is s and that we determine whether or not to travel to a state s' with $s'_i = j \notin s_K$ and $s'_K = s_K \cup j$.

1. End-Of-Tour method: prunes s' if traveling from j to the depot is infeasible.

2. Completion Bound method: prunes s' if there exists no feasible path from j to the depot with a strictly higher profit than the current highest profit.

3.1 End-Of-Tour method

The End-Of-Tour (EOT) method is based on the assumption that if, after visiting node j, reaching the depot is infeasible, then there is no other feasible way to end the tour from s' and it should therefore be pruned. This assumption is based on the assumption that the weight f_{ij} of every arc (i, j) corresponds to the shortest path from i to j, thus satisfying the following triangle inequality:

$$E[f_{ij}] \leq E[f_{ih}] + E[f_{hj}]; \forall h \neq i, h \neq j, \tag{3.1}$$

which must in particular be true for $j = 0$. This is a commonly used assumption in many routing problems with practical applications (see for example Righini and Salani [10, p. 156], Laporte and Martello [9, p. 193]). If this inequality

wasn't true for $j = 0$, then $E[f_{i0}] > E[f_{ih}] + E[f_{h0}]$ and so there would be the possibility that $j \to 0$ is infeasible while an indirect path $j \to h \to 0$ is feasible (which clearly has a higher profit than $j \to 0$). For the purpose of demonstrating the End-Of-Tour method, I will thus assume that (3.1) with $j = 0$ is true.

Implementing this method in Algorithm 1 is pretty straightforward: instead of checking if $\rho' > 0$ in line 15 we add this as a condition to the if-statement in line 10, so a state will only be considered potentially feasible if the expected profit from j to the depot is positive.

3.2 Completion Bound method

The Completion Bound (CB) method is a pruning method that tries to find a feasible path to the depot, say τ, starting from s' such that the profit of s' including τ is strictly greater than the current maximum. If it fails to find such a path then s' will be pruned. Or simply put: the CB method prunes any state if no improvement can be made from visiting that state. This method is very good at pruning large subtrees of the dynamic programming network at a very early stage, but as a consequence the computation time is relatively high. In this section I will discuss two variations of the Completion Bound method: the exact CB algorithm and the relaxed CB algorithm.

3.2.1 The exact Completion Bound algorithm

In the exact CB algorithm the goal is to determine whether a feasible path from an initial state s' to the depot exists with a *functional value* that is strictly larger than the current highest functional value. This problem is exactly like the OPDSW, but in this case the start and end nodes can be different. It can thus be solved by simply reapplying an adjusted dynamic programming algorithm with initial state $s_0 = s'$. Note that for this algorithm to determine if a specific state can be pruned it must have visited all possible feasible paths from that state, and thus an exact solution method (such as the dynamic programming algorithm) is required; using an approximation method to save time is not possible. The adjusted DP algorithm is shown in Algorithm 2.

Let f_{min} denote the required functional value. In line 2 it is checked if the precondition, namely that the functional value of s'_0 must still be improved, is true. From then on the algorithm is roughly the same as the reaching phase in Algorithm 1. The only differences are that this algorithm stops and returns true in line 15 if an improvement has been made, and that it returns false if it has reached the end of the reaching phase (since it has therefore not found an improvement). The implementation of Algorithm 2 in Algorithm 1 is again straightforward. After line 6 but before line 10 we determine `ExactCB(`s'`,`$f(s_{max})$`)` (since $f(s_{max})$ denotes the current highest functional value) and we simply add the condition that this must be true to the if-statement in line 10.

Algorithm 2: Exact Completion Bound algorithm

Input: Initial state s'_0; minimum functional value f_{min}
Output: **true** if there exists a path starting at state s'_0 to the depot with functional value $> f_{min}$; **false** otherwise

```
1  Function ExactCB(s'_0, f_min)
2      if f(s'_0) > f_min then return true
3      S ← {s'_0}, S' ← ∅
4      while S ≠ ∅ do
5          for s ∈ S do                                   // i ≡ s_i
6              for j ∈ N\s_K do
7                  ρ ← R(i, s_d, s_k, j)
8                  ρ' ← R(j, s_d + d_ij, s_k + k_ij, 0)
9                  s' ← (j, s_K ∪ j, s_d + d_ij, s_k + k_ij, s_r + r_j), f(s') ← f(s) + ρ
10                 if ρ > 0 then
11                     if ∃s* ∈ S' : s*_i = s'_i ∧ s*_K = s'_K then
12                         if f(s') > f(s*) then
13                             remove s* and add s' to S'
14                     else add s' to S'
15                     if ρ' > 0 and f(s') + ρ' > f_min then return true
16         S ← S', S' ← ∅
17     return false
```

Note that, since this is also an instance of the OPDSW, we can implement all the pruning algorithms from this chapter in this algorithm as well. So every combination of the exact CB algorithm plus other pruning algorithms, such as the End-Of-Tour method, the relaxed CB algorithm or a recursive implementation of the exact CB algorithm, are all possible and will be explored in Chapter 4.

3.2.2 The relaxed Completion Bound algorithm

Next I propose an alternative to the exact Completion Bound method. This method is based on the fact that with each arc (i, j) that we travel, we add a value $d_{ij} > 0$ to d and $k_{ij} > 0$ to k, both of which decrease the probability that the tour is feasible. So if we disregard the stochastic distance per arc (e.g. we set $k_{ij} = 0$), then there is a theoretical upper bound $\overline{d}$ on the deterministic distance that we have left to travel before the path becomes infeasible. For every actual path the deterministic distance will always be smaller than $\overline{d}$ since $k_{ij} > 0$. In the case of the OPDSW this upper bound can be defined as the x where $F(x; s'_k, \theta) = \alpha$, so

$$\overline{d} = T - F^{-1}(\alpha; s'_k, \theta), \tag{3.2}$$

Algorithm 3: Relaxed Completion Bound algorithm

Input: Initial state s'_0; maximum deterministic travel time d_{max}; minimum total reward r_{min}

Output: **true** if there exists a path starting at state s'_0 to the depot with total reward $> r_{min}$ and total deterministic travel time $\leq d_{max}$; **false** otherwise

Function `RelaxedCB`(s'_0,d_{max},r_{min})
- **if** $s'_{0,r} > r_{min}$ **then return true**
- **if** $s'_{0,d} + \min_{m \neq i}\{d_{im}\} + \min_{m \neq 0}\{d_{m0}\} > d_{max}$ **then return false**
- $S \leftarrow \{s'_0\}, S' \leftarrow \emptyset$
- **while** $S \neq \emptyset$ **do**
  - **for** $s \in S$ **do** // $i \equiv s_i$
    - **for** $j \in N \backslash s_K$ **do**
      - $\rho \leftarrow R'(i, s_d, j)$
      - $\rho' \leftarrow R'(j, s_d + d_{ij}, 0)$
      - $s' \leftarrow (j, s_K \cup j, s_d + d_{ij}, s_k + k_{ij}, s_r + r_j), f(s') \leftarrow f(s) + \rho$
      - **if** $\rho > 0$ **then**
        - **if** $\exists s^* \in S' : s^*_i = s'_i \wedge s^*_K = s'_K$ **then**
          - **if** $f(s') > f(s^*)$ **then**
            - remove s^* and add s' to S'
        - **else** add s' to S'
        - **if** $\rho' > 0$ **and** $s'_r > r_{min}$ **then return true**
  - $S \leftarrow S', S' \leftarrow \emptyset$
- **return false**

where $F^{-1}(p; k, \theta)$ is the inverse cdf of the Gamma distribution with $0 \leq p \leq 1$. We can further decrease this upper bound by noting that if an improvement must still be made then the path must visit at least one node, so we know that we can at least add the minimum stochastic distance from node i to another node and from any node to the depot. That is, we can add

$$\min_{m \in N \backslash s_K, m \neq i}\{k_{im}\} + \min_{m \in N \backslash s_K, m \neq 0}\{k_{m0}\}, \tag{3.3}$$

where $i \equiv s'_i$, to the shape parameter of F^{-1} in (3.2).

The disregard of the stochastic distance has effectively turned the OPDSW into the deterministic OP, so we can make use of the many fast known algorithms to solve the deterministic OP for the relaxed CB method. For the computational experiments in this paper I will apply a dynamic programming based algorithm, which is shown in Algorithm 3. Since the functional value depends on the expected profit, which requires the addition of a stochastic distance (which we have disregarded) we can not make use of the functional value for comparing the current solution with the optimal solution. Instead we must use the sum of

the profits of the previously visited nodes, s'_r, to do this. Furthermore, to suit the deterministic OP, the expected profit function is changed to

$$R'(i, d, j) = r_j(T - d - d_{ij}). \tag{3.4}$$

Here $T - d - d_{ij}$ denotes the time that is left to finish the tour after the arc (i, j) has been traveled, and it is negative if and only if the path is infeasible.

I will now explain Algorithm 3. Let r_{min} denote the required total profit and let d_{max} denote the limit on the total deterministic distance. The algorithm first starts by determining if an improvement must still be made by only continuing if $s'_{0,r} \leq r_{min}$. If this is true, then we check whether it is possible to at least travel the minimum deterministic distance from node i to another node and from any node to the depot (which is similar to Equation (3.3)). If this is false then obviously the path can not be improved; if this is true, then the algorithm continues in a similar way as Algorithm 2. The only differences are the replacement of the expected profit function from (2.6) to (3.4), and in line 16 the profit s_r is compared to r_{min} to determine if the algorithm can return true. Lastly, the implementation of this algorithm in Algorithm 1 is similar to the implementation of Algorithm 2 in Algorithm 1, but this algorithm is initialized by `RelaxedCB(`s',$\overline{d}$,$s_{max,r}$`)` (since $s_{max,r}$ denotes the current highest profit).

Chapter 4

Case study

In this chapter I will test the Dynamic Programming algorithm from Chapter 2 including multiple combinations of the pruning algorithms in Chapter 3, by solving the OPDSW for a dataset consisting of 30 nodes (excl. the depot). All algorithms are then compared based on computation time and the number of states that are visited.

4.1 Overview of algorithms

Table 4.1 provides an overview of different versions of the dynamic programming algorithm, all of which do not use the End-Of-Tour method. To avoid overly long labels, if the algorithm does include the EOT method then its label will be suffixed with a '+', so DP with EOT is DP+ and ECB with EOT is ECB+. Thus the following are all the algorithms that will be tested:

1. DP
2. DP+
3. DP/ECB
4. DP+/ECB
5. DP/ECB+
6. DP+/ECB+
7. DP/ECB/ECB
8. DP+/ECB/ECB
9. DP/ECB+/ECB+
10. DP+/ECB+/ECB+
11. DP/ECB/RCB
12. DP+/ECB/RCB
13. DP/ECB+/RCB
14. DP+/ECB+/RCB
15. DP/RCB
16. DP+/RCB

4.2 Implementation details

The dataset consists of 30 nodes plus a depot in Euclidean space. The deterministic distance is set to the Euclidean distance between two nodes. The scale

Label	Description
DP	Basic dynamic programming algorithm without the addition of pruning algorithms.
DP/ECB	DP with the exact Completion Bound method without pruning algorithms.
DP/ECB/ECB	DP with the exact CB method, which itself also uses the exact CB method.
DP/ECB/RCB	DP with the exact CB method, which itself uses the relaxed CB method.
DP/RCB	DP with the relaxed Completion Bound method.

Table 4.1: An overview of the different versions of the DP algorithm with their respective labels.

parameters of the stochastic distance of all arcs are independent random variables with a mean of approx. 3; the shape parameter θ is fixed at 0.5. For all algorithms the time deadline T varies between 20 and 40 with increments of 5. For all instances the required probability α is set to 0.95, meaning that at most 5% of the tours is allowed to be infeasible.

If an algorithm implements a Completion Bound method (exact or relaxed), then the algorithm is run twice: in the first run s_{max} is set to s_0 (so $f(s_{max}) = 0$ and $s_{max,r} = 0$), and in the second run s_{max} is set to the s_{max} of the first run (so s_{max} is the optimal solution). This is done to investigate the use of any initial solution (with $f(s_{max}) > 0$ and $s_{max,r} > 0$) to set a high initial benchmark in order to prune more states early on in the DP algorithm. Thus using the actual solution as the initial solution will show the results in the theoretical (yet plausible) case that the initial solution already is the optimal solution.

The implementation of all algorithms was done in Java and were run on a Windows PC with 3.10 GHz Intel Core i5-2400 64-bit quad core processor with 4.00 GB of RAM.

4.3 Results of the case study

The solutions of the DP algorithm are shown in Table 4.2. Every algorithm returns a tour with a profit that is equal to the profit of the solution; the tour itself must of course be feasible but it might differ from the tour in Table 4.2. The DP algorithm does however guarantee that its solution tour has the highest probability of being feasible. The solution tours from all algorithms in this case study are exactly the same as the tours from the DP algorithm, except for the DP/RCB and DP+/RCB algorithms, where the solution tour for $T = 35$ was different (it was reversed) yet still feasible.

Table 4.4 shows the runtimes (in seconds) of every algorithm over all time deadlines and Table 4.5 shows the number of visited states, where a 'visited' state is defined as a feasible state that passes all pruning algorithms that the

T	Profit	Probability	Tour
20	25	0.9852	$0 \to 25 \to 26 \to 0$
25	45	0.9633	$0 \to 27 \to 30 \to 25 \to 26 \to 0$
30	55	0.9593	$0 \to 27 \to 29 \to 30 \to 25 \to 26 \to 0$
35	65	0.9723	$0 \to 26 \to 30 \to 25 \to 21 \to 20 \to 11 \to 0$
40	80	0.9594	$0 \to 20 \to 21 \to 22 \to 24 \to 25 \to 30 \to 26 \to 0$

Table 4.2: Solutions to the problem of this case study for multiple time deadlines.

specific algorithm implements. When the DP algorithm implements the ECB or RCB methods, then the tables show two numbers: the number before the brackets shows the results of the first run (where s_{max} is set to s_0) and the number between brackets shows the results of the second run (s_{max} is set to the optimal solution).

4.3.1 Analysis of the results

The results show that the use of one or more pruning algorithms can lead to an improvement of the runtime when compared to using no pruning algorithms at all. These effects on the runtime also increase as T gets larger and thus as the total number of feasible tours increases. The following are the effects of the individual pruning algorithms on the DP algorithm.

1. *The End-Of-Tour method:* The EOT method has a substantial positive effect on the runtime for both the DP and the ECB algorithms, regardless of the other pruning algorithms that were used. Therefore I advise that the EOT method should be used in the DP algorithm, given that the data allows for it.

2. *The exact Completion Bound method:* The ECB method on its own (without implementing any pruning algorithms itself) only decreases the runtime when T gets large. This means that the effect of the ECB method will only be noticeable if there are many feasible tours, which is obvious since the number of states that are pruned as a result of implementing the algorithm will increase as the total number of states increases. If the ECB algorithm itself implements pruning algorithms, then the runtime will decrease; though this is not true for the DP/ECB/ECB and DP+/ECB/ECB algorithms. The biggest decrease in runtime can be made by letting the ECB method apply both the EOT and the RCB methods.

   If we compare the results of using no initial solution versus using an initial solution, then we see a general reduction of the runtime in the latter case. The effect of the reduction varies depending on the pruning algorithms that the ECB method implements. In the DP+/ECB and

T	DP/RCB	DP+/RCB
20	0.063	0.024
25	0.078	0.065
30	1.009	0.589
35	2.066	1.592
40	17.78	11.69

(a) The runtimes (in seconds)

T	DP/RCB	DP+/RCB
20	70	20
25	89	88
30	1509	1204
35	2883	2718
40	39910	31808

(b) The number of visited states

Table 4.3: Results of the RCB algorithms, if an initial solution is used, after applying the decision rule.

DP+/ECB/ECB algorithms the runtime surprisingly increases when an initial solution is used, and only for $T = 25$ and $T = 30$. Although I am unsure about the exact cause, it might be the result of the exponential increase of the runtime when the number of nodes n gets larger. That is, the algorithm might take more time to prune states at an early stage (where it must solve the OPDSW for lots of nodes) than it takes to visit other states first and to prune states at a later stage (when the OPDSW contains fewer nodes). As we can see in Table 4.5 that the number of visited states is particularly low for those problem instances, this further supports the previous statement.

3. *The relaxed Completion Bound method:* The RCB method on its own has a substantial positive effect on the runtime of the DP algorithm: it reduces the runtime of the DP algorithm approx. 513 times at $T = 40$ and approx. 677 times when the EOT method is also implemented. It also outperforms the other pruning algorithms by a wide margin.

   Using an initial solution with the RCB method seems to decrease the runtime for smaller T but the reduction gradually gets smaller as T gets larger and eventually causes a (quite substantial) increase in the runtime at $T = 40$. This is caused by the fact that the RCB method is bad at pruning states at the first few stages of the DP algorithm due to the imposed relaxation. It only actually prunes states at later stages of the algorithm, as opposed to the exact CB algorithm which is able to prune states from a very early stage. As a result, the RCB method causes unnecessary runtime (as it fails to prune states) relatively often if the algorithm has a lot of stages yet to visit. However, this problem can be omitted if a certain decision rule would be used to determine if the RCB algorithm should be applied at a specific state or not. A simple arbitrarily chosen rule such as "only apply the RCB method if the length of the current path is bigger than or equal to the length of the path of the initial solution divided by 2" has been applied, and the results are shown in Table 4.3. It shows a dramatic decrease of the runtimes for higher values of T and for all T a decrease when compared to using no initial solution. The DP+/RGB algorithm including this decision rule is also the fastest algorithm that has

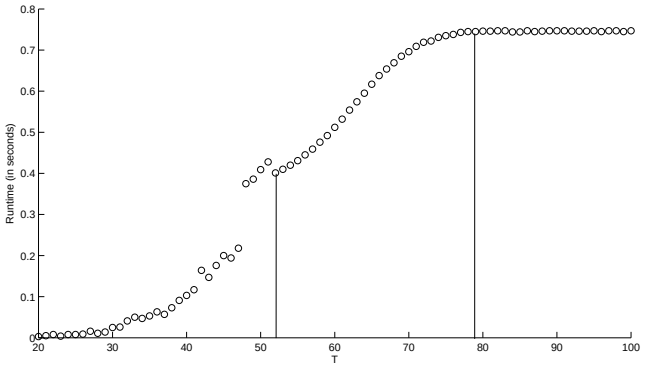

Figure 4.1: Runtimes of the fastest algorithm on a smaller dataset.

been tested.

4.3.2 Higher deadlines

To examine the effects of the dynamic programming algorithm on higher deadlines, I will apply the DP+/RGB algorithm (including the decision rule) to a smaller dataset, which uses the first 10 nodes of the previous dataset. The resulting runtimes can be found in Figure 4.1. The leftmost vertical line at $T = 52$ denotes the point where the optimal tour first contains all 10 nodes. $T = 48$ denotes the first point where the optimal tour contains 9 nodes. The big increase in runtimes shows that the algorithm performs poorly when the optimal tour contains 9 nodes[1]. After $T = 52$ the length of the tour stays equal to 10, but the composition of the tour still changes. This stops at $T = 79$, after which the tour remains the same. The figure shows that the runtime continues to increase exponentially until the tour contains all nodes. Then the runtime will still increase, but it will slowly converge at the point where the optimal tour remains the same.

Up to this point the deadlines for the experiments with the larger dataset have been relatively low and the optimal tours have been relatively short; the route at $T = 40$ consists of only 7 out of 30 nodes. Running the algorithm at $T = 60$ takes over 24 hours and contains merely 11 out of 30 nodes. Since the runtime is expected to increase exponentially even further, it is clear that this algorithm is not practical for large problem instances (i.e. high deadlines and/or a lot of nodes).

[1]This is a result of the fact that the RCB algorithm is bad at pruning states when the optimal tour is long. The current decision rule postpones the RCB algorithm to later stages in the DP algorithm, but the effects of it are still visible when the optimal tour contains a lot of nodes. A better decision rule could potentially reduce the effects of this problem completely.

T	DP	DP+	DP/ECB	DP+/ECB	DP/ECB+	DP+/ECB+	DP/ECB/ECB	DP+/ECB/ECB	DP/ECB+/ECB+	DP+/ECB+/ECB+	DP/ECB/RCB	DP+/ECB/RCB	DP/ECB+/RCB	DP+/ECB+/RCB	DP/RCB	DP+/RCB
20	1.351	0.136	1.527	0.624	0.576	0.140	1.542	0.653	0.578	0.138	1.580	0.674	0.591	0.142	0.344	0.097
			(1.403)	(0.518)	(0.399)	(0.077)	(1.403)	(0.531)	(0.399)	(0.078)	(1.433)	(0.549)	(0.414)	(0.076)	(0.100)	(0.034)
25	7.776	0.521	11.84	7.693	1.980	0.630	12.03	7.860	1.981	0.604	11.82	7.669	2.028	0.642	0.677	0.215
			(11.65)	(9.954)	(0.935)	(0.521)	(11.76)	(10.07)	(0.902)	(0.519)	(7.820)	(6.178)	(0.906)	(0.519)	(0.078)	(0.069)
30	51.50	3.034	96.73	77.21	10.05	4.845	106.1	86.76	10.18	5.059	87.61	68.00	9.933	4.733	1.667	0.893
			(89.08)	(87.39)	(4.672)	(4.408)	(99.13)	(97.39)	(4.606)	(4.384)	(20.33)	(18.71)	(3.119)	(2.854)	(0.257)	(0.222)
35	702.4	19.15	728.4	624.1	61.77	39.07	909.9	813.8	65.30	43.41	600.2	507.5	59.29	36.38	5.822	4.154
			(594.4)	(591.6)	(35.35)	(35.26)	(807.8)	(805.6)	(37.57)	(37.58)	(35.09)	(34.59)	(9.982)	(9.842)	(5.131)	(4.218)
40	14613	166.4	4761	4379	367.90	286.7	7211	6862	440.6	362.0	3457	3086	329.9	248.1	28.54	21.64
			(3653)	(3636)	(248.3)	(248.2)	(6390)	(6429)	(312.8)	(313.4)	(55.93)	(55.86)	(31.20)	(31.01)	(477.4)	(405.2)

Table 4.4: Comparison of runtimes (in seconds) of every algorithm over multiple time deadlines T.

T	DP	DP+	DP/ECB	DP+/ECB	DP/ECB+	DP+/ECB+	DP/ECB/ECB	DP+/ECB/ECB	DP/ECB+/ECB+	DP+/ECB+/ECB+	DP/ECB/RCB	DP+/ECB/RCB	DP/ECB+/RCB	DP+/ECB+/RCB	DP/RCB	DP+/RCB
20	1840	134	160	40	160	40	160	40	160	40	160	40	160	40	254	50
			(54)	(36)	(54)	(36)	(54)	(36)	(54)	(36)	(54)	(36)	(54)	(36)	(108)	(42)
25	14286	1274	435	200	435	200	435	200	435	200	435	200	435	200	891	316
			(59)	(58)	(59)	(58)	(59)	(58)	(59)	(58)	(59)	(58)	(59)	(58)	(89)	(88)
30	95620	10382	1518	932	1518	932	1514	928	1514	928	1518	932	1518	932	2854	1900
			(68)	(66)	(68)	(66)	(60)	(60)	(60)	(60)	(68)	(66)	(68)	(66)	(304)	(294)
35	560204	73726	5805	5376	5805	5376	5569	5140	5569	5140	5767	5338	5767	5338	12023	11426
			(60)	(60)	(60)	(60)	(60)	(60)	(60)	(60)	(60)	(60)	(60)	(60)	(1769)	(1744)
40	2921821	452978	24431	24150	24431	24150	23785	23504	23785	23504	24335	24054	24335	24054	64141	63598
			(60)	(60)	(60)	(60)	(60)	(60)	(60)	(60)	(60)	(60)	(60)	(60)	(5784)	(5782)

Table 4.5: Comparison of the number of visited states of every algorithm over multiple time deadlines T.

Chapter 5

Future research

Since the OPDSW and OPSW are largely unexplored, many directions are open for future research. The optimal solution could be improved even further by using a faster algorithm than the DP algorithm that was used in this paper to solve the deterministic OP in the relaxed CB method. This could for example be done by using an Integer Programming formulation (see Vansteenwegen et al. [12, p. 3]) or any of the other exact solution methods that are given in Feillet et al. [7]. For finding an initial solution to use in the Completion Bound methods, a heuristic can be applied to find an approximate solution of the OPDSW, where the emphasis lies on finding a solution fast rather than finding an accurate solution. Many heuristics that work for the OPSW (see Campbell et al. [3]) can be applied to the OPDSW to get this result, but fast heuristics that work specifically for the OPDSW could be developed. Also the results from Table 4.3 show that even a simple decision rule to determine when to apply the CB method has a big impact on the runtime. I am therefore curious in the effectiveness of other decision rules and if an 'optimal' decision rule would exist.

Besides improving the optimal solution, other Completion Bound methods could be explored. An example would be a method that determines, based on an initial solution, whether there exists a better solution that must include a certain node. If this is not the case then the node would never lie on the optimal path and so it can be removed from the graph. This method could then be applied a priori, which would result in a smaller graph and thus a reduction in the runtime for the remainder of the algorithm.

Bibliography

[1] M. D. Bailey, R. L. Smith, and J. M. Aiden. A reach and bound algorithm for acyclic dynamic-programming networks. *Wiley Periodicals*, 2007.

[2] S. P. Bradley, A. C. Hax, and T. L. Magnanti. *Applied Mathematical Programming*. Addison-Wesley Pub. Co., 1977.

[3] A. M. Campbell, M. Gendreau, and B. W. Thomas. The orienteering problem with stochastic travel and service times. *Annals of Operations Research*, 2011.

[4] E. V. Denardo and B. L. Fox. Shortest-route methods: 1. reaching, pruning, and buckets. *Operations Research*, 1979.

[5] L. Evers, T. Dollevoet, A. I. Barros, and H. Monsuur. Robust uav mission planning. *Annals of Operations Research*, 2011.

[6] L. Evers, K. Glorie, S. van der Ster, A. I. Barros, and H. Monsuur. A two-stage approach to the orienteering problem with stochastic weights. *Computers and Operations Research*, 2013.

[7] D. Feillet, P. Dejax, and M. Gendreau. Traveling salesman problems with profits. *Transportation Science*, 2005.

[8] P. A. Jensen and J. F. Bard. *Operations Research Models and Methods*. John Wiley and Sons, 2003.

[9] G. Laporte and S. Martello. The selective traveling salesman problem. *Discrete Applied Mathematics*, 1990.

[10] G. Righini and M. Salani. New dynamic programming algorithms for the resource constrained elementary shortest path problem. *Networks*, 2008.

[11] M. Sniedovich. A dynamic programming algorithm for the traveling salesman problem. *ACM SIGAPL APL Quote Quad*, 1993.

[12] P. Vansteenwegen, W. Souffriau, and D. V. Oudheusden. The orienteering problem: A survey. *European Journal of Operational Research*, 2011.