Efficient Itinerary Planning with Category Constraints

Paolo Bolzoni Sven Helmer
Kevin Wellenzohn Johann Gamper
Free University of Bolzano
Piazza Domenicani 3
Bolzano BZ, Italy
firstname.lastname@unibz.it

Periklis Andritsos
Université de Lausanne
Quartier UNIL-Dorigny
Bâtiment Internef
1015, Lausanne, Switzerland
periklis.andritsos@unil.ch

ABSTRACT

We propose a more realistic approach to trip planning for tourist applications by adding category information to points of interest (POIs). This makes it easier for tourists to formulate their preferences by stating constraints on categories rather than individual POIs. However, solving this problem is not just a matter of extending existing algorithms. In our approach we exploit the fact that POIs are usually not evenly distributed but tend to appear in clusters. We develop a group of efficient algorithms based on clustering with guaranteed theoretical bounds. We also evaluate our algorithms experimentally, using real-world data sets, showing that in practice the results are better than the theoretical guarantees and very close to the optimal solution.

Categories and Subject Descriptors

G.2 [**Discrete Mathematics**]: Graph Theory

General Terms

Algorithms, Experimentation, Performance

Keywords

Generalized Orienteering Problem, Route Planning

1. INTRODUCTION

When it comes to planning their trips, many tourists are overwhelmed by the choices they have. Large cities offer hundreds or even thousands of points of interest (POIs), but time and resources of a visitor are usually limited. Our goal is to provide a tool that generates itineraries staying within a time budget and selecting POIs along the route that are as attractive as possible. We have identified two important properties that help us in doing so. Usually POIs are not evenly distributed throughout a city or area, they tend to appear in clusters. In addition to this, most tourists are particularly interested in certain types of POIs and not in others. For example, a visitor may want to view cultural venues, such as museums, art galleries, and theaters, while someone else may mainly be interested in shopping. Categorizing points of interests makes it much easier for a user to provide input about the POIs he or she is interested in. However, even though a tourist may like to go and see museums, coming up with an itinerary containing five, six, or even more of them is not a good idea, as this will probably result in some form of sensory overload. Another example is someone giving high ratings to POIs that are restaurants not wanting an itinerary consisting entirely of restaurants. So a user should be able to restrict the number of POIs for each selected category.

SIGSPATIAL'14, November 04 - 07 2014, Dallas/Fort Worth, TX, USA
Copyright 2014 ACM 978-1-4503-3131-9/14/11 ...$15.00
http://dx.doi.org/10.1145/2666310.2666411.

A recent survey on algorithmic approaches for tourist itinerary planning has identified crucial topics that have been mostly neglected so far [4]. Among them is formulating more realistic scenarios for tourist trip planning by integrating more complex requirements and constraints into the search. An important open problem explicitly stated in [4] is a max-n type constraint, i.e., limiting the number of visits of POIs of certain categories. This is exactly the issue we are tackling in our work. As we show in Section 2 on related work, introducing max-n type constraints into existing trip planning algorithms is far from trivial, meaning we had to come up with a new approach for generating tourist itineraries efficiently.

In summary, we make the following contributions:

- We formally define the problem of itinerary planning with category constraints, which facilitates the formulation of user preferences.
- We show that the problem of finding an optimal route with max-n type constraints is NP-hard and we develop three variants of an algorithm (called CLIP) based on clustering.
- Two of the three variants provide guarantees on time budget approximation and run time, the third only on run time.
- In an experimental evaluation we demonstrate that CLIP's itineraries are very close to the optimal solution (much better than the theoretical guarantees), while still being efficient.

The remainder of the paper is organized as follows. In the next section we cover related work and in Section 3 we formalize the problem. Section 4 contains a detailed description of our algorithm, while in Section 5 theoretical bounds are provided. This is followed by an experimental evaluation, comparing our algorithm to a simple greedy heuristic and the optimal one. Finally, Section 7 concludes the paper.

2. RELATED WORK

Solving a tourist trip planning problem algorithmically boils down to finding a solution to an instance of the orienteering problem (OP). Introduced by Tsiligrides in [17], OP is about determining a path from a starting node to an ending node in an edge-weighted graph with a score for each node, maximizing the total score while staying within a certain time budget. Orienteering is an NP-hard problem and algorithms computing exact solutions using branch and bound [5, 11] as well as dynamic programming techniques [10, 13] are of limited use, as they can only solve small problem instances. Consequently, there is a body of work on approximation algorithms and heuristics, most of them employing a two-step approach of partial path construction [7, 17] and (partial) path improvement [1, 2, 14]. Meta-heuristics, such as genetic algorithms [16], neural networks [19], and ant colony optimization [9] have also been tested. For a recent overview on orienteering algorithms, see [4]. However, none of the approaches investigate OP generalized with categories.

There is also work on planning and optimizing errands, e.g., someone wants to drop by an ATM, a gas station, and a pharmacy on the way home. The generalized traveling salesman version minimizes the time spent on this trip [12], while the generalized orienteering version maximizes the number of visited POIs given a fixed time budget. However, in these variants the POIs have no scores, so no trade-offs between scores and distances have to be considered.

A natural starting point for developing a max-n type constraint approximation algorithm for orienteering would be to adapt an existing algorithm for traditional OP. However, many of the existing algorithms have a high-order polynomial complexity or no implementation exists, due to their very complicated structure. Two of the most promising approaches we found were the segment-partition-based technique by Blum et al. [1] and the method by Chekuri and Pál, exploiting properties of submodular functions [3]. The latter approach, a quasi-polynomial algorithm, is still too slow for practical purposes. Nevertheless, Singh et al. modified the algorithm by introducing spatial decomposition for Euclidean spaces in the form of a grid, making it more efficient [15]. Common to all of the approaches, though, is breaking down the itinerary recursively into smaller and smaller segments, which get assembled into a complete tour. If we just run these algorithms without any alterations on POIs with categories, it is very likely that the solution violates the max-n type constraints. A fix would be to try out all possible distributions of max-n type constraints for every recursive call. For example, given a max-n type constraint of 3 for a category and assuming `recl` computes the left half of a route and `recr` the right one, we would have to make the following calls with max-n type constraints: `recl`(0), `recr`(3); `recl`(1), `recr`(2); `recl`(2), `recr`(1); `recl`(3), `recr`(0). While this would guarantee an answer respecting the max-n type constraints, it would also blow up the computational costs.

3. PROBLEM FORMALIZATION

In our problem, we assume a set of n POIs, represented by $\mathbf{P}$, with members $p_i, 1 \leq i \leq n$. The POIs, together with a starting and a destination node, denoted by s and d, respectively, are connected by a complete, metric, weighted, undirected graph $G = (\mathbf{P} \cup \{s, d\}, \mathbf{E})$, whose nodes are the set of POIs $\mathbf{P}$ as well as nodes s and d, and whose edges, $e_l \in \mathbf{E} = \{(x, y) \mid x, y \in \mathbf{P} \cup \{s, d\}\}$ are the connections between them. Each edge e_l has a cost $c(p_i, p_j)$ that signifies the duration of the trip from p_i to p_j, while every node $p_i \in \mathbf{P}$ has a cost v_{p_i} that denotes the visiting time of that POI. Each POI belongs to a certain category, such as *museums, restaurants*, or *galleries*. The set of m categories is denoted by $\mathbf{K}$ and each POI p_i belongs to exactly one category $k_j, 1 \leq j \leq m$. Given a p_i, $\texttt{cat}(p_i)$ denotes the category p_i belongs to and $\texttt{score}(p_i)$ its score, with higher values indicating higher interest to the user. Finally, users have a certain maximum time in their budget to complete the itinerary, denoted by $t_{\max}$.

We now define the notion of an itinerary.

Definition 1. (Itinerary) *An itinerary $\mathcal{I}$ starts from a starting point s and finishes at a destination point d (note that s and d can be identical). It includes an ordered sequence of connected POIs $\mathcal{I} = \langle s, p_{i_1}, p_{i_2}, \ldots, p_{i_q}, d\rangle$, each of which is visited once. We define the* cost *of itinerary $\mathcal{I}$ to be the total duration of the path from s to d passing through and visiting the POIs in $\mathcal{I}$, $\mathtt{cost}(\mathcal{I}) = c(s, p_{i_1}) + v_{p_{i_1}} + \sum_{j=2}^{q}(c(p_{i_{j-1}}, p_{i_j}) + v_{p_{i_j}}) + c(p_{i_q}, d)$, and its* score *to be the sum of the scores of the individual POIs visited, $\mathtt{score}(\mathcal{I}) = \sum_{j=1}^{q} \mathtt{score}(p_{i_j})$.*

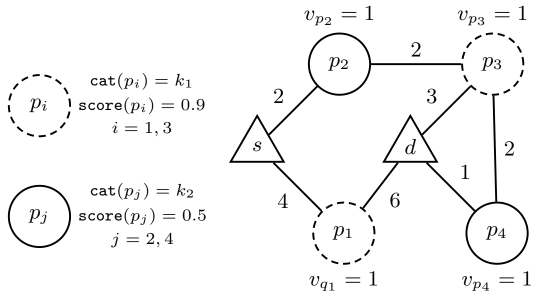


Figure 1: Itinerary including $n = 4$ POIs

Example 1. *Figure 1 shows nodes s and d, four POIs, p_1, p_2, p_3, and p_4 as well as the duration of traveling between places, the visiting time of each POI and the scores of the categories to which each POI belongs. We simplify the graph slightly to keep it readable: all POIs of a category share the same score and we omit some edges. One example itinerary between s and d is the one that includes only p_1,* i.e., *$\mathcal{I}_1 = \langle s, p_1, d\rangle$, while a second one includes p_2 and p_3,* i.e., *$\mathcal{I}_2 = \langle s, p_2, p_3, d\rangle$, a third itinerary can be $\mathcal{I}_3 = \langle s, p_2, p_3, p_4, d\rangle$. Given the values for edge costs, visiting times, and scores of nodes, we can compute the costs and scores of the itineraries:*

- *$\mathcal{I}_1 = \langle s, p_1, d\rangle$:*
  $\mathtt{cost}(\mathcal{I}_1) = 4 + 1 + 6 = 11$, $\mathtt{score}(\mathcal{I}_1) = 0.9$;
- *$\mathcal{I}_2 = \langle s, p_2, p_3, d\rangle$:*
  $\mathtt{cost}(\mathcal{I}_2) = 2 + 1 + 2 + 1 + 3 = 9$,
  $\mathtt{score}(\mathcal{I}_2) = 0.5 + 0.9 = 1.4$;

- $\mathcal{I}_3 = \langle s, p_2, p_3, p_4, d \rangle$:
  $\mathtt{cost}(\mathcal{I}_3) = 2 + 1 + 2 + 1 + 2 + 1 + 1 = 10$,
  $\mathtt{score}(\mathcal{I}_3) = 0.5 + 0.9 + 0.5 = 1.9$.

Example 1 shows a few of the possible itineraries that can be constructed between points s and d using four POIs. In general, given a subset P of $\mathbf{P}$, our goal is to place them in an itinerary from s to d as defined in Definition 1. Given the traveling and visiting times as well as the scores, we need to build an itinerary with duration smaller than $t_{\max}$ and maximum cumulative score.

As already mentioned, we introduce an additional constraint specifying the number of POIs per category that can be included in the final itinerary. More precisely, we introduce a parameter $\max_{k_j}$ for each category k_j that is set by the user to the maximum number of POIs in a category that he or she prefers to visit during the trip. We are now ready to define the *Orienteering Problem with Maximum Point Categories, (OPMPC).*

Definition 2. (OPMPC) *Given a starting point s, a destination point d, n points of interest $p_i \in \mathbf{P}$, with scores described by the function $\mathtt{score}(p_i)$, visiting times $v_{p_i}, 1 \leq i \leq n$, traveling times $c(x, y)$ for $x, y \in \mathbf{P} \cup \{s, d\}$, categories $k_j \in \mathbf{K}, 1 \leq j \leq m$, 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 the category k_j $(1 \leq j \leq m)$, 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, $\mathtt{score}(\mathcal{I})$, is maximized;*
- *no more than $\max_{k_j}$ POIs are used for category k_j;*
- *the time constraint is met,* i.e., $\mathtt{cost}(\mathcal{I}) \leq t_{max}$.

Example 2. *In the presence of categories k_1 with $\max_{k_1} = 1$ and k_2 with $\max_{k_2} = 1$, and assuming that $t_{max} = 10$, we can observe the following about the itineraries in Example 1: Itinerary $\mathcal{I}_1$ is infeasible since its cost is greater than t_{max}, while the other two fulfill the time requirement. Comparing $\mathcal{I}_2$ and $\mathcal{I}_3$, we can see that $\mathcal{I}_3$ is of higher benefit to the user, even though it takes more time to travel between s and d. However, it cannot be chosen since it contains two POIs from k_2. Itinerary $\mathcal{I}_2$ contains two POIs, each from a different category and it could be one recommended to the user.*

Theorem 3.1. *The Orienteering Problem with Maximum Point Categories (OPMPC) is NP-hard.*

Proof. We can reduce the original (NP-hard) orienteering problem to OPMPC by placing each point of interest into its own category k_i, $1 \leq i \leq n$, and setting each $\max_{k_i}$ to 1. 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. $\square$

4. CLIP ALGORITHM

Figure 2 gives a graphical, high-level overview of the components of our algorithm, which is called CLIP (CLuster Itinerary Planning). It is split into four major parts, represented by the shaded boxes; the white boxes illustrate the data that flows through the different parts.

In a first step we reduce the set of all POIs, $\mathbf{P}$, to a subset of points that belong to the categories a user asks for. This is fed into a clustering algorithm that groups the nodes into clusters. This set of clusters in turn serves as input for the cluster route generator, a component that systematically generates paths from the starting point s to the destination d visiting clusters on the way. As we will see, by introducing pruning we do not have to generate all possible cluster paths. Once we have a cluster path, we can approximate the time spent on traveling from s to d via the selected clusters, which means we also have a good idea on how much time we can spend visiting POIs on this route. For selecting POIs from the visited clusters in a cluster path we utilize knapsack algorithms. So far we have only dealt with paths on the level of clusters. Finally, we turn these into itineraries that traverse the individual nodes selected in the previous step.

During the early stages of developing an algorithm for OPMPC, we found it extremely difficult to optimize POI selection and route generation at the same time. Nodes with a high score may be far away and low-scoring ones close by, so we cannot just focus on absolute scores. However, utility functions considering score and distance can also be misleading, as a greedy strategy is easily led astray by one good, isolated POI, while it may be much more worthwhile to go into the direction of a whole cluster of slightly less valuable POIs. We solved this issue by separating the creation of routes from the selection of POIs. In terms of performance, we managed to reduce the size of the data input considerably by removing irrelevant POIs belonging to unwanted categories and by dealing with whole clusters instead of individual POIs. In the following we provide more details on the different parts of our algorithm.

4.1 Clustering

We apply *agglomerative clustering.* Initially each POI is placed in a separate cluster, and we then proceed by continuously merging (*agglomerating*) the two nearest clusters into one until the desired number of clusters k is reached. We discuss the choice of k in our experiments.

4.1.1 Definition

The distance between two POIs is determined by the underlying road network of the map and can be determined by running an all-pairs shortest path algorithm, resulting in a completely connected graph. We denote the distance between the POIs p_i and p_j by the cost of the edge connecting them: $c(p_i, p_j)$. In order to measure the distance between two clusters, we use medoid distance: for two clusters $\mathbf{c}_1$ and $\mathbf{c}_2$, where $\mathbf{c} = \mathbf{c}_1 \cup \mathbf{c}_2$, their distance is defined as $c(\odot_1, \odot_2)$ where $\odot_1$ and $\odot_2$ are the medoids of $\mathbf{c}_1$ and $\mathbf{c}_2$, respectively.

The intracluster distance *icd* of a POI p is the sum of distances to all other POIs in the cluster $\mathbf{c}$: $icd(p) = \sum_{q \in \mathbf{c}} c(p, q)$. The medoid of a cluster is the POI with the minimal *icd*. In case of a tie, we take the POI with the smaller index. Basically, a medoid acts as a representative or exemplar of its cluster. In our case it is very important to have representatives that minimize the distance to all other POIs in the cluster, as this helps us in approximating travel time more closely. Popular clustering techniques, such as group-average agglomerative clustering (GAAC), did not achieve this: we noticed that the variance of distances between a medoid and the POIs in its cluster was much greater than the variance found in our approach.

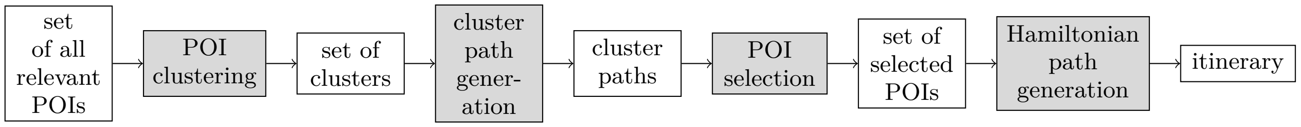


Figure 2: Overview of CLIP algorithm (white boxes represent data, shaded boxes algorithms)

4.1.2 Precomputation

Figure 2 indicates that before clustering we first remove irrelevant POIs, i.e., points belonging to unwanted categories or not reachable within the time constraint. However, reclustering the POIs every time we create an itinerary is fairly expensive (costing $O(n^3)$). Since maps rarely change, we want to speed up the processing by precomputing the clustering in form of a dendrogram and materializing it to disk. In order to make this work, we have to deal with two issues. We have to be able to create a k-clustering for any k on the fly and to eliminate the irrelevant POIs (as the precomputed clustering contains all the POIs).

With the help of a priority queue, which is initialized with the root node of the precomputed dendrogram, we now create a k-clustering. Until we have reached k entries in the queue we repeat the following steps. Take the first node from the queue and split it into its two subclusters, re-inserting the subclusters that contain at least one relevant POI into the queue. We always split the largest cluster since the nodes in the queue are sorted by the number of relevant POIs in their cluster. If a single-POI cluster has made it to the front of the queue we stop the process and basically run our algorithm on the level of individual POIs, since k was larger than the number of relevant POIs. In a final step we filter out all the non-relevant POIs.

4.2 Route Generation

We now take a closer look at the component generating routes. First, we define the concept of a cluster path before showing how we can approximate travel costs given a cluster path. Second, we illustrate how to generate cluster paths in a systematic way to explore the search space. Finally, we use pruning to reduce the number of generated paths.

4.2.1 Cluster Paths

A cluster path is very similar to an itinerary (see Definition 1), we replace the sequence of POIs with a sequence of clusters.

Definition 3. (Cluster path) *A cluster path* $\mathcal{C} = \langle s, \mathbf{c}_{i_1}, \mathbf{c}_{i_2}, \ldots, \mathbf{c}_{i_q}, d\rangle$ *starts from a starting point* s *and finishes at a destination point* d *and includes an ordered sequence of clusters.*

Figure 3 depicts an itinerary $\langle s, p_1, p_2, p_3, d\rangle$ and its cluster path $\langle s, \mathbf{c}_1, \mathbf{c}_2, d\rangle$. Mapping an itinerary into a cluster path is straightforward, we replace every POI between s and d with the cluster it belongs to. In contrast to the sequence of POIs in an itinerary, the clusters in a cluster path do not need to be unique, i.e., we can visit a cluster more than once. However, in order to get rid of some of the duplicates in a second step we reduce all contiguous sequences of identical clusters to single clusters.

4.2.2 Travel Costs

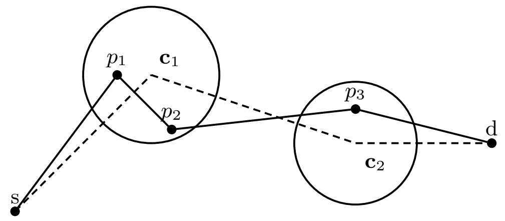


Figure 3: Approximation for the travel cost

We use a cluster path to approximate the time it takes to traverse an itinerary, not including any visiting times.

Definition 4. (Cluster path travel cost) *Given a cluster path* $\mathcal{C} = \langle s, \mathbf{c}_{j_1}, \mathbf{c}_{j_2}, \ldots, \mathbf{c}_{j_q}, d\rangle$*, its cost is computed as follows. We sum up the cost of the edges between* s*, the medoids of the clusters, and* d*:* $\mathtt{cost}(\mathcal{C}) = c(s, \odot_{j_1}) + \sum_{i=1}^{q-1} c(\odot_{j_i}, \odot_{j_{i+1}}) + c(\odot_{j_q}, d)$.

The sum of the lengths of the dashed lines in Figure 3 is the travel cost estimation. We assume it does not cost us anything to move within a cluster. Consequently, the smaller the clusters, the better the estimation.

4.2.3 Generating Cluster Paths

Algorithm 1 gives an overview of the main structure of our algorithm. When reading it for the first time, we suggest skipping the shaded parts, which do some pruning to make the algorithm run faster. If we removed all the pruning, the while loop (line 9) would systematically generate all possible cluster paths, eventually finding an optimal cluster path.

First, we initialize the queue keeping track of the generated cluster paths with the smallest possible path going from the starting node s directly to the destination d. As some of the paths can go over budget (cluster paths provide an approximation of the travel cost), we store potential solutions in a priority queue to determine their feasibility later on. This priority queue is seeded with a solution generated by a greedy algorithm to make sure that there is always at least one feasible solution in the queue. Then, while not empty, we take a path from the cluster path queue (line 10) and hand it to the KnapsackSolver, which selects a set of POIs from the cluster path trying to maximize the score of the path (this function is discussed in more detail in Section 4.3). We memorize this POI set as a potential solution in the priority queue (lines 20–21). Next we expand the current path with clusters not yet visited and append it to the cluster path queue (lines 24–27), i.e., inserting it between the current last cluster and d.

Once we exit the while loop, we convert potential solutions into true itineraries that traverse individual POIs rather than clusters. This is done by the function HamiltonianPathGen that takes s, d, and a set of POIs and builds a Hamiltonian path starting at s and ending at d visiting all the points in the POI set (we discuss this function in

Algorithm 1: CLIP

```
Input: poi_graph, s, d, categories_limit, budget, $\varphi$, ClusteringAlgo, KnapsackSolver, HamiltonianPathGen
Output: Approximation of $\mathcal{I}_{opt}$
1  $clusters$ = ClusteringAlgo($poi\_graph, budget, s, d, categories\_limit$)
2  $cpathqueue \leftarrow$ FifoQueue()
3  $cpathqueue$.enqueue(Path($\langle s, d\rangle, peak\_score \leftarrow 0$))
4  $same\_prefix \leftarrow$ AssociativeArray($key \leftarrow$ Pair(cluster set, cluster), $value \leftarrow cost$)
5  $l_{\max} \leftarrow$ sum of all category limits
6  $potential\_solutions \leftarrow$ PriorityQueue()
7  $greedy\_solution \leftarrow$ greedy($poi\_graph, budget, s, d, categories\_limit$)
8  $potential\_solutions$.push(score($greedy\_solution$), $greedy\_solution$)
9  while not $cpathqueue$.empty() do
10     $cpath \leftarrow cpathqueue$.dequeue()
11     if $budget < cpath$.cost() then
12         continue
13     if $cpath$.no_of_clusters() $> l_{max}$ then
14         continue
15     $sp\_cost \leftarrow same\_prefix$.get(Pair(Set($cpath$.clusters()), $cpath$.last_cluster()))
16     if $sp\_cost \neq$ None and $sp\_cost \leq cpath$.cost() then
17         continue
18     else
19         $same\_prefix$.put(Pair(Set($cpath$.clusters()), $cpath$.last_cluster()), $cpath$.cost())
20     $cpath.pois\_set \leftarrow$ KnapsackSolver($poi\_graph, cpath, budget -$ cost($cpath$), $categories\_limit$)
21     $potential\_solutions$.push(score($cpath.pois\_set$), $cpath.pois\_set$)
22     if score($cpath.pois\_set$) $< (1 - \varphi) \cdot cpath.peak\_score$ then
23         continue
24     foreach $c$ in $clusters \setminus cpath$.clusters() do
25         $new\_cpath \leftarrow$ append_cluster($cpath, c$)
26         $new\_cpath.peak\_score \leftarrow$ max(score($cpath.pois\_set$), $cpath.peak\_score$)
27         $cpathqueue$.enqueue($new\_cpath$)
28 loop
29     $solution \leftarrow potential\_solutions$.pop()
30     $itinerary \leftarrow$ HamiltonianPathGen($poi\_graph, s, d, solution$)
31     if cost($itinerary$) $\leq budget$ then
32         return $itinerary$
33     else
34         $itinerary \leftarrow$ shortcut($itinerary$)
35         $potential\_solutions$.push(score($itinerary$), $itinerary$)
```

Section 4.4). If the produced itinerary stays within budget, we return it as our solution. As *potential_solutions* is sorted by score in descending order, the returned itinerary is the best feasible solution found by our algorithm. If the itinerary goes over budget, we remove a POI from it and return it to the queue (more details on shortening paths in Section 4.4 as well).

If the potential solution priority queue becomes too large and we need to free memory, we execute the Hamiltonian path generation earlier, dropping all the infeasible itineraries and keeping the highest-scoring feasible one.

Example 3. *Assume that potential_solutions contains the following entries:* $[(80, o_3), (70, o_2), (\mathbf{65}, \boldsymbol{\mathcal{I}_1})]$, *where the bold pair is an itinerary and its score, while the other pairs consist of a score and a set* o_i *of POIs. Currently we only know that* $(65, \mathcal{I}_1)$ *is feasible, e.g. it is the solution generated during initialization (see line 8 in the algorithm) and we are not sure about the other pairs.*

If the knapsack solver finds a set of POIs with score lower than 65, this solution can be discarded immediately. Now imagine that the algorithm adds $(88, o_4)$ *to the queue, but we are running out of space and have to free memory. We apply* HamiltonianPathGen *to the elements in the queue, starting with* $(88, o_4)$. *If* $(88, \mathcal{I}_4)$ *goes over budget, we shorten and re-insert it (assume that the shorter itinerary has a score below 65 and is discarded). Next up is* $(80, o_3)$, *which we determine to be feasible with itinerary* $\mathcal{I}_3$. *Then we discard the rest of the queue and continue with* $[(\mathbf{80}, \boldsymbol{\mathcal{I}_3})]$.

4.2.4 Pruning

Compared to an algorithm generating paths containing individual POIs, switching to cluster paths brings down the run time considerably. However, enumerating all possible cluster paths is still too slow, so we prune aggressively. A rather straightforward elimination of paths is done in lines 11–12: if the travel time between clusters in a path exceeds the total budget, we discard it.

In lines 13–14, we eliminate a path when it includes more than $l_{\max}$ clusters, which is the sum of all category constraints, i.e., $l_{\max} = \sum_{i=1}^{m} \max_{k_i}$. Adding the $l_{\max}$ + 1st cluster will result in one cluster not contributing any POIs to the solution. In this case, we would have been better off skipping this cluster altogether, reducing the travel time and gaining more time for visiting (better) POIs.

A third kind of pruning starts in line 15: two paths visiting the exact same clusters provide the knapsack algorithm with the same set of POIs to choose from. We cannot just keep the one with the smaller travel cost, though, as it may leave us stranded in an inconvenient place for further expansions, leading to higher follow-up costs. Nevertheless, if both paths visit the exact same POI cluster before reaching the destination, then we can discard the one with the higher cost. We say that the path with the shorter travel time *dominates* the other path. By replacing the dominated path with the dominating one in further expansions, we can only make this solution better.

We now explain the final pruning step shown in line 22–23 of the algorithm. In general expanding paths with clusters will increase the part of the budget used for traveling and decrease the part used for visiting POIs. At some point this will lead to a situation in which an expanded path has a lower score than the original path. This is due to the reduced visiting budget forcing us to remove POIs and the newly added cluster not being able to make up for this loss. One could argue to prune paths as soon as they start losing score, but we have to be careful when doing so. On the way to the final destination we may still be able to expand such a path with some clusters containing very high-scoring POIs that are able to make up for the loss and produce a better solution. Skipping one of the clusters included in the current path before reaching the high-scoring ones may also not be an option, if all of them contribute to the score. The following example illustrates this.

EXAMPLE 4. *Figure 4 illustrates why a naïve pruning strategy may fail to work. As usual s and d are the start and destination, respectively. Clusters c_1 and c_2 are composed of the same (average) kind of POIs, while cluster c_3 includes a high-scoring POI, in particular:*

cluster	*POI name*	*visiting time*	*score*
c_1	p_{11}	*1*	*1*
c_1	p_{12}	*5*	*3*
c_2	p_{21}	*1*	*1*
c_2	p_{22}	*5*	*3*
c_3	p_{31}	*2*	*6*

Assuming $t_{max} = 9$, CLIP first inserts cluster c_1 into the empty path, resulting in $\langle s, c_1, d\rangle$ with a score of 3 using all of the available budget: the travel time is 4 which leaves 5 for visiting POIs, in our case p_{12} with a score of 3 (we could have visited p_{11} instead, but the score would have been lower). When this path is expanded to $\langle s, c_1, c_2, d\rangle$, this results in a score of 2 visiting p_{11} and p_{21}, using up 7/9 of the budget.[1] *So the score drops, but pruning at this point means we lose a solution with a score of 8: $\langle s, c_1, c_2, c_3, d\rangle$.*

Nevertheless, we can prune some of the paths with decreasing scores, guaranteeing, in the worst case, a loss of only a fraction of the best score achievable by that path. Formally speaking, we want to make sure that a solution computed by our algorithm provides at least a factor of φ, $0 < \varphi \leq 1$, of the best score of the cluster path. In order to achieve this, we drop a path only if it has lost score and the current score is below a factor of $(1 - \varphi)$ of the best score found so far, which we call the peak score of this cluster path (see line 26). That means, even if the path were able to catch up, **at least** $\varphi \cdot peak\_score$ has to come from future high-scoring clusters, while **at most** $(1 - \varphi) \cdot peak\_score$ comes

[1]The same holds for $\langle s, c_2, d\rangle$ and $\langle s, c_2, c_1, d\rangle$.

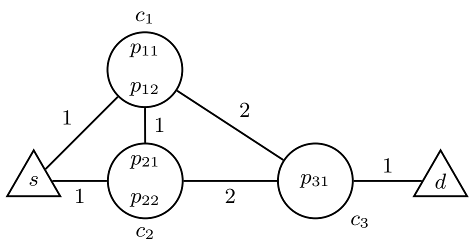


Figure 4: Naïve pruning missing a better solution

from the clusters in the path we are dropping. Moreover, the fraction of $(1 - \varphi) \cdot peak\_score$ cannot go up in future expansions, because the expansions are further reducing the visiting budget. If it turns out that the pruned path would have produced an optimal solution, the systematic generation of cluster paths keeps a path containing the high-scoring clusters, guaranteeing at least φ of that score.

4.3 POI Selection

Given a cluster path created by the path generator we have to select a subset of POIs from the clusters contained in that path so as to maximize the score of the path (see line 20). In essence we have to solve a multidimensional knapsack problem in which each of the m category constraints and the visiting time constraint make up the $d = m + 1$ different dimensions.

EXAMPLE 5. *A user has selected three categories with a maximum constraint of two each and after subtracting the travel costs from the total budget, there are 240 minutes left to visit POIs. So the capacity of the four-dimensional knapsack is $(2, 2, 2, 240)$. Let us assume that the Eiffel Tower is one of the potential POIs: if it belonged to the second category and required two hours to visit, it would be described by the cost vector $(0, 1, 0, 120)$.*

The multidimensional knapsack component of our problem can be defined as an integer linear programming (ILP) problem as follow:

$$\text{maximize} \sum_{i=1}^{n} s_i p_i \tag{1}$$

$$\text{subject to} \sum_{i=1}^{n} v_i p_i \leq b \text{ and } \sum_{i=1}^{n} \mathbb{1}_{ij} p_i \leq c_j, \quad 1 \leq j \leq m$$

where p_i is 0 or 1 and it determines whether POI p_i is part of the solution, s_i stands for its score, v_i for its visiting time. b is the time budget and c_j is the category k_j limit $\max_{k_j}$. Finally $\mathbb{1}_{ij} = 1$ if POI p_i belongs to category k_j and $\mathbb{1}_{ij} = 0$ otherwise.

Depending on the size of the problem instance and the time available for computing a solution, we use the algorithms described in Sections 4.3.1, 4.3.2, and 4.3.3 to solve this multidimensional knapsack problem. This gives rise to the three different variants of CLIP.

4.3.1 Exact Solution

In our implementation we employ the GNU Linear Programming Kit (GLPK) [6], which uses a branch-and-bound technique to solve integer linear programming problems.

In the general case, solving a multidimensional knapsack problem is NP-hard, as is solving an ILP. Furthermore, ILP problems encoding multidimensional knapsack problems are especially hard to solve: according to [8] even finding feasible solutions, let alone optimal solutions, is hard. However, this refers to knapsack problems with high-dimensional objects. In our case, every object has only two dimensions, a category and a visiting time. Nevertheless, this version of the algorithm should only be used for small problem instances.

4.3.2 Relaxed Linear Programming

In this version, we apply a fast approximation technique by relaxing the ILP problem to a linear programming (LP) problem, allowing any value between 0 and 1 for the p_i. We use the approach described by Kellerer et al. in [8] to relax the problem and compute a solution for the resulting LP utilizing GLPK again.

In the following, we give a brief description of the properties of this approximation. Let d be the number of dimensions, n be the number of items, then solving the relaxed ILP problem can lead to at most $\min(d, n)$ fractional values in the solution vector $\mathbf{p}$. We either pick all the items with $p_i = 1$ or we pick the best one with a fractional value, whichever provides a better score. This guarantees at least $1/(d+1)$ of the optimal score, or in our case $1/(m+2)$ where m is the number of categories

4.3.3 Greedy Solution

In this approach we sort the POIs in descending order according to their utility and then keep picking the best available POI that still fits into the knapsack. In our experiments we tried different utility functions, including simply taking the score of a POI or its score/visiting time ratio. However, the following utility described in [8] turned out to be most effective. Given a POI p whose category max visits is denoted by $k = \max_{\mathtt{cat}(p)}$, its utility is defined as $\mathtt{score}(p)/\left(1/k + v_p/t\right)$ where t is the knapsack time constraint. Clearly, we also have to consider that we are dealing with a multidimensional knapsack, so every time we want to pick a POI p, we have to make sure that we do not violate the category constraint, i.e., we have not reached $\max_{\mathtt{cat}(p)}$ POIs in this category yet. The greedy knapsack algorithm does not guarantee any theoretical bounds in terms of the quality of the solution, but it is the fastest to compute.

4.4 Hamiltonian Path Generation

Converting the set of selected POIs to an itinerary means finding the shortest Hamiltonian path with the two specified end points s and d, see line 30 in the pseudocode. This is equivalent to finding the shortest tour connecting all POIs and passing through the edge between s and d since removing the edge from s to d yields the Hamiltonian path.

Since computing the shortest tour is a hard problem, we use an approximation algorithm based on constructing a minimum spanning tree (MST) [18] (for computing the MST we use Kruskal's algorithm). However, before doing so, we need to make sure that the edge between s and d is included in the MST. We achieve this by setting its cost to -1, making it the shortest edge in the graph connecting the selected POIs. (In the case of $s = d$, we skip this step and directly compute the tour, which will be our itinerary.) We turn the MST into a tour by interpreting the MST as a tree rooted in d, having s as its left-most child, and traversing this tree in pre-order. Going back to d after the traversal forms a tour $\langle d, s, p_{i_1}, p_{i_2}, \dots, p_{i_n}, d\rangle$. All we need to do now to get a Hamiltonian path is to drop the first edge from d to s. In a metric graph this algorithm guarantees an itinerary that is at most a factor of 2 longer than the shortest path [18].

If we find that an itinerary goes over budget we have two options. If a user allows us to spend slightly more time, we can guarantee a lower bound on the score (more on this in Section 5). If we have to adhere strictly to the time budget, we can salvage the solution by removing the worst-scoring POI from the itinerary. Due to the triangle inequality, this cannot result in a longer path. We place the shortened path back into the queue of potential solutions.

5. THEORETICAL BOUNDS

An important property of our algorithm is that we can prove theoretical bounds for the time budget approximation as well as the number of generated cluster paths. But we first need to define the *radius* of a cluster.

DEFINITION 5. (Cluster radius) *The radius $r(\mathbf{c})$ of a cluster $\mathbf{c}$ is the longest edge between a POI $p \in \mathbf{c}$ and the medoid $\odot_{\mathbf{c}}$ of $\mathbf{c}$: $r(\mathbf{c}) = \max_{p\in\mathbf{c}} c(p, \odot_{\mathbf{c}})$. We denote the largest radius over all clusters as r_{max}.*

5.1 Bounding the Time Budget

For a moment let us assume that we know the optimal itinerary $\mathcal{P}^*$ for a given instance of OPMPC and compare it to the corresponding cluster path solution generated by our algorithm. If the travel cost is correct, depending on the knapsack solver we use, we can guarantee at least φ or $\varphi \cdot 1/(m+2)$ of the optimal score using an ILP or relaxed LP solver, respectively. But since we use an approximation for the travel time costs in a cluster path, the visiting time in the cluster path might be different to the one in the optimal itinerary. We might need extra time so that the knapsack solver is able to select the optimal set of POIs. The additionally required time is bounded, though.

THEOREM 5.1. *Let $\mathcal{P}^*$ be the optimal path, then there exists a cluster path $\mathcal{P}^*_{\mathbf{c}} = (s, \mathbf{c}_1 \dots, \mathbf{c}_{\mathcal{T}-1}, d)$ that contains the same POIs. The budget needed by $\mathcal{P}^*_{\mathbf{c}}$, denoted by $\hat{\mathcal{B}}$, is bounded by $\hat{\mathcal{B}} \le l_{max} \cdot 2r_{max} + \mathcal{B}$, where $\mathcal{B}$ is the budget used by $\mathcal{P}^*$ and $\mathcal{B} \le t_{max}$.*

PROOF. Every POI belongs to exactly one cluster, so given $\mathcal{P}^*$ we can map it into its corresponding cluster path $\mathcal{P}^*_{\mathbf{c}}$. In the following let a and b be two POIs. In $\mathcal{P}^*$ the cost for moving between a and b is $c(a, b)$. In CLIP we estimate the cost for moving from a to b within the same cluster by 0, which means CLIP can never overestimate the travel time for this case.

Now let a and b be POIs from two different clusters A and B, respectively. In CLIP we move from A to B via the medoids $\odot_A$ and $\odot_B$, costing us $c(\odot_A, \odot_B)$. Clearly, $c(a, \odot_A)+c(b, \odot_B) \le 2r_{\max}$ and therefore $c(a, \odot_A)+c(b, \odot_B)+c(\odot_A, \odot_B)-c(\odot_A, \odot_B) \le 2r_{\max}$. Due to the triangle inequality, this gives us $c(a, b) - c(\odot_A, \odot_B) \le 2r_{\max}$. So the difference between the true costs and the approximating cluster path costs is at most $2r_{\max}$.

In the worst case, we change clusters with every POI, i.e., $l_{\max} + 1$ times. Considering that we have the exact costs for leaving s and reaching d an upper bound for the additional time needed by CLIP is $l_{\max} \cdot 2r_{\max}$. Therefore, $\hat{\mathcal{B}} \le l_{\max} \cdot 2r_{\max} + \mathcal{B}$. $\square$

In the final step of our algorithm we have to turn a set of POIs into an itinerary connecting individual POIs.

Lemma 5.2. *Converting the set of POIs produced by* KnapsackSolver *to an itinerary connecting individual POIs can add at most $l_{max} \cdot 2r_{max}$ to the cost.*

Proof. We estimate the travel cost for the set of POIs with its original cluster path. First we create a spanning tree out of the cluster path in two steps. In the first step we connect the starting node to the destination node via the medoids of the clusters. This cost is already fully considered in our estimation. In the second step we connect all POIs directly with their corresponding medoid. An upper bound for the error in estimation is $r_{\max}$ per POI. In the worst case we have $l_{\max}$ POIs to connect to their medoids. Then we create a path from this spanning tree by doubling the edges between medoids and POIs and traversing the graph via an Euler tour. This adds at most $l_{\max} \cdot 2r_{\max}$ to the cost. □

In the algorithm we use for constructing a Hamiltonian path (Section 4.4), we create the minimum spanning tree and avoid visiting edges twice by shortcutting directly to the next POI. Consequently, in practice the additional time that CLIP would need is much lower than $l_{\max} \cdot 2r_{\max}$.

5.2 Bounds on Run Time

The key factor in determining the run time of our algorithm is the number of generated cluster paths. A naïve brute-force approach on the level of individual POIs would generate $\sum_{i=0}^{n-1} \prod_{j=0}^{i} (n-j)$ different paths (where n is the total number of POIs), which is clearly infeasible. A first step is discarding all POIs belonging to unwanted categories and creating a set of clusters C of the POIs that are left. This would still generate $\sum_{i=0}^{|C|-1} \prod_{j=0}^{i} (|C|-j)$ cluster paths. However, by introducing additional pruning we are able to push this number down further.

We have the constraint $l_{\max}$ on the maximum number of POIs per category, this means we have an upper bound of $\sum_{i=0}^{l_{\max}-1} \prod_{j=0}^{i} (|C|-j)$. Due to the removal of dominated paths, for all paths containing q clusters ($q \geq 3$), we only keep $\frac{1}{(q-1)!}$ of these paths. So in total we are left with $|C| \sum_{i=0}^{l_{\max}-1} \prod_{j=1}^{i} \frac{|C|-j}{j}$ paths. Note that this is an upper bound, as we are not considering here the two other pruning strategies we employ (going over the budget $t_{\max}$ and eliminating paths whose score falls below a certain threshold), as it is difficult to exactly quantify their impact.

6. EXPERIMENTAL EVALUATION

We evaluated CLIP experimentally, comparing different variants of it with two baseline algorithms. The two aspects we are mainly interested in are the run time performance and the score of the generated itineraries.

6.1 Compared Algorithms

In addition to the CLIP algorithm, we implemented a simple greedy heuristic and an algorithm that computes the optimal solution. The optimal algorithm generates all possible itineraries on a POI level up to the time budget $t_{\max}$. Although we introduce some pruning, such as the $l_{\max}$ and domination pruning also used for CLIP, the run time performance of this algorithm is not competitive. We only use it as a baseline for small problem instances.

In the following we describe the simple greedy heuristic. Given a partial itinerary $\mathcal{I} = \langle s, p_1, \ldots, p_i, d \rangle$, the simple greedy heuristic expands $\mathcal{I}$ by selecting as p_{i+1} the POI that currently improves $\mathcal{I}$ the most. The contribution of a potential POI p_{i+1} is determined by its utility, which is computed by dividing its score by its cost: $\texttt{score}(p_{i+1})/(v_{p_{i+1}} + c(p_i, p_{i+1}) + c(p_{i+1}, d))$. A POI p_{i+1} that violates a category constraint or is too far away to be reached can be discarded immediately. After adding the POI with the best utility to $\mathcal{I}$, we continue the algorithm with the remaining set of POIs (recomputing the utilities each time) until we can add no further points due to the constraints. This algorithm is also used to initialize CLIP, see line 8 of the pseudocode.

6.2 Data Sets

For the real-world data we obtained a data set for the city of San Francisco (with a total of 4215 POIs spanning 30 categories) and one from the municipality of Bolzano (1830 POIs in nine different categories).

For some tests, such as the comparison with the optimal algorithm, the real data sets were too large, so we generated some artificial networks as well. Nevertheless, we made sure that these networks imitated the structure of the real-world data sets closely. In particular, there are three important properties we identified. First of all, we use a grid-like structure to model the street network of a city. Second, cities tend to have areas in which POIs of certain types cluster, e.g., areas with restaurants, theaters, or shops. For each category a random point is selected, this point becomes the center of the cluster. Other POIs belonging to the same category are placed at a random angle and random distance from this center. Third, the distances between points of interests on a map follow a power law distribution, so we used a Pareto distribution for computing the random distances (the angles are uniformly distributed).

6.3 Hardware and Software

The experiments were conducted on a server machine with 2 Intel Xeon X5550 CPUs (2.67GHz), each of them having 4 cores and hyper-threading enabled, thus resulting in overall 16 virtual cores. The server has 48GB of main memory and the Java virtual machine was setup to halt the execution if it needed more than 45GB, but this never happened.

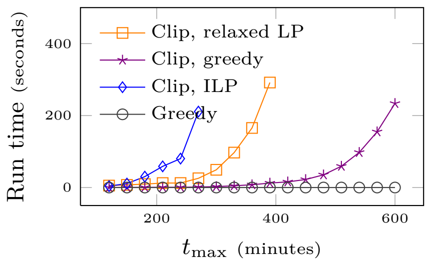
The server is running Ubuntu 12.04 server edition and the algorithms were implemented in Java, compiled with javac version 1.7.0_45. CLIP uses the GNU Linear Programming Kit (GLPK), version 4.45, to solve the multidimensional Knapsack problem. The optimal algorithm was implemented in C++ and compiled with gcc version 4.8.1.

6.4 Experimental Results

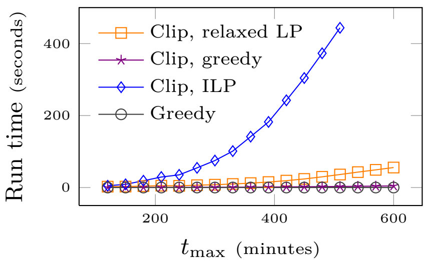
In order to get informative and representative results each individual experiment was repeated thirty times, ten times with different starting and destination points for three different sets of category constraints. The first set was composed of three categories, the second of five, and the third of eight. The maximum number of visits was two per category. If we did not get a result within 20 minutes, we stopped the process (that is the reason for some measurements stopping in the middle of a graph).

Run times.

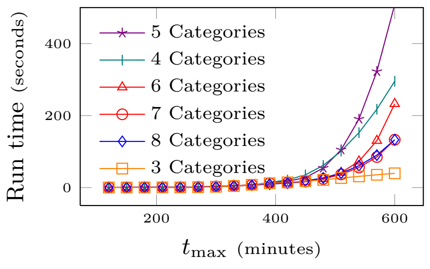
Figures 5(a) and (b) show the run times of the different



(a) San Francisco

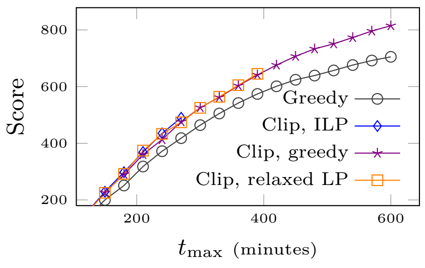


(b) Bolzano

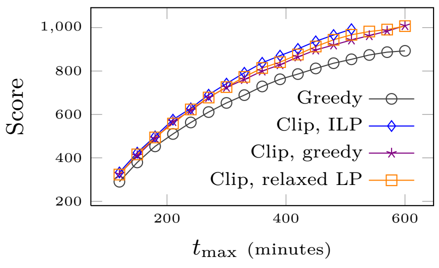


(c) San Francisco (various category constraints)

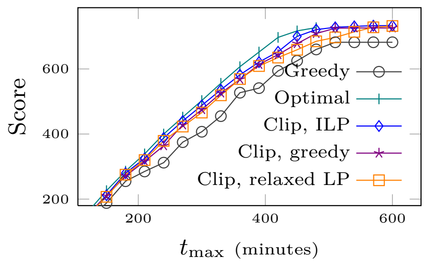
Figure 5: Results for run time



(a) San Francisco



(b) Bolzano



(c) Artificial network

Figure 6: Results for score

algorithms plotted against the budget $t_{\max}$. For large budgets the pure greedy strategy is the fastest algorithm. Since all variants of CLIP use the same cluster path generator, the difference in run time is caused by the knapsack solver. Clearly, the ILP solver is slowest, as in general solving an integer linear programming problem is an NP-hard problem. Using a relaxed LP solver brings down the run time considerably, but not to the level of the greedy knapsack solver. We were not overly surprised by these result, however, an interesting question is: what do we get in return for the additional run time of the ILP and relaxed LP solver? But before answering this question, we present one intriguing observation in connection with the run time. Figure 5(c) shows the run times of greedy CLIP for different sets of category constraints, using 3, 4, 5, 6, 7, and 8 categories with maximally two visits each. We expected the run time to go up with an increasing number of categories (and therefore increasing number of potential POIs to visit). Instead the run time goes up with 3, 4, and 5 different categories and then decreases again with 6, 7, and 8 categories. We have no explanation for this effect yet, but we suspect it has to do with properties of the underlying network, as we did not observe it for the map of Bolzano. Currently, we are investigating this further.

Score.

From the results in Figures 6(a) and (b) we can see that all CLIP variants outperform the pure greedy strategy in terms of the achieved score (POIs have scores between 0 and 99). There is almost no difference between ILP and relaxed LP, and the greedy knapsack solver is only slightly worse. In summary, the cost/benefit ration of running ILP (or even relaxed LP) is in most cases not good enough to justify running them. Figure 6(c) shows a comparison to the score of the optimal solution. As the San Francisco and Bolzano networks were too large to compute the optimum, we used a smaller artifical network. We show that the solutions computed by CLIP are usually closer to the optimal score than the one generated by the pure greedy algorithm.

φ-pruning.

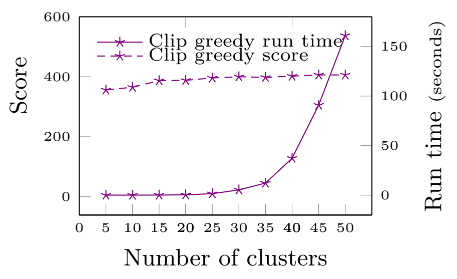
We ran CLIP with different values for φ: 0.1, 0.5, and 0.9. For the San Francisco and Bolzano data sets CLIP always generated the same scores, regardless of the value of φ (only for one artificial network we observed a difference of a single point in the scores). Thus, we run CLIP with $\varphi = 0.1$, resulting in the most aggressive pruning of the three values.

Clustering.

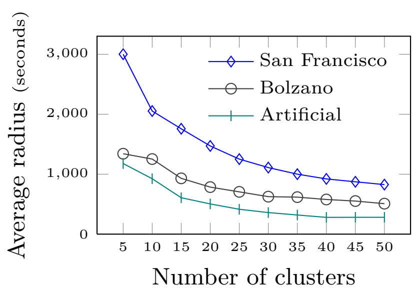
One question that is still open is the number of clusters we should use. Figure 7(a) shows how the run time of CLIP with a greedy knapsack solver goes up with the number of clusters (curve starting at bottom left). What we can also see is that the score only improves marginally when increasing the number of clusters, i.e., we already get good solutions with a fairly small number of clusters. In the case of the San Francisco data set (which is shown in Figure 7(a)), 20 to 30 clusters are already enough to find an itinerary with a good score (curve starting at top left). The important parameter to optimize, however, is not the number of clusters, but the radius. As shown in Section 5, the radius has a big impact when it comes to the accuracy of our approach. Figure 7(b) illustrates the diminishing returns we get for increasing the number of clusters: at the beginning the radius drops rapidly, but then the curve starts to level off.

“Magic Clusters”.

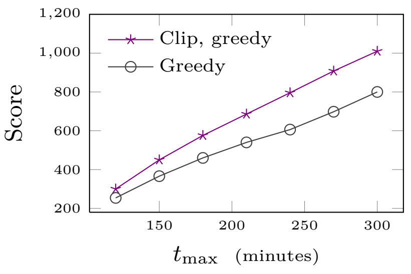
One pathological case for heuristics solving the orienteering problem are high-scoring POIs tucked away in a hard-to-reach corner of the network. We call these groups of POIs “magic clusters”, as they can turn a mediocre partial itinerary into a top-scoring one. Heuristics have to find these clusters and also have to make the correct decision whether it is worthwhile to visit them. Figure 7(c) shows the results



(a) Run time/score against no of clusters



(b) Radius of clusters



(c) "Magic clusters"

Figure 7: Other results

of an experiment using "magic clusters" in an artificial network. As can be clearly seen, the simple greedy heuristic struggles to find a good solution, while CLIP is able to cope with this situation.

7. CONCLUSION AND FUTURE WORK

We define a more realistic context for tourist trip planning by adding categories to points of interests, making it easier for a user to provide feedback on preferred types and numbers of POIs to be included in an itinerary. We develop an algorithm, CLIP, for efficiently solving orienteering problems with maximal cardinality constraints and show that it is able to generate itineraries very close to the optimal.

For future work we plan to parallelize CLIP, making it faster and scalable. More importantly, we have identified a set of other features that will help us in creating an even more realistic user experience when it comes to trip planning. Possible extensions to CLIP that we are looking into are hierarchies of categories, POIs belonging to multiple categories, time windows, financial budget constraints, and arrangements for group travel. Another important case we plan to investigate are directed graphs, which may make it more difficult to prove theoretical bounds.

8. ACKNOWLEDGMENTS

This work was partially funded by the INTERREG IV project O-STAR, no. 5343. We also thank Ulf Leser and the anonymous referees for their suggestions on improving an earlier version of the paper.

9. REFERENCES

[1] 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.*, 37(2):653–670, May 2007.

[2] C. Chekuri, N. Korula, and M. Pál. Improved algorithms for orienteering and related problems. In *SODA'08*, pages 661–670, 2008.

[3] C. Chekuri and M. Pál. A recursive greedy algorithm for walks in directed graphs. In *FOCS'05*, pages 245–253, 2005.

[4] D. Gavalas, C. Konstantopoulos, K. Mastakas, and G. Pantziou. A survey on algorithmic approaches for solving tourist trip design problems. *Journal of Heuristics*, 20(3):291–328, 2014.

[5] M. Gendreau, G. Laporte, and F. Semet. A branch-and-cut algorithm for the undirected selective traveling salesman problem. *Networks*, 32(4):263–273, 1998.

[6] GLPK. GNU Linear Programming Kit. `http://www.gnu.org/software/glpk`.

[7] C. Keller. Algorithms to solve the orienteering problem: A comparison. *European Journal of OR*, 41:224–231, 1989.

[8] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[9] Y.-C. Liang, S. Kulturel-Konak, and A. Smith. Meta heuristics for the orienteering problem. In *CEC '02*, pages 384 –389, 2002.

[10] E. H.-C. Lu, C.-Y. Lin, and V. S. Tseng. Trip-mine: An efficient trip planning approach with travel time constraints. In *MDM'11*, pages 152–161, 2011.

[11] R. Ramesh, Y.-S. Yoon, and M. H. Karwan. An optimal algorithm for the orienteering tour problem. *INFORMS Journal on Computing*, 4(2):155–165, 1992.

[12] M. N. Rice and V. J. Tsotras. Parameterized algorithms for generalized traveling salesman problems in road networks. In *ACM GIS'13*, pages 114–123, Orlando, Florida, 2013.

[13] G. Righini and M. Salani. Decremental state space relaxation strategies and initialization heuristics for solving the orienteering problem with time windows with dynamic programming. *Computers & OR*, 36(4):1191–1203, 2009.

[14] Z. Sevkli and F. E. Sevilgen. Variable neighborhood search for the orienteering problem. In *ISCIS'06*, pages 134–143, 2006.

[15] A. Singh, A. Krause, C. Guestrin, W. J. Kaiser, and M. A. Batalin. Efficient planning of informative paths for multiple robots. In *IJCAI'07*, pages 2204–2211, 2007.

[16] F. Tasgetiren and A. Smith. A genetic algorithm for the orienteering problem. In *IEEE Congress on Evolutionary Computation*, 2000.

[17] T. A. Tsiligrides. Heuristic methods applied to orienteering. *J. Operation Research Society*, 35(9):797–809, 1984.

[18] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2003.

[19] Q. Wang, X. Sun, B. L. Golden, and J. Jia. Using artificial neural networks to solve the orienteering problem. *Annals of OR*, 61:111–120, 1995.