Free University of Bolzano
Faculty of Computer Science

An Efficient Heuristic for Orienteering with Categories

Submitted for the degree of the Bachelor of Computer Science and Engineering

Author
Kevin Wellenzohn

Supervisor
Prof. Dr. Sven Helmer

Co-Supervisor
Prof. Dr. Johann Gamper

July, 2013

Acknowledgments

At this point I would like to thank a couple of people who helped me tremendously during the last months and without whom this thesis would not exist in the present form. First, I would like to thank my supervisor, Prof. Sven Helmer, who was always willing to give me helpful advice.

Part of this group effort were also my co-supervisor Prof. Johann Gamper and my friends and fellow students Hannes Mitterer, Manuel Stuefer and Paolo Bolzoni, who patiently listened to my proposals and gave my sincere feedback.

On the other side I would like to thank my girlfriend, Katrin Kaserer, and my family, who were a great source of motivation and inspiration for me.

Abstract

We present an efficient heuristic for the *Orienteering Problem (OP)* using the example of a tourism application. Our system recommends custom-tailored itineraries based on the tourists' preferences and time constraints, guiding them from a starting point to a destination point and visiting a series of Points of Interest (POIs) in between. This problem is generally known to be NP-hard and therefore no algorithm to compute an optimal solution efficiently is likely to be found.

In order to recommend sufficiently interesting itineraries we generalize the *Orienteering Problem* by assigning each POI to a category, thus obtaining the *Orienteering Problem with Maximum Point Categories (OPMPC)*. Travelers define which categories they like and how many POIs they would like to visit at most for each of those categories. They benefit from this approach because they can easily pick those types of POIs that are important to them, say museums, while quickly sorting uninteresting POIs out.

We first show, that *OPMPC* exposes a natural diminishing returns property, called *submodularity*. This concept has lately found wide application in the area of optimization algorithms. Based on the *submodularity* of our problem we present our heuristic algorithm, discuss its strengths but also its weaknesses.

The algorithm was implemented and thoroughly evaluated, benchmarks show that the approach taken is promising and the proposed itineraries are in the most cases close to optimal.

Abstract

Wir präsentieren eine effiziente Heuristik für das *Orienteering Problem (OP)* am Beispiel einer Reiseroutenplanung für den Toursimusbereich. Unser System schlägt den Touristen anhand ihrer Interessen personalisierte Routen vor, die sie von einem angegebenen Startpunkt aus hin zu einer Reihe sog. Points Of Interest (POIs) führt und schließlich zu dem gewünschten Endpunkt bringt. Da diese Problem zur Klasse der NP-harten Probleme gehört ist es unwahrscheinlich, dass ein effizienter Algorithmus zum Berechnen der optimalen Lösung gefunden werden kann.

Um hochwertige Reiserouten für Touristen vorschlagen zu können, generalisieren wir das *Orienteering Problem* indem wir jeden POI einer Kategorie zuordnen und erhalten so das *Orienteering Problem with Maximum Point Categories (OPMPC)*. Reisende suchen sich aus den gegebenen Kategorien jene aus die ihnen zusprechen und geben jeweils an wie viele POIs sie maximal zu dieser Kategorie sehen möchten, z.B. nicht mehr als 1 Restaurant soll in der Route vorhanden sein. Touristen profitieren von diesem Ansatz, weil sie so ganz einfach POIs aussortieren können die ihnen nicht gefallen.

Wir zeigen, dass OPMPC die Eigenschaft der Submodularität besitzt, eine Eigenschaft die in den letzten Jahren häufig bei Optimierungsalgorithmen angewendet wurde. Basierend auf der Submodularität unseres Problems präsentieren wir unsere Heuristik und beschreiben ihre Stärken, als auch ihre Schwächen.

Die vorgeschlagene Heuristik wurde implementiert und ausgiebig getestet, Benchmarks zeigen dass der Ansatz vielversprechend ist und in den meisten Fällen Lösungen generiert die nahezu optimal sind.

Abstract

Usando come esempio un'applicazione turistica presentiamo un'euristica efficiente per il *Orienteering Problem (OP)*. Il nostro sistema raccomanda ad un turista un itinerario personalizzato basandosi su preferenze e su requisiti temporali. Il sistema guida l'utente da un punto di partenza ad un punto di destinazione suggerendo di visitare una serie di attrazioni (Point of Interest o POI) durante il cammino. Questo problema è NP-Completo e di conseguenza probabilmente non esiste alcun algoritmo esatto ed efficiente.

Al fine di raccomandare buoni itinerari generalizziamo il OP assegnando ogni POI ad una categoria. Otteniamo così il *Orienteering Problem with Maximum Point Categories (OPMPC)* dove il viaggiatore esplicita le categorie che preferisce e quanti POI vuole al massimo visitare per ognuna.

Prima di tutto mostriamo che il *OPMPC* mostra la proprietà dei ritorni marginali decrescenti, formalizzata dalla *submodularità*. Questo concetto ha recentemente trovato ampie applicazioni negli algoritmi di ottimizzazione. Infine presentiamo la nostra euristica basata sulla submodularità e discutiamo i suoi punti di forza e debolezza.

L'algoritmo è stato implementato e testato accuratamente. I benchmarks mostrano che l'approccio preso è promettente e gli itinerari proposti sono nella maggior parte dei casi quasi ottimi.

Contents

Chapter 1

Introduction

The latest boom in the smartphone industry has brought small GPS devices to everyone's pocket. Location based services are rapidly growing in popularity and especially for tourism applications these new devices offer tremendous opportunities.

Usually tourists are new to the region and do not know the attractions, not to mention the best way to get there. To assist tourists in such situations we focus on efficient algorithms that recommend itineraries based on their preferences and time constraints. Given a starting point, for instance the current location, we guide the tourists through the streets of the city, visiting as many attractions, also called points of interest (POIs), as possible along the road until they reach their destination point within their time budget.

This problem is generally known as the *Orienteering Problem* (OP), a combinatorial problem having its roots in the *Knapsack Problem* (KP) and the *Traveling Salesman Problem* (TSP), all of them being NP-complete. In its original setting, OP assumes no scores, thus the problem boils down to visiting as many points as possible within the time budget. Whereas, if each POI has a score describing its attractiveness, our goal is to maximize the sum of the scores collected along the road. OP differs from TSP in that it is not required to visit every POI once and we do not search for the shortest path, but a tour having a maximum possible score. While the TSP is a minimization problem, our problem is concerned with maximizing the score. The *Orienteering Problem* is also different from KP, where the items being added to the knapsack have a constant weight and the order in which the items are added does not matter, whereas in OP the order of visited POIs is important.

In order to recommend sufficiently interesting POIs, the attractions are categorized and the users express their own preferences in terms of the maximum number of POIs they want to visit for a particular category, say no more than two museums and one restaurant. This information helps both, the users because they get a custom-tailored trip that hopefully meets the expectations and the system because it can reduce the possibly thousands of POIs to maybe some hundred POIs.

We start by formalizing the problem as a maximization problem in chapter 2, give an NP-completeness proof and introduce the notation we use throughout this paper. Still

in that chapter we analyze the problem more deeply and elaborate on the property of submodularity (section 2.3). Continuing in chapter 3 we describe our heuristic, called ARKTIS and present its pseudo code. In the next chapter we describe a state of the art algorithm, called eMIP and adapt it for our problem domain. Both algorithms were implemented, thoroughly evaluated and compared to each other in our benchmarks (chapter 5). Finally, we conclude with outlining what still needs to be worked on in the future.

Chapter 2

Problem Formalization

2.1 Model and Problem Definition

Consider a set of n POIs, denoted by $\mathbf{P}$ and containing $p_i, 1 \leq i \leq n$. Let s and d represent the starting and destination point, respectively. Then the n POIs together with s and d form the node set of a complete metric weighted undirected graph $G = (\mathbf{P} \cup \{s, d\}, \mathbf{E})$. Whereas the edges $e_l \in \mathbf{E} = \{(i, j)\}, 1 \leq i, j \leq n$ form the connections between the nodes. Each edge e_l has an associated weight $w_{i,j}$ that represents the travel time in seconds from p_i to p_j. Note that since the graph is undirected we have that $w_{i,j} = w_{j,i}$. Similarly, each POI p_i has a visiting time v_{p_i}, measured in seconds, as well as a score s_{p_i} that denotes the benefit of visiting POI p_i. Moreover, each p_i belongs to a category, for instance the POI *National History Museum* belongs to the category *museums*. The set of m categories is denoted by $\mathbf{K}$ and each POI p_i belongs to exactly one category $cat(p_i) = k_j, 1 \leq j \leq m$.

The user input consists of a time budget t_{max}, again measured in seconds and a rating of the categories. Let s_k denote the array of user scores for the categories, indexed by the ID of the category and holding the interest for that particular category, $s_k[i] \in [0, 1], 1 \leq i \leq m$, where 1 corresponds to the maximal possible interest. Additionally, the user provides the system with the maximum number of POIs that may be included in the final itinerary on a per category basis, let $max_k[i] \geq 0, 1 \leq i \leq m$. Before stating the optimization goal we first formally introduce the notion of an itinerary $\mathcal{I}$:

Definition 1 *An itinerary $\mathcal{I}$, starts from a starting point s, finishes at a destination point d[1] and includes an ordered sequence of connected POIs $\mathcal{I} = \langle s, p_{i_1}, p_{i_2}, \ldots, p_{i_q}, d \rangle$ that are visited at most once. We define the* cost *of itinerary $\mathcal{I}$ to be the total duration of the path from s to d passing through the POIs in $\mathcal{I}$, $cost(\mathcal{I}) = w_{s,i_1} + v_{p_{i_1}} + \sum_{j=2}^{q} (w_{i_{j-1},i_j} + v_{p_{i_j}}) + w_{i_q,d}$ and its* score *to be the sum of the scores of the individual POIs visited, $score(\mathcal{I}) = \sum_{j=1}^{q} s_{cat(p_{i_j})}$.*

[1] s and d may be identical.

As an output the user expects an itinerary $\mathcal{I}$ that respects the time budget t_{max} and the constraints imposed by the maximum visits max_k. The itinerary should be sufficiently interesting, measured in terms of its *score*.

We are ready to state the generalization of the *Orienteering Problem* and call it the *Orienteering Problem with Maximum Point Categories (OPMPC)*:

Definition 2 *Given a starting point s, a destination point d, n points of interest $p_i \in \mathbf{P}$, with visiting times $v_i, 1 \leq i \leq n$, traveling times $w_{s,i}, w_{i,d}, 1 \leq i \leq n$, and $w_{i,j}, 1 \leq i, j \leq n, i \neq j$, categories $k_j \in \mathbf{K}, 1 \leq j \leq m$ with scores s_{k_j}, and the following two parameters: (a) the maximum total time, t_{max} a user can spend on the itinerary and, (b), the maximum number of POIs max_{k_j} that can be used for each one of the k_j categories, a solution to the OPMPC is an itinerary $\mathcal{I} = \langle s, p_{i_1}, p_{i_2}, \ldots, p_{i_q}, d \rangle$, $1 \leq q \leq n$, such that*

- *the total score of the points, $score(\mathcal{I})$, is maximized;*
- *no more than max_{k_j} POIs are used for category k_j;*
- *the time constraint is met,* i.*e., $cost(\mathcal{I}) \leq t_{max}$.*

For the remainder of this section we will reason about the complexity of *OPMPC*. The original *Orienteering Problem* roots in the *Knapsack Problem* and the *Traveling Salesman Problem*, all of them being NP-complete. In the following proof we will show that *OPMPC* is a generalization of the original *Orienteering Problem* and thus NP-complete as well.

Theorem 2.1.1 *The Orienteering Problem with Maximum Point Categories (OPMPC) is NP-complete.*

Proof Given an itinerary $\mathcal{I}$, a bound t_{max}, and a bound S for the score, we can easily show that $cost(\mathcal{I}) \leq t_{max}$, $score(\mathcal{I}) \geq S$, and that no more than max_{k_j} POIs are used for category k_j in polynomial time. Thus, OPMPC $\in$ NP.

For the remainder of the proof we reduce the original Orienteering Problem to OPMPC by placing each point of interest into its own category k_i, $1 \leq i \leq n$ and setting max_{k_i} to 1. The score s_{k_i} of the category for p_i is set to the value of p_i in the original problem. Solving this instance of the OPMPC gives us an answer for a given Orienteering Problem, as the categories do not constrain the solution in any way. Consequently, OPMPC is a true generalization of the original orienteering problem.

2.2 Model Simplifications

In order not to over-complicate the problem we made some simplifying assumptions. For instance, a city's street network was modeled as an undirected graph where the POIs are the nodes and streets between them are the edges. But in reality, the street network of a city is a much more complex system than a simple undirected graph. In our assumptions

the edge weights are symmetric, so the cost to go from s to d is the same as to go the from d to s. However, in a real city this is hardly the case, as e.g. one-way streets may be used along the road.

For many graph related problems there are more efficient algorithms for undirected graphs than for directed graphs, for instance there is a constant factor approximation for TSP in undirected graphs but no comparable algorithm in the directed case (see section 3.4 for a more detailed discussion). Hence, we assume at the moment an undirected graph and leave the directed case for future work.

Moreover we assume that our graph G is metric, which means that G satisfies the triangle inequality, $w_{i,k} \leq w_{i,j} + w_{j,k}, \forall i, j, k$. As a consequence, the edge $w_{i,j}$ from p_i to p_j is actually the shortest path between these two nodes. This assumptions lead to a dramatic improvement in performance, since no shortest path computations have to be carried out in the algorithm itself. There are several algorithms to compute the all-pairs shortest path on a set of nodes and in our implementation we used the Floyd–Warshall algorithm. It has a time-complexity of $O(n^3)$ and space-complexity of $O(n^2)$, where n is the number of nodes, in our case POIs, in the graph.

2.3 Submodularity

In this section we first introduce the concept of *submodularity*, give evidence that this concept was successfully applied in optimization problems and then show that also *OPMPC* exposes the *submodularity* properties. Then, continuing in chapter 3 we show that by exploiting these properties we can give an efficient heuristic for *OPMPC*.

Due to its diminishing returns property, which we will see shortly, *Submodular Set Functions* have found wide application in the field of approximation algorithms, for both minimization and maximization problems. For instance, Krause et al. used the concept of *submodularity* in the fields of observation selection [1], Sviridenko studied the maximization of submodular functions subject to a knapsack constraint in [2] and *submodularity* has already been applied to the *Orienteering Problem* by Chekuri et al. in [3].

2.3.1 Definition

Consider the problem of placing cameras in a large, contorted building with many rooms. The task is now to find a placement such that the largest possible area is covered by the cameras. Intuitively, adding a camera to an empty room increases the coverage highly, but placing yet another camera in a room with already five cameras will not improve the surveillance greatly. Thus the problem of placing cameras exhibits a natural diminishing returns property, adding another camera to an already large set has lower impact than adding one to a small set[2].

This concept is known as *submodularity* and a set function f is called *submodular* if it satisfies the following two equivalent definitions [4]:

[2]This example was taken from [1] because it explains the concept of submodularity very well

- For every $A \subseteq B \subseteq \mathbf{P}$ and every $p \subseteq \mathbf{P} \setminus B$, it holds that $f(A \cup \{p\}) - f(A) \geq f(B \cup \{p\}) - f(B)$
- For every $A, B \subseteq \mathbf{P}$, it holds that $f(A) + f(B) \geq f(A \cup B) + f(A \cap B)$

Equivalently, a function f is called *modular* if both inequalities are changed to equalities. Let $\Delta(p \mid A) = f(A \cup \{p\}) - f(A)$ be the *marginal increase*, then the function f is *modular* iff $\Delta(p \mid A) = \Delta(p \mid B), \forall p \in \mathbf{P}$. This means basically that the *marginal increase* does not depend on the set A nor B but only on the element p. If the *inverse* inequalities hold for all A, B the function f is called *supermodular*.

2.3.2 Application in Orienteering

The *score* function of the OPMPC to compute the overall score of an itinerary $\mathcal{I}$ is also a *submodular set function*, the larger a set of POIs becomes the less likely it is that additional POIs can be introduced in an itinerary. Or put differently, the larger the set becomes the smaller is the *marginal increase* in score that can be obtained by adding yet another POI. Say you have a set of only two POIs and t_{max} is sufficiently large, then chances are good that a third POI can be added to an itinerary. However, if you have a set of 20 POIs adding another POI will possibly violate the time constraints and not increase the score at all.

In the following proofs Δ_s and Δ_c denote the *marginal increase* in terms of *score* and *cost*, respectively.

Lemma 2.3.1 *The score function of the OPMPC is submodular.*

Proof Let $A \subseteq B \subseteq \mathbf{P}$ be two sets of POIs and $p \in \mathbf{P} \setminus B$. The goal is to add POI p in order to increase the total score by the score of p, s_p, and to show that $\Delta_s(p \mid A) \geq \Delta_s(p \mid B)$. We distinguish the following three different cases:

1. p can be added to both, A and B
2. p can be added to both, A and B, but some lower-scoring POIs have to be removed
3. p cannot be added to B

In the first case p can be added to both sets and thus the marginal increase is the same for both, i.e. $\Delta_s(p \mid A) = \Delta_s(p \mid B) = s_p$.

The second case is more difficult, though, because we have to remove some lower-scoring POIs in favor of p. Therefore let $X \subseteq A$ and $Y \subseteq B$ be two sets of POIs with scores $s_X = \sum_{x \in X} s_x$ and $s_Y = \sum_{y \in Y} s_y$, such that $s_X \leq s_Y \leq s_p$. If the set Y exists we have that $\Delta_s(p \mid B \setminus Y) = s_p - s_Y$. The existence of the set Y implies also the existence of the set X of POIs, because if we can successfully build an itinerary out of $B \setminus Y \cup \{p\}$ then we can also build successfully an itinerary out of its subset $A \setminus X \cup \{p\}$. As a consequence $\Delta_s(p \mid A \setminus X) = s_p - s_X$. Recall that by construction $s_X \leq s_Y$, thus $\Delta_s(p \mid A \setminus X) \geq \Delta_s(p \mid B \setminus Y)$.

The third case is easier, because if we cannot add p to B then $\Delta(p \mid B) = 0$ and no matter if we add p to A or not, the marginal increase is greater than or at least equals to 0, thus $\Delta(p \mid A) \geq \Delta(p \mid B)$.

Next, let us study the properties of the second function of interest, the *cost* function, which computes the total duration going from s to d taking into account the visiting times of the POIs in between.

Lemma 2.3.2 *The cost function of the OPMPC is neither submodular, nor supermodular.*

Proof A simple counterexample shows that the *cost* function does not exhibit any *modularity* at all. Consider the graph in figure 2.1 with the four nodes $\{1, 2, 3, 4\}$ and edges with weight $w_{1,2} = 1$, $w_{1,3} = 3$, $w_{1,4} = 2$, $w_{2,3} = 2$, $w_{2,4} = 3$ and $w_{3,4} = 1$.

In the first example let $A = \{3, 4\}$, $B = \{2, 3, 4\}$. The itinerary starts from $s = 4$ and finishes at $d = 3$. In this setting the cost function satisfies the *submodularity* condition.

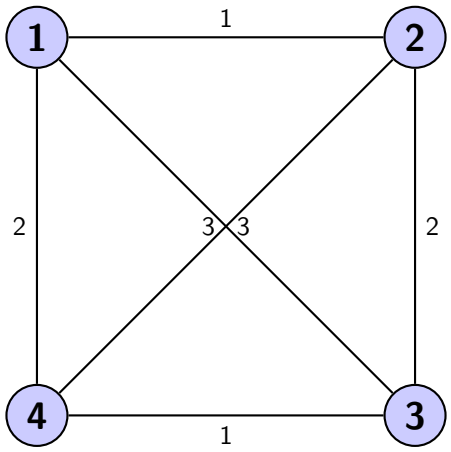

Figure 2.1: Graph containing $n = 4$ nodes

$$\Delta_c(1 \mid A) \geq \Delta_c(1 \mid B)$$
$$cost(A \cup \{1\}) - cost(A) \geq cost(B \cup \{1\}) - cost(B)$$
$$cost(\{4, 1, 3\}) - cost(\{4, 3\}) \geq cost(\{4, 1, 2, 3\}) - cost(\{4, 2, 3\})$$
$$(2 + 3) - 1 \geq (2 + 1 + 2) - (3 + 2)$$
$$4 \geq 0$$

However, in the following setting the *cost* function has *supermodular* behavior. Let $A = \{2, 4\}$, $B = \{2, 3, 4\}$, $s = 4$ and $d = 2$.

$$\begin{aligned}
\Delta_c(1 \mid A) &\leq \Delta_c(1 \mid B) \\
cost(A \cup \{1\}) - cost(A) &\leq cost(B \cup \{1\}) - cost(B) \\
cost(\{4,1,2\}) - cost(\{4,2\}) &\leq cost(\{4,3,1,2\}) - cost(\{4,3,2\}) \\
(2+1) - 3 &\leq (1+3+1) - (1+2) \\
0 &\leq 2
\end{aligned}$$

Both examples together show that the *cost* function is neither submodular, nor supermodular.

Chapter 3

ARKTIS Algorithm

Building on the concept of submodularity that we discussed in the previous chapter we start this chapter by explaining how we actually exploit this powerful property. In the course of the following sections we will then introduce our heuristic called ***A**lte**R**nating **K**napsack and **T**ravel**I**ng **S**alesman*, or ARKTIS in short.

The name ARKTIS already indicates how our approach to an efficient heuristic for orienteering works. However, the way how the Knapsack and the TSP problem are combined and how alternating approximations for both algorithms yields a heuristic, is the interesting part and also the main contribution of this chapter.

3.1 Modularity

As we have seen in section 2.3.2 it is not possible to predict how the *cost* function changes as new POIs are added to an itinerary, which makes the function difficult to approximate. However, in order to design an efficient approximation algorithm we need a *cost* function that is easier to deal with. Therefore in this section we introduce a *modular cost* function that will be an important building block of ARKTIS.

But before doing so, we first introduce in the following section the notion of a *root*, which is necessary to derive our new approximative cost function.

3.1.1 Root Construction

It lies in the nature of our problem that it is very dynamic and changing dramatically depending on the parameterization, e.g. increasing t_{max} only by some minutes may change the resulting itinerary completely. This makes the problem so difficult and fascinating at the same time.

In the hope that it facilitates the approximation we give the problem more structure by introducing a skeleton that every itinerary builds upon.

Definition 3 *Let a root $\mathcal{R} = \langle s, p_{i_1}, \ldots, p_{i_q}, d \rangle$ be the skeleton of an itinerary $\mathcal{I}$, that contains only the start and destination points s and d and a fixed number $q \geq 0$ of*

additional POIs. Each node of the root $\mathcal{R}$ is itself the root of a subtree of depth at most 1 and has as leaves other POIs $p \in \mathbf{P} \setminus \{p_{i_1}, \ldots, p_{i_q}\}$.

The root $\mathcal{R}$ can be easily modeled as a *directed* semi-Eulerian graph[1] $\hat{G} = (\hat{\mathbf{P}} \cup \{s, d\}, \hat{\mathbf{E}})$, where $\hat{\mathbf{P}}$ are all the POIs that belong to the root $\mathcal{R}$ and $\hat{\mathbf{E}}$ are the edges that connect this tree-like structure. Every root node p_i is connected to its next neighbor root node p_{i+1} by a single outgoing edge with weight $w_{i,i+1}$, while every leaf node is linked to its parent root node with both an incoming and outgoing edge. Based on the graph notation of the root $\mathcal{R}$ we can easily define the score and the cost of it:

Definition 4 *The score of a root $\mathcal{R}$ is defined as the sum of the scores of every single POI in $\mathcal{R}$, $score(\mathcal{R}) = \sum_{p_i \in \hat{\mathbf{P}}} s_{cat(p_i)}$. Similarly, the cost is defined as the weight of every single edge in $\hat{G}$ in addition to the visiting time of each POI, $cost(\mathcal{R}) = \sum_{w_i \in \hat{\mathbf{E}}} w_i + \sum_{p_i \in \hat{\mathbf{P}}} v_i$.*

The intuition behind this concept is to build several roots according to some selection strategy for the q POIs and then iteratively expand those skeletons until we get a sufficiently good itinerary that does not violate any constraints. Figure 3.1 shows a bare root with no additional POIs as leaves. In contrast, the root in figure 3.2 has 8 leaves. Note that, the edges between the root nodes $\{s, p_{i_1}, \ldots, p_{i_q}, d\}$ are colored in red, whereas the remaining edges in both directions are colored black.

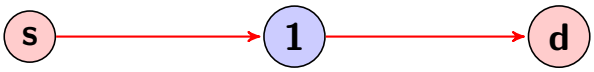

Figure 3.1: Bare root, without any leaves

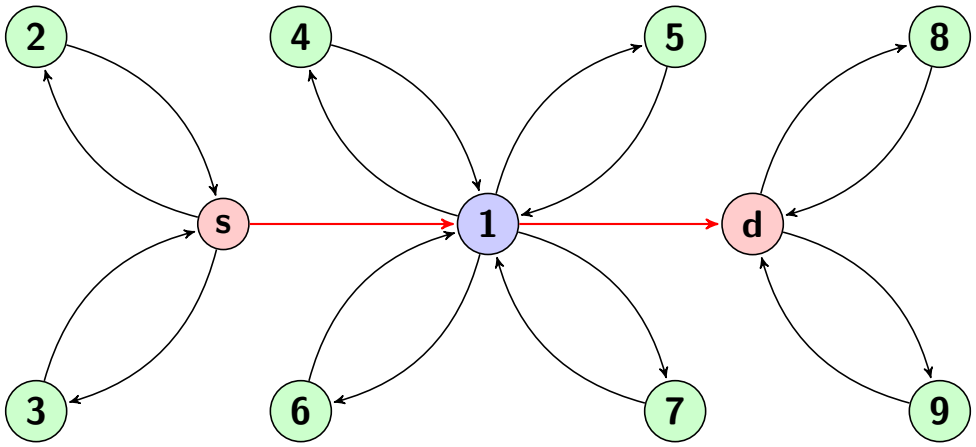

Figure 3.2: Root with 8 leaf POIs

[1] or Eulerian if $s = d$

The question which q POIs to choose remains still to be answered. In fact, the accuracy of ARKTIS depends heavily on the choice made in this step, and later in chapter 5 we will get into more detail.

What all strategies have in common is that the first root is always the direct connection between start and destination $\langle s, d\rangle$, i.e. $q = 0$. Algorithm 1 builds roots for q up to 1 by first sorting the POIs according to one of the utility functions described in section 3.5 and then selecting always the one with highest utility. This algorithm can be extended to work with $q > 1$ by taking the q POIs with highest utility and form a root out of them. However, suppose that $q = 2$ and p_1 and p_2 are the highest-ranked POIs, then one has to decide whether for example $\langle s, p_1, p_2, d\rangle$ and $\langle s, p_2, p_1, d\rangle$ are treated as different roots.

Algorithm 1 `Construct-Roots` ($\mathbf{P}$, $\mathbf{K}$, s, d)

Require: n POIs, m categories, start and destination point
Ensure: $m + 1$ roots with $q \leq 1$

1: $roots \leftarrow \{\langle s, d\rangle\}$
2: **for** (every category $k_i \in \mathbf{K}$) **do**
3: let $\mathcal{P} \subseteq \mathbf{P}$, s.t. $cat(p) = k_i, \forall p \in \mathcal{P}$
4: sort $\mathcal{P}$ according to some utility function in descending order
5: $roots \leftarrow roots \cup \{\langle s, \mathcal{P}[0], d\rangle\}$
6: **end for**
7: **return** $roots$

3.1.2 Modular Cost Function

Recall from section 2.3 that the *marginal increase* of a *modular set function* no longer depends on the set it operates on, but only on the element that is being added, i.e. $\Delta(p \mid \mathcal{A}) = \Delta(p \mid \mathcal{B}), \forall p \in \mathbf{P}, \mathcal{A} \subseteq \mathcal{B} \subseteq \mathbf{P}$.

The roots we previously defined are a simplification of an itinerary and when ARKTIS picks a new POI it is attached to the closest node in the root. Note that, a POI is never attached to a leaf node, but always to a root node. Knowing this, we can pre-compute for all $p \in \mathbf{P} \setminus \{p_{i_1}, \ldots, p_{i_q}\}$ the closest node in a root. As a result, we know in advance the *constant* cost of adding a certain POI to the root and therefore this cost function is modular. The procedure to pre-compute the constant costs is straightforward and shown in algorithm 2.

Since root r is a semi-Eulerian graph we can always find a tour that visits all edges once and such a tour is called an *Eulerian trail*. Clearly the cost of an Eulerian trail is necessarily at least as big as the TSP tour on graph $\hat{G}$, therefore the *cost* function is an overestimate. Unfortunately this modular cost function does not always capture reality very well, there are pathological cases were the modular cost diverges from the actual TSP cost quite heavily. In fact, as we are going to show next, the approximation can be arbitrarily bad.

Lemma 3.1.1 *The approximation of OPMPC's cost function through the cost function for a root $\mathcal{R}$ can be arbitrarily bad.*

Proof Consider the graph G in figure 3.3, which contains besides the start and destination node 4 POIs, figure 3.4 shows the respective root for graph G. Suppose that the visting times for all POIs is 0 and the max visits are such that an itinerary containing all POIs is feasible. Moreover assume that p is the number of leaf nodes in root $\mathcal{R}$, so in this graph $p = 3$. The optimal solution to the OPMPC problem is $\mathcal{I}_{opt} = \langle s, 1, 2, 3, 4, d \rangle$ and has a cost of $cost(\mathcal{I}_{opt}) = 4x + p\epsilon$, whereas the cost of root $\mathcal{R}$ is $cost(\mathcal{R}) = 2x + 2px$. Since p can be arbitrarily large and ϵ can be arbitrarily small the the approximation $cost(\mathcal{I}_{opt}) - cost(\mathcal{R})$ can be arbitrarily bad.

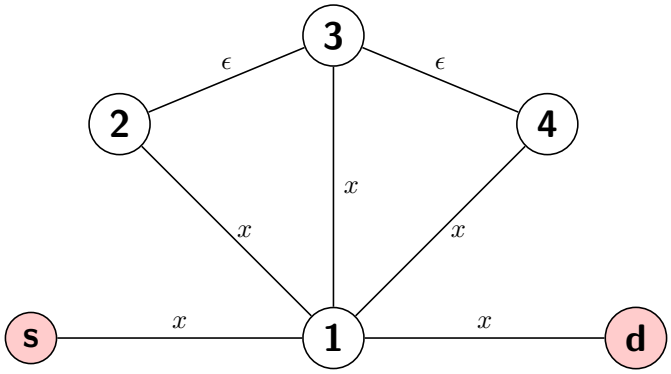

Figure 3.3: Graph G with 4 POIs

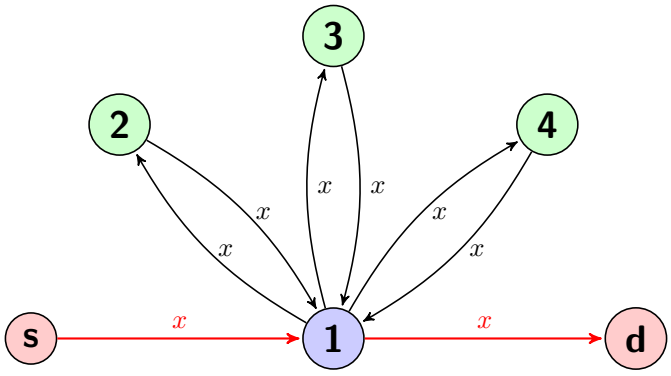

Figure 3.4: Root $\mathcal{R}$, where $cost(\mathcal{R}) \gg cost(\mathcal{I}_{opt})$

3.2 ARKTIS Algorithm

We have now all necessary pieces in place in order to explain the actual heuristic for OPMPC. We start this section by first giving a very high level introduction to the

Algorithm 2 `Constant-Cost` (**P**, *root*)

Require: The set of POIs and a *root*

Ensure: An n-ary array indexed by $p_i \in \mathbf{P}, 1 \leq i \leq n$ and holding the cost to the closest node in the root.

```
1: cost ← []
2: for (every p_i ∈ P) do
3:     min ← ∞
4:     for (every n_j ∈ root) do
5:         if w_{p_i,n_j} < min then
6:             min ← w_{p_i,n_j}
7:         end if
8:     end for
9:     cost[p_i] ← min
10: end for
11: return cost
```

algorithm such that one can easily grasp the intuition behind it and then continue to explain it in a more detailed fashion.

3.2.1 Intuition

The algorithm starts by creating a set of roots using algorithm 1. Our final itinerary will eventually build upon one of the roots, but which one is not known in advance, rather we figure it out during the course of the algorithm by checking which one generates the itineraries with highest score.

For each root we then continue to compute the *constant costs* with algorithm 2. By exploiting the *modularity* of our *constant costs* we know exactly to which root node a particular POI is attached. With this knowledge we can construct an itinerary $\mathcal{I}_{cur}$ out of the root and check whether it beats the current best itinerary $\mathcal{I}_{best}$.

Unfortunately, as we have seen our cost approximation is arbitrarily bad and we cannot know how far the calculated cost is from the real cost of the optimal itinerary $\mathcal{I}_{opt}$. This means the itinerary construction may generate a trip whose actual cost is much lower than t_{max}, i.e. the time budget is not really exhausted and we could potentially add more POIs. As a consequence we artificially increase the time budget t_{max} for the itinerary construction, here denoted t'_{max}, until the cost of the resulting itinerary $\mathcal{I}_{cur}$ is as close to t_{max} as possible. To find an appropriate value for t'_{max} we use a *binary search* approach to quickly narrow the search range.

Once we completed this process for each root, $\mathcal{I}_{best}$ contains the best itinerary found so far and we return it to the user. In the following few sections we will focus on each aspect of ARKTIS more deeply and finally present it as pseudo-code.

3.2.2 Itinerary Construction

The itinerary construction is the most important part of the algorithm as it is the piece of code which decides which POIs are chosen to be part of an itinerary and how the trip finally looks like. We recall that the Orienteering problem has its roots on the one hand in the *Knapsack Problem*, which takes care of the POI selection and the *Traveling Salesman Problem* for the routing aspect.

Our approach to the itinerary construction is simple, we first use a knapsack approximation with time budget t'_{max} to select among all POIs $\mathbf{P}$ a subset of POIs $\mathcal{S} \subset \mathbf{P}$ that has highest possible score, while no constraint is violated. Once the POI selection phase is over we can go ahead an make an itinerary out of this unordered set $\mathcal{S}$ using a TSP approximation. The goal is to find a trip that is as short as possible, because the shorter the itinerary is, the more remaining time we have to visit even more POIs, thus increasing the overall score.

Whenever we create an itinerary $\mathcal{I}_{cur}$ using the procedure just explained we compare it to the current best trip $\mathcal{I}_{best}$ and if it beats it in terms of score, i.e. $score(\mathcal{I}_{cur}) > score(\mathcal{I}_{best})$, we update the current optimum.

3.2.3 Binary Search

Since the unit of measurement of t'_{max} is time we have a natural ordering and can therefore easily apply a *binary search* for searching. Besides an ordered sequence of elements the binary search requires also a range where the value in question can be searched in. Unfortunately this range is not known in advance due to the lack of an approximation guarantee of our constant cost function. However, the binary search can be adapted such that the appropriate range is detected at run-time.

Before entering the actual binary search we initialize the search range to 0 as lower bound and t_{max} as upper bound. Then as long as we did not create an itinerary whose real cost is higher than t_{max} we double the search range by assigning the upper bound to the the lower bound and doubling the upper bound. As soon as this loop finished the range has been defined and the usual binary search can be used.

As usual we cut the search range in half, i.e. $t'_{max} = \frac{lo+hi}{2}$, perform an itinerary construction to obtain an itinerary $\mathcal{I}_{cur}$ and check its cost, $cost(\mathcal{I}_{cur})$. The decision whether to use the lower range $[lo, t'_{max} - 1]$ or the upper range $[t'_{max} + 1, hi]$ in the next turn of the loop is made by comparing the absolute time budget t_{max} with the cost of the resulting itinerary $\mathcal{I}_{cur}$. If the current itinerary is too expensive we obviously pick the lower range, otherwise we did not exhaust our time thoroughly and therefore use the upper range.

Since the search range is reduced in exponential steps the difference between the previous t'_{max} and the current t'_{max} becomes quickly very small. As a consequence the POI selection may return twice the same set $\mathcal{S}$, because with the few remaining seconds no more POIs can be visited. If this happens we stop our binary search.

3.2.4 Categories

When we construct an itinerary we should never forget about the constraints imposed by the categories. Especially the knapsack approximation has to take those constraints into account, see section 3.3 for more details. The categories have also an impact on the binary search, both on the initial search for the appropriate range and for the binary search itself. Previously, we mentioned that we try to get the cost of the itinerary as close as possible to t_{max}, but in some cases we cannot even get close to it because of the categories. Take for example a user input with t_{max} equals to 8 hours and only one category with maximum visits equals to 1, in this case it is not likely that we ever get even close to the time limit, as there are most probably not many POIs with such a high visiting time.

In that case the search for the appropriate search range would enter an infinite loop, as we are trying to look for an itinerary that has a higher cost than t_{max}, but this is impossible, since the categories do not allow such an itinerary. Therefore we have to add another stop condition, which covers this edge case. By doubling the search range one should observe a change in the POI selection through the knapsack approximation. However, if the set $\mathcal{S}$ remains unchanged in a cycle of the loop, we can assume that there is no more room for further optimization and can break the loop.

3.3 Knapsack Approximation

The optimization goal of OPMPC that we are trying to achieve is to maximize the score of an itinerary and the algorithm presented in this section is very important to achieve this goal, because it selects those POIs that are ultimately part of an itinerary.

The problem we are trying to solve in this section is in its core a *Knapsack Problem*, because we are given a set of POIs together with their visiting time and score, that resemble the items with weights and profits, and a time budget that we cannot exceed. The objective in the *Knapsack Problem* is to select a subset of the items such that the weight does not exceed the given budget and the profit is maximized.

Our requirements are different, tough, as the cost of a POI p is not only given by its visiting time v_p, but also by the time to reach that POI. Even worse, the travel time to reach p depends also on the position of that POI in the whole itinerary, so the ordering does matter. This is very different from the original *Knapsack Problem*, where the order of the items does not matter at all. But luckily, thanks to the modularity of our artificial cost function we are able to ignore the ordering completely, as we know in advance where to insert that particular POI in an itinerary. There is yet another difference to the original problem setting, namely the presence of POI categories and the maximum visits that a user provides as part of the input.

3.3.1 Greedy Algorithm

For the standard knapsack problem exists a greedy algorithm with an approximation guarantee of being no worse than 1/2 of the optimal solution [5]. The algorithm is

Algorithm 3 ARKTIS ($\mathbf{P}$, $\mathbf{K}$, s, d, t_{max}, $max_k[]$, $s_k[]$)

Require: n POIs, m categories, start and destination point, time budget t_{max}, constraints and scores.

Ensure: $\mathcal{I}$, an itinerary respecting all constraints with hopefully high score.

$\mathcal{I}_{best} \leftarrow \langle\rangle$
$roots \leftarrow$ CONSTRUCT-ROOTS($\mathbf{P}$, $\mathbf{K}$, s, d)
for (every $r_i \in roots$) **do**
    $cost' \leftarrow$ CONSTANT-COST($\mathbf{P}$, r_i)
    $lo \leftarrow 0$
    $hi \leftarrow t_{max}$
    $\mathcal{S} \leftarrow$ KNAPSACK-APPROXIMATION($\mathbf{P}$, $\mathbf{K}$, r_i, $cost'$, hi, max_k, s_k)
    $\mathcal{I}_{cur} \leftarrow$ TSP-APPROXIMATION($\mathcal{S}$)
    while ($\mathcal{S}$ changed $\wedge$ $cost(\mathcal{I}_{cur}) < t_{max}$) **do**
        $lo \leftarrow hi$
        $hi \leftarrow 2hi$
        $\mathcal{S} \leftarrow$ KNAPSACK-APPROXIMATION($\mathbf{P}$, $\mathbf{K}$, r_i, $cost'$, hi, max_k, s_k)
        $\mathcal{I}_{cur} \leftarrow$ TSP-APPROXIMATION($\mathcal{S}$)
    end while
    while ($\mathcal{S}$ changed) **do**
        $t'_{max} \leftarrow \frac{lo+hi}{2}$
        $\mathcal{S} \leftarrow$ KNAPSACK-APPROXIMATION($\mathbf{P}$, $\mathbf{K}$, r_i, $cost'$, t'_{max}, max_k, s_k)
        $\mathcal{I}_{cur} \leftarrow$ TSP-APPROXIMATION($\mathcal{S}$)
        if $score(\mathcal{I}_{cur}) > score(\mathcal{I}_{best}) \wedge cost(\mathcal{I}_{cur}) \leq t_{max}$ **then**
            $\mathcal{I}_{best} \leftarrow \mathcal{I}_{cur}$
        end if
        if $cost(\mathcal{I}_{cur}) > t_{max}$ **then**
            $hi \leftarrow t'_{max} - 1$
        else
            $lo \leftarrow t'_{max} + 1$
        end if
    end while
end for
return $\mathcal{I}_{best}$

straightforward and is here just informally summarized: sort the items in non-increasing order by *efficiency*, which is the profit divided by the weight. Then take the items with highest *efficiency* as long as the budget is not exceeded, once this happens sum up the score of those items and check whether it is larger than the score of the next remaining item. If that is the case, return the first items, otherwise return the single next item.

Due to the presence of categories and maximum number of visits per category we cannot exactly use this greedy approach and have to adapt it slightly. But before looking at the algorithm itself, we first look at the rather long list of parameters. Besides the

obvious parameters $\mathbf{P}$ and $\mathbf{K}$, which are the POIs and the categories, the approximation algorithm takes also a root r, as it is the backbone of the itinerary that we are trying to fill up with POIs, an array of constant costs, the time budget t_{max}, the max visits and the scores. In contrast, the output is simply a set $\mathcal{S}$ of POIs that have been selected to be in the respective itinerary. Note that the set $\mathcal{S}$ is unordered, i.e. the order in which the POIs appear does not resemble the actual order of the POIs in the itinerary. The task of ordering the POIs is left to the TSP approximation in the next section.

Having the in- and output of the procedure defined, we can continue with the actual explanation of the algorithm itself. Before the algorithm can choose a POI, it has to make sure that the maximum visits constraint is not violated. To do so, the algorithm has an array of length m, called *visits*, which contains for each category k_i, $1 \leq i \leq m$, the number of POIs that have already been selected. This array is first initialized to all 0 and in a second step we have to take care of the POIs that are already part of the itinerary, namely the POIs of the root r, denoted by $\hat{\mathbf{P}}$. For each $p_i \in \hat{\mathbf{P}}$ we increment $visits[cat(p_i)]$ by 1. We continue by initializing the output set $\mathcal{S}$, pre-filled with the POIs of the root. Similarly to the standard knapsack approximation algorithm, where the items are ordered by *efficiency*, we order the POIs by the constant cost *utility* function (utility 3.3 defined in section 3.5). Then we continue by trying to insert every $p_i \in \mathbf{P}$ into $\mathcal{S}$ by checking whether neither the time nor the maximum visits constraints are violated.

Unlike the standard algorithm, in our problem setting we cannot take the first k items until the budget is violated, because we also have to take the categories into account. Therefore we cannot guarantee that this modified algorithm has an approximation guarantee of 1/2. In both algorithms the sorting algorithm for the items is the dominant factor in terms of runtime with an asymptotic complexity of $O(n \log(n))$ and therefore the overall complexity is $O(n \log(n))$ as well.

3.4 TSP Approximation

The *Traveling Salesman Problem* is probably the best known NP-hard problem, many new optimization techniques have first been applied to it because it serves as a good performance benchmark. Its popularity lies in the fact that the problem is intuitive and very easy to state: given a set of cities and its pairwise distances, find the shortest tour that visits each city once and returns to the origin city. The TSP can be modeled as a graph problem where the vertices are the cities and the paths between them are the edges of the graph. Each edge has an associated weight which is the length of the path between the two cities.

The TSP arises in many areas, especially in logistics where for example one tries to deliver packets as quickly as possible. It is also a sub-problem of many other problems, as it is, for instance, the case with the *Orienteering Problem*. If we have two itineraries $\mathcal{I}_1$ and $\mathcal{I}_2$ with $cost(\mathcal{I}_1) < cost(\mathcal{I}_2)$, both visiting the same set of POIs but in a different order, we prefer $\mathcal{I}_1$, because with the additional remaining time we might be able to visit yet another POI, thereby increasing the overall score.

Algorithm 4 `Knapsack-Approximation` ($\mathbf{P}$, $\mathbf{K}$, r, $cost$, t_{max}, $max_k[]$, $s_k[]$)

Require: n POIs, m categories, a *root*, constant costs, time budget, max visits and scores

Ensure: A set of selected POIs

let $\hat{\mathbf{P}}$ be the POIs of root r
$visits \leftarrow []$
for (every $c_i \in \mathbf{K}$) **do**
    $visits[c_i] = 0$
end for
for (every $p_i \in \hat{\mathbf{P}}$) **do**
    $visits[cat(p_i)] \leftarrow visits[cat(p_i)] + 1$
end for
$\mathcal{S} \leftarrow \hat{\mathbf{P}}$
sort $\mathbf{P}$ by *utility* in descending order
for (every $p_i \in \mathbf{P} \setminus \hat{\mathbf{P}}$) **do**
    if $cost(r) + cost[p_i] + v_{p_i} \leq t_{max} \wedge visits[cat(p_i)] < max_k[cat(p_i)]$ **then**
        $\mathcal{S} \leftarrow \mathcal{S} \cup \{p_i\}$
        add p_i to root r
        $visits[cat(p_i)] \leftarrow visits[cat(p_i)] + 1$
    end if
end for
return $\mathcal{S}$

A TSP solution is in the same time a solution to the *Hamiltonian Cycle* problem as well, because it is a cycle that includes every graph vertex exactly once. However, in our problem most of the times the tourists will plan an itinerary that takes them from one point in the city to another, i.e $s \neq d$. While the general TSP was studied thoroughly, the generalization with source and destination point not necessarily being the same, has not yet been mentioned much in research. As we will see soon, Christofides' TSP approximation algorithm that we used in our implementation has a worse approximation guarantee in the case where $s \neq d$.

In the general case where we do not know anything about the properties of the graph, it has been proven that TSP cannot be approximated within a factor of $\alpha(n)$, unless $P = NP$ [6], which is in fact a very strong nonapproximability result. However, our graph is embedded in a metric space, since we pre-computed the all-pairs shortest path at the very beginning and thus the triangle inequality holds. Luckily, for the metric TSP there is a constant-factor approximation algorithm due to Nicos Christofides.

3.4.1 Christofides' Approximation Algorithm

In this section we will discuss Christofides' approximation algorithm for the metric TSP where $s = d$ [7] and an extension for the $s \neq d$ case. This algorithm achieves a 3/2

approximation guarantee and is over 35 years after its publication still the best known performance guarantee for the metric TSP. We just give a very high-level introduction to the algorithm, as the algorithms that this approximation builds upon are well known and not very relevant to our research.

The algorithm consists of 4 steps, first find a Minimum Spanning Tree (MST) on the graph G, using for instance Kruskal's algorithm. In a second step, take all odd-degree vertices of the resulting MST and build a Minimum Cost Perfect Matching with the Blossom algorithm. Combine the MST and the matching result to obtain an Eulerian graph, since each node has now an even degree. Find a Eulerian tour in that graph using for example Hierholzer's algorithm resulting in a walk that uses every edge of it. Note that this walk may visit nodes several times, thus it is shortcut by taking the vertices in the order of their first appearance. The shortcutting procedure can only reduce the cost of the walk, since the triangle inequality holds. The resulting walk is at most 3/2 times as long as the optimal TSP path.

Algorithm 5 `Christofides-Approximation` (G)

Require: complete graph G
Ensure: TSP tour

1: Find an MST of G and denote it by T
2: Compute a Minimum Cost Perfect Matching, M, on the set of odd-degree vertices of T
3: Merge M and T, and find an Euler tour, $\mathcal{T}$, on the resulting graph
4: Shortcut $\mathcal{T}$, such that each vertex is visited only once
5: **return** $\mathcal{T}$

With some easy adjustments it is possible to extend Christofides' algorithm to the case where $s \neq d$ [8]. However, the approximation guarantee slightly increases from 3/2 to 5/3.

Step 1 of the original algorithm remains unchanged, while in the seconds step we do not take the vertices with *odd*-, but *wrong*-degree. A vertex is of wrong degree if it belongs either to $\{s, d\}$ and has even degree or if it is an intermediate vertex and has odd degree. With the resulting wrong-degree nodes we compute the Minimum Cost Perfect Matching and again combine it with the minimum spanning tree. The resulting graph is no longer Eulerian, because the two nodes s and d are of odd-degree. However, this is not a problem, since we are now looking for a *path* instead of a *cycle*. Finding an Eulerian path can again be done with Hierholzer's algorithm, with a slight adjustment. The fourth and last step remains the same as before and the result is a path that is no worse than 5/3 times the optimal TSP tour.

Since the most expensive step in both algorithms, namely the Minimum Cost Perfect Matching with complexity $O(n^3)$, remains unchanged, also the overall asymptotic complexity of both approximations does not change either and remains $O(n^3)$.

Algorithm 6 `Christofides-Approximation` (G)

Require: complete graph G
Ensure: TSP tour

1: Find an MST of G and denote it by T
2: Compute a Minimum Cost Perfect Matching, M, on the set of *wrong*-degree vertices of T
3: Merge M and T, and find an Euler *path*, $\mathcal{T}$, on the resulting graph
4: Shortcut $\mathcal{T}$, such that each vertex is visited only once
5: **return** $\mathcal{T}$

3.4.2 Implementation Details

A usual itinerary of a tourist is half a day or maybe a day long and visits up to 7 or 8 attractions approximately. This assumption is not based at all on any kind of empirical evidence, but on personal experience.

Solving a TSP instance optimally for up to such small numbers of POIs can be done reasonably quickly on today's hardware. Assuming that most of the times the TSP approximation step in ARKTIS is working on 7 or less POIs we can easily gain more precision by doing an optimal computation if we stay under a certain threshold and switch to the TSP approximation otherwise.

There are many algorithms to compute TSP optimally, as for instance by simply enumerating all possible permutations of the vertices, which has an asymptotic complexity of $O(n!)$, i.e. for 7 POIs this corresponds to only 5040 different combinations. In fact, we use this simple bruteforce algorithm, because for small number of POIs this approach is still efficient enough. There are, of course, more efficient techniques for solving small TSP instances optimally, e.g. the family of branch and bound algorithms. Please check the experimental evaluation in chapter 5 to see that the performance of our heuristic ARKTIS algorithm does not suffer greatly by using an optimal algorithm for a NP-hard problem in small problem instances.

Algorithm 7 `TSP-Approximation` $(\mathcal{S})$

Require: a set of POIs
Ensure: TSP tour

1: Let E be the edge-weights between each pair of POIs in $\mathcal{S}$
2: $graph \leftarrow G(\mathcal{S}, E)$
3: **if** $|S| \leq 7$ **then**
4: **return** TSP-Optimal($graph$)
5: **else**
6: **return** TSP-Approximation($graph$)
7: **end if**

3.5 Utilities

The accuracy of ARKTIS depends mostly on two decisions, on the one hand which POIs are chosen in the root construction (algorithm 1) and on the other hand how well the knapsack approximation (algorithm 4) performs. In both algorithms we try to pick those POIs that are likely to be part of a high-scoring itinerary and in this section we discuss different such POI selection strategies.

We have already seen that each POI has a score describing its attractiveness, but this measure seems too static since it does not take into account any user information. Moreover, the score by itself does not describe how well a POI p fits into an itinerary, because if p has a very high score but is close to the city's periphery it might drag an itinerary far away from the center where most of the interesting POIs are probably located. Therefore we assign each POI $p_i \in \mathbf{P}, 1 \leq i \leq n$ a *utility* value u_i and both algorithms will then select those POIs with the highest utility.

Remember that part of the user input are the scores that the tourist assigns to each category, $s_k[i] \in [0,1], 1 \leq i \leq m$. Let $\alpha \in [0,1]$ be a parameter describing the influence of the user scores for the single categories, then we can define our first utility that relies on both, category scores from the users and background information for each single POI provided by e.g. the tourism office.

$$u_1(p_i) = \alpha s_k[cat(p_i)] + (1-\alpha)s_{p_i} \tag{3.1}$$

This utility, however, still suffers from the same problem described above, namely that the location of a POI is not taken into account. To circumvent this problem, we define yet another utility function that builds upon the previously defined utility and additionally takes into account the time to reach that particular POI, as well as its visiting time. The following "location-aware" utility favors those POIs that are closer to the direct connection between the start and the destination point.

$$u_2(p_i, s, d) = \frac{u_1(p_i)}{w_{s,p_i} + v_{p_i} + w_{p_i,d}} \tag{3.2}$$

In the knapsack approximation the notion of "location-awareness" is different, though, because unlike in the previous utility, we precomputed the cost of adding some POI to a root already in the constant cost computation phase. Therefore the knapsack utility looks like this:

$$u_3(p_i, cost) = \frac{u_1(p_i)}{cost[p_i]} \tag{3.3}$$

There are clearly other utility functions which weight the POIs differently and each of them will work in a certain problem setting better than the others, but generally speaking the utilities that we have defined so far are sufficiently good as our benchmarks in chapter 5 suggest.

3.6 Complexity Analysis

The worst case complexity analysis of ARKTIS is difficult for one reason. Recall that the cost approximation is arbitrarily bad and therefore we do not know how much larger our approximated cost is compared to the real cost. As a result we perform a search for the lower and upper bounds for the binary search. However, there is no theoretical bound on how often the range has to be doubled before we find the appropriate bounds, due to the lack of an approximation guarantee. Therefore, we cannot give any worst case upper bound of the algorithm runtime.

The best case analysis is easier, though, assuming that the appropriate bounds of the binary search are 0 and t_{max}, respectively. Then the binary search has the well known worst case complexity of $\mathrm{O}(\log n)$, where n is the size of the range an corresponds to t_{max} in our case. The most expensive computation inside the binary search is the TSP approximation with complexity $\mathrm{O}(n^3)$, with n being the number of POIs in the input. The binary search is executed $k+1$ times, as the root construction yields $k+1$ roots, where k is the number of categories. This suggests that the overall best case runtime complexity is $\mathrm{O}(k \log(t_{max}) n^3)$.

Even though the worst case runtime is not known, experiments described in chapter 5 have shown that the runtime was in all benchmarks acceptable, also for large problem instances.

Chapter 4

eMIP Algorithm

In order to have a comparison for our algorithm we have implemented the eMIP Algorithm presented by Singh et al. in [9], because it is a state of the art algorithm and it has been used also in practice. This algorithm builds on the Recursive Greedy algorithm presented by Chekuri et al. in [3].

In this chapter we briefly describe their algorithm and show how we adapted it to our problem domain.

4.1 Original Problem Setting

The original problem setting in which this algorithm found practical application at first glance seems different from our itinerary planning. Basically, Singh et al. want to monitor algae biomass in a lake in order to prevent its pollution. They have a couple of small robotic boats available that they can use to take measurements all around the lake. The task is to plan the paths of the boats in such a way that the most information is collected. With the resulting information, including the temperature at each measuring point, they can judge the degree of pollution in the lake. A natural constraint on the length of such a path is the storage battery energy that is consumed during the movement of the boat as well as during the measurement phase.

Our application is different in some aspects, for example, we are (at least for now) not interested in planning itineraries for multiple groups of tourists. Instead, one could extend this idea to plan multiple itineraries not for the same day, but for say a week. Obviously tourists do not want to visit POIs over and over again when they use the software multiple times in their holidays.

In their problem setting, taking a measurement imposes a certain cost, which corresponds to the visit time of a POI in our domain and our budget constraint is measured in time instead of the limited energy available for the boats.

4.2 Intuition

The original Recursive Greedy algorithm by Chekuri et al. works as follows: Given a start and destination point their algorithm splits the path from s to d in two parts, one from s to m and the other from m do d, and then tries to optimize each part recursively. This splitting is done by trying each point that has not been visited yet as middle point. Additionally, both halves get a portion of the overall time budget assigned, but the optimal split for that budget is not known in advance, therefore they try each budget split from 0 up to the time budget.

To control the maximum depth of the recursion they pass a counter, called iter, to the algorithm which is then reduced by 1 in every recursive call. The base case occurs once iter reaches 0 and in that case simply the path from s to d is returned. To avoid that a POI is chosen multiple times in different branches of the recursion tree a set $\mathbf{X}$, called the residual, is passed into the function. The first recursive call within the algorithm yields a set $\mathcal{A}$ of POIs and the residual for the second recursive call is updated to contain also the POIs in $\mathcal{A}$, i.e. $\mathbf{X} \leftarrow \mathbf{X} \cup \mathcal{A}$.

This algorithm has only a quasi-polynomial runtime and is therefore not very efficient. This is the reason why Singh et al. introduced a couple of changes to make the algorithm run faster. Due to the large number of measuring points on a lake trying each point as a middle point m in each recursive call is infeasible and therefore they do a spatial decomposition of the lake into cells. Every cell is basically a square of the same size. Their modified algorithm then chooses among all cells a middle cell and in the base case they do a greedy selection of points in the start and destination cells. The greedy selection works similarly to the knapsack approximation that we use in our algorithm.

Since the authors assume that the traveling time within a cell is 0, their heuristic may yield results that violate the budget constraint by a certain factor. According to definition 2 a solution $\mathcal{I}$ to OPMPC must not violate the budget constraint, i.e. $cost(\mathcal{I}) \leq t_{max}$ and therefore the eMIP algorithm may not always generate valid OPMPC solutions.

Despite the spatial decomposition the algorithm has still super-polynomial runtime and therefore in an additional attempt to reduce the asymptotic complexity they apply certain branch-and-bound techniques to cut the search space.

4.3 Spatial Decomposition

Our approach to the spatial decomposition differs in that we do not split the network of a city into cells of equal size, rather we perform a clustering of the POIs. We think this suits our application better, because a city network is usually more heterogeneous as some areas are more densely populated with POIs than others, whereas the measuring points on a lake are more uniformly distributed.

The hierarchical clustering method that we use is straightforward, given a number k, representing the number of desired clusters, the algorithm initially puts every POI into its own cluster. Then as long as we have not reduced the number of clusters to k, we compute a metric between all pairs of clusters and join those with the smallest metric.

The code for the algorithm and the metric are shown in in 8 and 9, respectively.

We use the mean distance between each element of two clusters as a metric to describe the closeness of two clusters. In mathematical notation this corresponds to a binary function which is given two clusters:

$$metric(A, B) = \frac{1}{|A| \cdot |B|} \sum_{x \in A} \sum_{y \in B} d_{x,y} \tag{4.1}$$

From a performance point of view it is reasonable to have only a small number k of clusters, because the algorithm will make fewer recursive calls. However, the algorithm will improve in accuracy for larger values for k. This is the usual trade-off between performance and accuracy and in our experiments we have seen that the square root of the number of POIs, i.e. $k = \sqrt{|\mathbf{P}|}$ works well.

Algorithm 8 `Spatial-Decomposition` $(\mathbf{P})$

Require: a set of POIs

$k = \sqrt{|\mathbf{P}|}$
$clusters = \text{list}()$
for $(p \in \mathbf{P})$ **do**
    append$(clusters, \{p\})$
end for
$cluster1 \leftarrow \{\}$
$cluster2 \leftarrow \{\}$
while (length$(clusters) > k$) **do**
    $min \leftarrow \infty$
    for $(i = 0; i <$ length$(clusters); ++i)$ **do**
        for $(j = i + 1; j <$ length$(clusters); ++j)$ **do**
            $metric \leftarrow$ METRIC$(clusters[i], clusters[j])$
            if $(metric < min)$ **then**
                $min \leftarrow metric$
                $cluster1 \leftarrow clusters[i]$
                $cluster2 \leftarrow clusters[j]$
            end if
        end for
    end for
    remove$(clusters, cluster1)$
    remove$(clusters, cluster2)$
    append$(clusters, cluster1 \cup cluster2)$
end while
return *clusters*

Algorithm 9 `Metric` ($cluster1$, $cluster2$)

Require: two distinct clusters

1: $sum \leftarrow 0$
2: $length1 \leftarrow \text{length}(cluster1)$
3: $length2 \leftarrow \text{length}(cluster2)$
4: **for** $(i = 0; i < length1; ++i)$ **do**
5: **for** $(j = 0; j < length2; ++j)$ **do**
6: $sum \leftarrow d(cluster1[i], cluster2[j])$
7: **end for**
8: **end for**
9: **return** $\frac{sum}{length1+length2}$

4.4 Budget Splitting

Besides splitting the path between start and destination point in two halves, the algorithm splits the available visit time also in two parts, one for the first recursive call and the other for the second recursive call. The algorithm tries several splits until it finds one which yields a good solution. Singh et al. describe two splitting strategies, the one-sided and two-sided budget splits. We will focus on the one-sided for now. The idea is to start from 0 and then go in exponential steps to the base 2 towards some budget limit B, i.e. $\{0, 2^0, 2^1, 2^2, \ldots, B\}$. This guarantees that the chosen budget is no more than twice as large as the optimal split. The two-sided splits are simply the one-sided splits starting from both sides, on the one side incrementally adding to 0 and on the other side subtracting from B, i.e. $\{0, 2^0, 2^1, 2^2, \ldots, B\} \cup \{B - 0, B - 2^0, B - 2^1, B - 2^2, \ldots, 0\}$.

Since we measure our budget limit in seconds this approach feels somewhat unnatural, because we initially waste much time with very small values like 1 second, 2 seconds etc., but than due to the exponential growth the values become soon too large. Therefore we introduce a scaling factor, denoted BUDGET_SCALE_FACTOR, which allows us to scale the seconds to more reasonable time intervals. For example, setting the scale factor to 300 (5 minutes) yields the following splits: $\{0, 300, 600, 1200, \ldots, B\}$.

Moreover, to get a flatter growth curve we can change the base of the exponential steps, where the default base still remains 2. Reducing the number means that more splits are tried and thus the runtime is negatively affected, but again the accuracy may improve.

The pseudo code to calculate the one-sided budget splits for a given budget limit is shown in algorithm 10 and reasonable default values for the just described constants are shown in table 4.1.

Name	Default Value
BUDGET_SCALE_FACTOR	300 (i.e. 5 minutes)
BUDGET_SPLIT_GROWTH_RATE	2

Table 4.1: eMIP default values for constants

Algorithm 10 `Budget-Splits` (*budget*)

Require: *budget* in seconds

1: $base$ = BUDGET_SPLIT_GROWTH_RATE
2: $rate$ = BUDGET_SCALE_FACTOR
3: $scaledBudget \leftarrow \lceil \frac{budget}{rate} \rceil$
4: $nrSplits \leftarrow \lceil \log_{base} scaledBudget \rceil + 2$
5: let $splits$ be an array of length $nrSplits$
6: $splits[0] = 0$
7: $splits[nrSplits - 1] = budget$
8: **for** $(i = 1; i < nrSplits - 1; ++i)$ **do**
9: $\quad splits[i] = base^{i-1} \cdot rate$
10: **end for**
11: **return** $splits$

4.5 Greedy Subset

Algorithm 11, called Greedy Subset, is the procedure which selects the POIs in the eMIP algorithm. The name Greedy Subset comes from the fact that the algorithm greedily selects those POIs that have the highest *marginal increase* in terms of score, thereby exploiting the submodularity of the score function. This idea is very similar to the knapsack approximation that we used in our algorithm (see section 3.3). Therefore, we just briefly describe the algorithm on a very high level.

Initially the POIs are sorted according to score, which represents the order of highest marginal increase. Then as long as we have not exhausted our time budget we pick POIs from the head of the sorted list. POIs are only selected, however, if they do not violate any constraint, time-wise or due to the max visits per category. The residual **X** helps to not visit a POI multiple times in the recursion tree and to add only those POIs whose max visits allow that. At the end we return the set of POIs that we have selected greedily.

4.6 eMIP Algorithm

The eMIP algorithm consists of two parts, a non-recursive and a recursive function. The first one, called eMIP, does the initialization and then in turn calls the recursive-eMIP

Algorithm 11 `Greedy-Subset` (**P**, **X**, $visitTime$)

Require: a set of unvisited POIs, residual containing visited POIs, visit time

```
1: time ← 0
2: i ← 1
3: selection ← {}
4: sort P according to score in descending order
5: while (i ≤ length(P) ∧ time < visitTime) do
6:     p ← P[++i]
7:     if (p ∈ X) then
8:         continue
9:     end if
10:    if (time + v_p > visitTime) then
11:        continue
12:    end if
13:    if (p violates the category constraint) then
14:        continue
15:    end if
16:    selection ← selection ∪ {p}
17:    time ← time + v_p
18: end while
19: return selection
```

function. Part of the initialization is for instance the spatial decomposition, where the POIs are grouped together in clusters. Every cluster is represented by a centroid, which is the POI that has the smallest accumulated distance to all other POIs in that particular cluster.

The eMIP algorithm has a clear distinction between time spent for traveling and for visiting POIs. It is the task of the non-recursive part to initially split the overall time budget t_{max} into $travelTime$ and $visitTime$. It does so by using the budget split function described in section 4.4. For each such split the recursive part is called.

Still before calling the recursive part, the algorithm makes sure that the destination cluster $\mathbf{C}_d$ is reachable from the start cluster $\mathbf{C}_s$. It also initializes the residual to an empty set, because so far no POIs have been selected. If the itinerary returned by the recursive-eMIP call is better than the currently best itinerary in terms of score, then the best itinerary is updated. The route of the best itinerary may not be optimal, therefore before returning the itinerary to the user, it is first smoothed using Christofides' TSP approximation described in section 3.4.

Now we focus on the recursive-eMIP part, which first checks whether it is even possible at all to add a POI by comparing the available $visitTime$ to the $smallestVisitTime$

of all POIs and returns an empty itinerary if it is not the case. It then proceeds with computing the greedy subset of the POIs in the start and destination clusters $\mathbf{C}_s \cup \mathbf{C}_d$. The resulting POI selection serves as a reference point of what is achievable given the current parameters. In case the parameter iter is 0, i.e. the maximum recursion depth has been reached and the algorithm simply returns the solution of the greedy subset selection.

Otherwise the algorithm loops through all possible middle clusters and all the budget splits returned by the function described in section 4.4. For each such configuration the recursive function is called twice, once from the start cluster to the middle cluster and then from the middle cluster to the destination cluster. The second recursive call learns about the results of the first recursive call by updating the residual $\mathbf{X}$, this guarantees that no POI is selected twice. As a result we obtain two sub-itineraries and if they have together a higher score than the currently best known itinerary we concatenate them and update the current best itinerary.

Algorithm 12 `eMIP` ($\mathbf{P}$, t_{max}, s, d)

Require: n POIs, budget, start and destination point

let $smallestVisitTime$ be the smallest visit time of all $p \in \mathbf{P}$
$\mathbf{C} \leftarrow$ SPATIAL-DECOMPOSITION($\mathbf{P}$)
let $\mathbf{C}_s$ and $\mathbf{C}_d$ be two new empty clusters around s and d
find a centroid for every cluster

$\mathcal{I}_{best} = \langle\rangle$
$splits \leftarrow$ BUDGET-SPLITS(t_{max})

for ($iter = 0; iter < \text{length}(splits) - 1; ++iter$) **do**
    $travelTime \leftarrow splits[iter]$
    $visitTime \leftarrow t_{max} - travelTime$

    if ($d(\mathbf{C}_s, \mathbf{C}_d) > travelTime$) **then**
        continue
    end if

    $\mathcal{I} \leftarrow$ RECURSIVE-EMIP($\mathbf{C}_s$, $\mathbf{C}_d$, $visitTime$, $travelTime$, {}, $iter$)
    if (score($\mathcal{I}$) > score($\mathcal{I}_{best}$)) **then**
        $\mathcal{I}_{best} \leftarrow \mathcal{I}$
    end if
end for

$\mathcal{I}_{best} \leftarrow$ TSP-APPROXIMATION($\mathcal{I}_{best}$)
return $\mathcal{I}_{best}$

Algorithm 13 `recursive-eMIP` ($\mathbf{C}_s$, $\mathbf{C}_d$, $visitTime$, $travelTime$, $\mathbf{X}$, $iter$)

Require: start and destination clusters, visit time, travel time, the residual containing all visited POIs so far, maximum recursion depth

if $(visitTime < smallestVisitTime)$ **then**
  return $\langle\rangle$
end if
$nextTravelTime \leftarrow \frac{1}{2} travelTime$
$\mathcal{I} \leftarrow$ GREEDY-SUBSET($\mathbf{C}_s \cup \mathbf{C}_d$, $\mathbf{X}$, $visitTime$)
if $(iter = 0)$ **then**
  return $\mathcal{I}$
end if
for $(c \in \mathbf{C} \setminus \{\mathbf{C}_s, \mathbf{C}_d\})$ **do**
  if $(d(\mathbf{C}_s, c) > nextTravelTime \vee d(c, \mathbf{C}_d) > nextTravelTime)$ **then**
    continue
  end if
  $splits \leftarrow$ BUDGET-SPLITS(visitTime)
  for $(vt1 \in splits)$ **do**
    $vt2 \leftarrow visitTime - vt1$
    $\mathcal{I}_1 \leftarrow$ RECURSIVE-EMIP($\mathbf{C}_s$, c, $vt1$, $nextTravelTime$, $\mathbf{X}$, $iter - 1$)
    $\mathcal{I}_2 \leftarrow$ RECURSIVE-EMIP(c, $\mathbf{C}_d$, $vt2$, $nextTravelTime$, $\mathbf{X} \cup \mathcal{I}_1$, $iter - 1$)
    if $(\text{score}(\mathcal{I}_1 \oplus \mathcal{I}_2) > \text{score}(\mathcal{I}))$ **then**
      $\mathcal{I} \leftarrow \mathcal{I}_1 \oplus \mathcal{I}_2$
    end if
  end for
end for
return $\mathcal{I}$

Chapter 5

Experimental Evaluation

In the following sections we are empirically evaluating the two discussed algorithms and compare them to each other. Specifically, we are interested in two main aspects of the algorithms, the score of the itineraries generated by the algorithms and their runtime.

Our input consists mainly of three different parameters: the start and destination point, the time budget t_{max} and the categories with their respective max visits. In order to get informative and representative benchmarks we will test the algorithms by varying only one parameter at a time, while keeping the others fixed.

5.1 Benchmarked Algorithms

In our benchmarks we are testing the two algorithms explained in chapter 3 and 4. For comparison we will evaluate also a third, optimal algorithm, which will be explained shortly.

As already mentioned previously the accuracy of ARKTIS depends heavily on the choice of a utility function for the root construction, there is, however, no single utility that is always better than the others. Fortunately our algorithm is efficient enough to run it multiple times with different utility functions. The resulting hybrid algorithm uses the different utilities, partially explained in section 3.5. Despite the multiple execution, our algorithm is still competitive in terms of runtime as our benchmarks suggest.

The eMIP implementation described in chapter 4 lacks some features that Singh et al. mentioned in their paper. The missing features do not change the outcome of the computation, because we just skipped some performance improvements. For instance, we did not implement the branch and bounding that they use to improve the runtime, as it was rather complex to implement and the runtime was not that much of a problem after all. Remember that the eMIP algorithm does not guarantee to stay within the provided time budget. Therefore it is sometimes hard to really compare the different algorithms to each other.

In order to see how well the algorithms approximate the optimal solution for each problem instance we implemented also a third algorithm, called the optimal algorithm and it computes, unsurprisingly, the optimal solution. However, due to the NP-hardness

of the problem the optimal algorithm is not scalable and therefore we are not able to compute the best solution for any problem instance within a reasonable time. In fact, in all our benchmarks the maximum allowed runtime of the optimal algorithm was set to three hours, if that limit was exceeded the algorithm would be aborted and the benchmarks would proceed with the other algorithms. The optimal algorithm uses dynamic programming and certain pruning techniques to improve the runtime and thus we are still able to compute the optimum for small to medium sized problem instances.

5.2 Benchmark Environment

The hardware running the benchmarks is a server machine with 2 Intel Xeon X5550 CPUs, each of them having 4 cores and multi threading enabled, thus resulting in overall 16 virtual cores. The CPU frequency is 2.67GHz. The server has 48GB of main memory from which 45GB are used for the benchmarking process. Please note that, although the hardware has multiple cores available, the algorithms are not multi-threaded and thus cannot fully profit from the existence of multiple cores. The server is running Ubuntu 12.04 server edition and the algorithms were implemented in Java, compiled with javac version 1.7.0_17. The optimal algorithm to which both algorithms are compared to, was implemented in C++ and was compiled with gcc version 4.8.1.

The benchmarks are conducted on several input graphs, including artificially created street networks and real-world cities, like the city of Bolzano (1830 POIs) in northern Italy. The synthetic networks are especially important because we could manually tweak every single input parameter, as for instance the size of the network, the number of POIs and categories, the distribution of the POIs over the map, etc.

Figure 5.1 shows such a synthetic network created for benchmarking only, it consists of 1521 nodes of which 400 are POIs (the blue colored dots). The POIs are split into 8 different categories with 50 POIs each. The POIs are distributed across the network using a pareto distribution, which naturally clusters POIs and thus resembles the network of a city well. The color of the edges indicates the distance between the nodes, where the color range spans from green (very close) to red (far away). Besides the color, also the thickness of an edge indicates its length, so in figure 5.1 there are some edges which are particularly long. The graph diameter is about 90 minutes walking time from one corner to the opposite one.

5.3 Varying t_{max}

In the first benchmark we want to evaluate the impact of the time budget t_{max} on the runtime and score. We start with $t_{max} = 5400$ seconds, i.e. 1.5 hours and slowly but steadily increase it to 6 hours in steps of half an hour. The other parameters are fixed at 2 categories with 50 POIs and $max_k = 8$ each, resulting in itineraries of length at most $2 \times 8 = 16$. The benchmark is repeated with 20 different start and destination points and the mean for both, runtime and score is taken for each time slot.

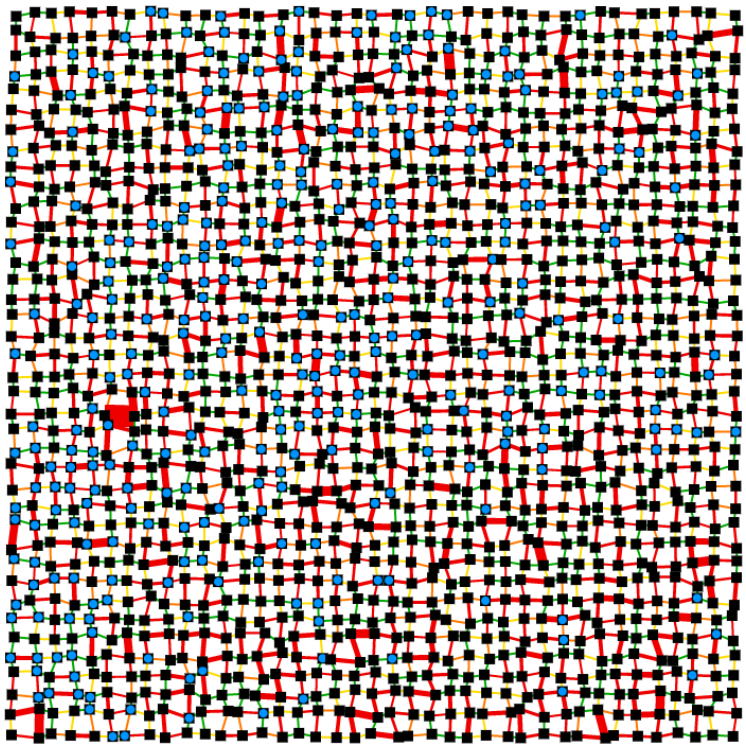

Figure 5.1: Artificial city network with pareto distributed POIs

Figure 5.2 shows how the score increases as we increase t_{max}. Unfortunately, due to the high number of 8 max visits per category the optimal solution could only be computed for t_{max} up to 2.5 hours. The eMIP algorithm performs generally better than ARKTIS in terms of the score of the generated solutions. Initially the curves are close together, but then the eMIP gains a noticeable lead. However, starting from t_{max} equals to 4.5 hours both curves flatten out and almost coincide, this is due to the fact that both algorithms have hit the limit of 16 POIs in total and therefore they cannot improve further.

Figure 5.3 shows the results of the same benchmark but on a different artificial network with uniformly distributed POIs. The shape of the curves looks mostly the same, though the hybrid ARKTIS algorithm beats the eMIP algorithm most of the times. Also the optimal algorithm can be computed for more problem instances, probably because in such a network the POIs are not so densely clustered and thus the tourist has to spent more time walking from POI to POI.

It is also worth to see how well the algorithms exhaust their time budget and figure 5.6 and 5.7 show exactly that for both types of networks. The purple line represents the time budget t_{max} and in order to be a valid solution to OPMPC the algorithms have to stay under that line. The eMIP algorithm, however, constantly uses more time than allowed, this also explains why it gains such a lead over the hybrid ARKTIS algorithm.

In figure 5.4 and 5.5 we compare the increase of t_{max} to the runtime of the algorithms again on both types of networks. Unsurprisingly the optimal algorithm quickly becomes very expensive and had to be aborted. The hybrid algorithm in turn has an almost constant runtime of 100 ms, this means that increasing t_{max} has almost no impact on the runtime of the hybrid algorithm. This is due to the fact that the work intensive parts like sorting the POIs for the knapsack approximation or computing the TSP path do not depend on t_{max}. In contrast, the runtime of the eMIP algorithm depends heavily on the time budget as the depth of the recursion is bound to it. For example in both figures (5.4 and 5.5) there is a drastic increase in runtime at about t_{max} equals to 4.5 hours and this is probably caused by the automatic increase of the maximum iteration depth *iter*. However, remember that as we already said the eMIP algorithm has not all performance improvement techniques integrated that Singh et al. described in [9] and therefore the algorithm could definitely be improved.

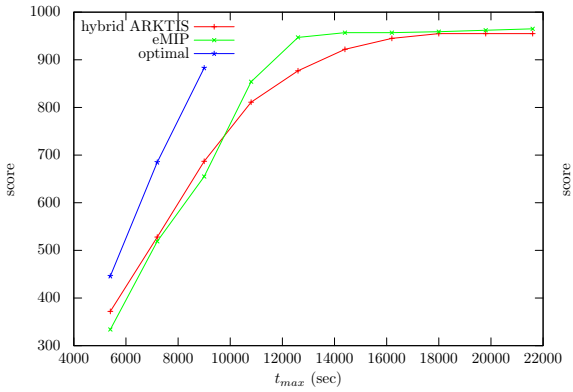

Figure 5.2: t_{max} to score (pareto)

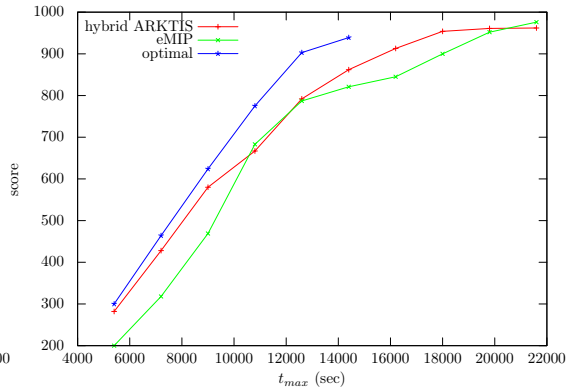

Figure 5.3: t_{max} to score (uniform)

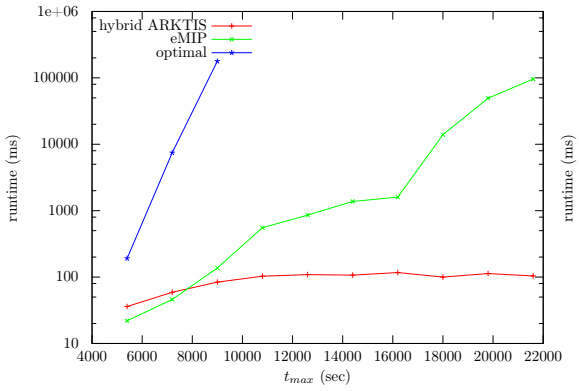

Figure 5.4: t_{max} to runtime (pareto)

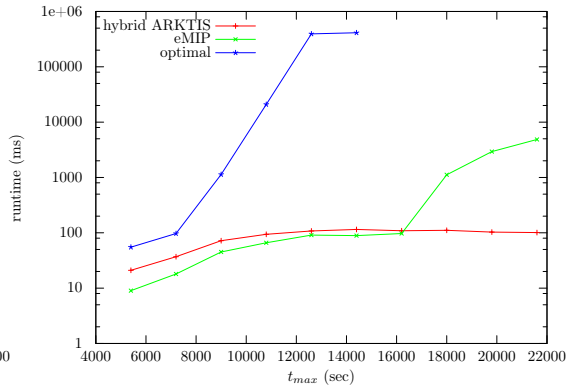

Figure 5.5: t_{max} to runtime (uniform)

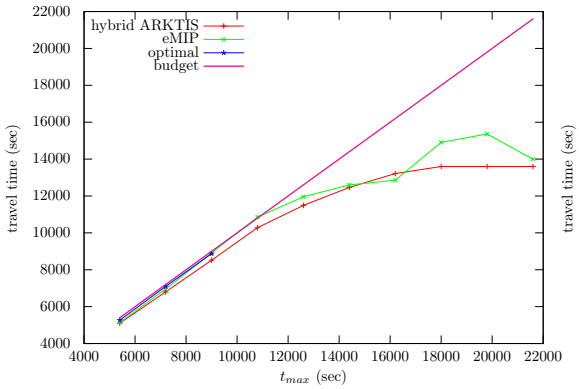

Figure 5.6: t_{max} exhaustion (pareto)

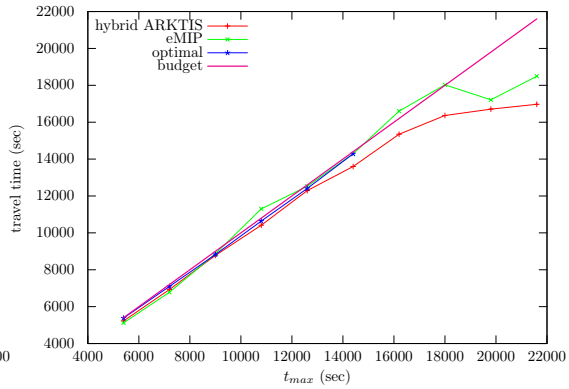

Figure 5.7: t_{max} exhaustion (uniform)

5.4 Varying Number of POIs, n

The next benchmark studies the runtime and score of the algorithms as the number of POIs, n, is increased. We do so by increasing the number of categories, starting from 1 category up to 8 categories in steps of 1. This corresponds to an increase in n of 50 POIs per step, as each category has 50 POIs. Additionally every category has max visits set to 2 and t_{max} is set to 4 hours. To sum up, the first measurement will have $k = 1$ category with max visits $max_k[1] = 2$ and $n = 50$ POIs, whereas the last measurement has $k = 8$ categories, with max visits 2 each and $n = 400$ POIs.

Figure 5.8 compares the number of categories to the runtime of the algorithms. Similar to the previous benchmark also here ARKTIS is the most efficient one, initially it has a runtime of ca. 10ms and increases to about 200ms towards to end. Again the optimal algorithm did not scale and had to be aborted already with $n = 100$ POIs. The eMIP algorithm takes already a whole second for the first measurement and increases steadily until it reaches about 1 minute in the last measurement.

In terms of score both algorithms perform pretty well in the very beginning when the optimal solution could be still computed (figure 5.9), in fact they even coincide. Starting from $k = 5$ categories the ARKTIS algorithm cannot keep pace with eMIP and loses gradually. The plots, however, do not show that the eMIP algorithm uses in some measurement again more time than allowed.

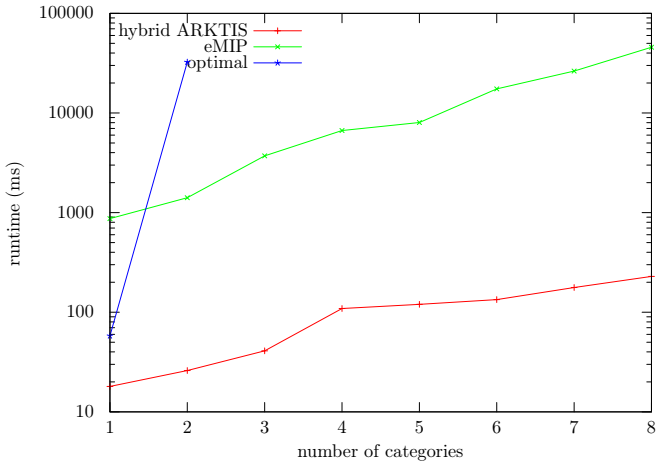

Figure 5.8: Number of categories to runtime

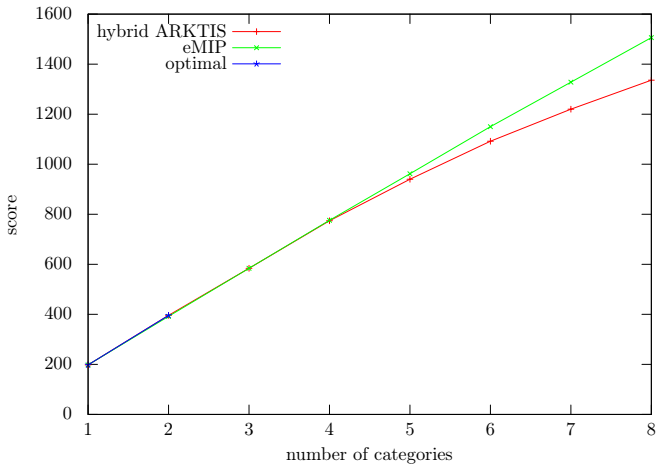

Figure 5.9: Number of categories to score

Chapter 6

Related Work

The Orienteering Problem is known under a couple of different names, as for instance the *Selective Traveling Salesman Problem* [10], the *Generalized Traveling Salesman Problem* or the *Bank Robber Problem* [11]. The name *Orienteering* comes originally from a game with the same name, where the participants have to reach as many checkpoints as possible in difficult terrain within a given time limit.

A recent survey by Vansteenwegen et al. [12] lists a number of applications for the Orienteering Problem besides its use for itinerary planning. It was first mentioned in 1984 by Tsiligirides who described a TSP variant where the salesman does not have enough time to visit all cities and has therefore to carefully choose among them in order to maximize the number of cities in the final trip. This is similar to our problem setting in the case where all POIs have the same score and we simply try to visit as many of them as possible. The OP found also application in all kinds of routing problems where for example the fuel is a natural constraint on the length of a path.

Since the Orienteering Problem is NP-hard the literature is mainly concerned with two approaches: on the one hand algorithms to find the optimal solution are of interest, which apply certain pruning techniques in order to cut the search as well as possible. Especially algorithms of the branch-and-bound family are very popular. On the other hand heuristic algorithms are designed that focus on efficiency and try to approximate the optimal solution as well as possible. The design of an efficient and accurate heuristic is, however, difficult because of the nature of the problem. Gendreau et al. argue in [10] that the independence of an element's profit and its cost makes it difficult to achieve the optimization goal, because to maximize the score while keeping the path length as short as possible is often contradictory. Moreover they point out that heuristic approaches frequently suffer from the same problem. If an element that looks initially very promising is selected, it can easily happen that it drags the solution far away in the wrong direction and therefore it becomes difficult to add other elements. If a POI, for example, is close to the city boundaries but still has a very high score it drags the whole itinerary away from the city center, even though there are a lot of maybe lower scoring POIs.

Just recently the concept of *submodularity* was exploited to design heuristics for the

Orienteering Problem. Chekuri et al. have published a series of algorithms that provide theoretic guarantees on the quality of the solution. We have already discussed in great detail the Recursive Greedy algorithm presented in [3] in chapter 4 and showed how Singh et al. improved it in [9]. Besides that, Chekuri et al. improved their algorithm independently and achieved an ϵ-approximation for the Orienteering Problem in [13]. Their algorithm with an approximation guarantee of $(2 + \epsilon)$ beats the before then best known approximation guarantee of 3 published in [14]. However, these algorithms are very theoretic and are unlikely to ever be implemented in the present form.

Both algorithms have in common that they exploit a $(2 + \epsilon)$ approximation for the minimum-excess problem presented in [15]. Given a weighted graph G together with start and end nodes s and d, Blum et al. defined the excess of a path to be the difference between a prize-collecting $s - d$ path and the length of the shortest path from s to d. The minimum excess problem is in turn, given a quota k find a minimum-excess path from s to d having a reward of at least k. The reward of a node corresponds to the score of a POI in our terminology. Blum et al. moreover showed that an approximation to the minimum-excess problem implies an approximation to the Orienteering problem and could therefore prove the APX-hardness of the problem. The main idea behind both algorithms is that a minimum-excess path of length at most t_{max} with a quota k yields a solution to the Orienteering problem. However, the quota k corresponds to the total score of the optimal solution, which is obviously not known in advance. Therefore their algorithm performs a "guess" on k and if not satisfactory the value is refined, which is essentially a binary search on k. We instead apply a binary search to find the proper value for t'_{max} since our cost approximation is not precise enough.

The approach to categorize the elements and let the tourists define the maximum number of POIs they want to visit per category is novel. It is a natural and easy, yet efficient additional constraint for the Orienteering Problem applied in the tourism sector, that helps tourists to filter among the possibly thousands of POIs only the interesting ones.

Chapter 7

Conclusion

Orienteering is a fascinating NP-hard problem, which is, however, not easy to approximate well. In this paper we have seen that the Orienteering Problem is at its core a TSP and a Knapsack problem and by merging them together we provided an algorithm that exploits the concept of submodularity. We tested the algorithm thoroughly on several different networks, some of them being real-world cities and others being just artificially created networks. Moreover we implemented a state of the art algorithm, called eMIP, and compared our algorithm to it. As our benchmarks suggest, our algorithm is efficient and approximates the optimal solution most of the times very well.

7.1 Future Work

There are a lot of areas where the current algorithm could be further improved and made more flexible. For instance, one of our simplifying assumptions were that the graph is undirected, but for real-world networks this is hardly the case as a street network is a complex system with many peculiarities, e.g. one way roads. Moreover, once we take different transportation means into account the situation becomes more difficult as the shortest path between two points depends also on the schedule of the public transportation means etc.

Clearly POIs have opening hours which we currently do not take into account, but would be very important once the algorithm is used in a real-world itinerary planning software. Moreover the score of a POI might be related to the time it is being visited, because a restaurant at 9 am is less interesting for most people than at noon.

One could think about the roots as being a "template" for an itinerary and the users could then choose the one template that suits them most. For example, one such template could have culturally interesting POIs in the morning, a restaurant around noon and some entertainment related POIs in the afternoon.

Clearly there are numerous improvement ideas, some of them are more interesting from an algorithmic point of view and others are more important in a real-world usage of the system. But this shows, that the research in such systems has a very practical application and can be useful for many people.

Bibliography

[1] C. G. Andreas Krause, "Near-optimal observation selection using submodular functions," in *In AAAI Nectar*, 2007.

[2] M. Sviridenko, "A note on maximizing a submodular set function subject to a knapsack constraint," *Oper. Res. Lett.*, vol. 32, no. 1, pp. 41–43, 2004.

[3] C. Chekuri and M. Pal, "A recursive greedy algorithm for walks in directed graphs," (Los Alamitos, CA, USA), pp. 245–253, IEEE Computer Society, 2005.

[4] G. Nemhauser, L. Wolsey, and M. Fisher, "An analysis of approximations for maximizing submodular set functions-i," *Mathematical Programming*, vol. 14, no. 1, pp. 265–294, 1978.

[5] A. Souza, "Combinatorial algorithms." Lecture Notes, Humboldt University Berlin, 2011.

[6] V. V. Vazirani, *Approximation algorithms*. New York, NY, USA: Springer-Verlag New York, Inc., 2001.

[7] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[8] J. A. Hoogeveen, "Analysis of Christofides' heuristic: Some paths are more difficult than cycles," *Operations Research Letters*, vol. 10, no. 5, pp. 291 – 295, 1991.

[9] A. Singh, A. Krause, C. Guestrin, W. Kaiser, and M. Batalin, "Efficient planning of informative paths for multiple robots," in *In IJCAI*, 2007.

[10] M. Gendreau, G. Laporte, and F. Semet, "A tabu search heuristic for the undirected selective travelling salesman problem," *European Journal of Operational Research*, vol. 106, no. 2–3, pp. 539 – 545, 1998.

[11] E. M. Arkin, J. S. B. Mitchell, and G. Narasimhan, "Resource-constrained geometric network optimization," in *Proceedings of the fourteenth annual symposium on Computational geometry*, SCG '98, (New York, NY, USA), pp. 307–316, ACM, 1998.

[12] P. Vansteenwegen, W. Souffriau, and D. V. Oudheusden, "The orienteering problem: A survey," *European Journal of Operational Research*, vol. 209, no. 1, pp. 1 – 10, 2011.

[13] C. Chekuri, N. Korula, and M. Pál, "Improved algorithms for orienteering and related problems," *ACM Trans. Algorithms*, vol. 8, pp. 23:1–23:27, July 2012.

[14] N. Bansal, A. Blum, S. Chawla, and A. Meyerson, "Approximation algorithms for deadline-tsp and vehicle routing with time-windows," in *Proc. of ACM STOC*, pp. 166–174, 2004.

[15] A. Blum, S. Chawla, D. R. Karger, T. Lane, A. Meyerson, and M. Minkoff, "Approximation algorithms for orienteering and discounted-reward tsp," *SIAM J. Comput.*, vol. 37, pp. 653–670, May 2007.