# Hybrid Best-First Greedy Search for Orienteering with Category Constraints

Paolo Bolzoni and Sven Helmer(✉)

Free University of Bozen-Bolzano, Bolzano, Italy
{paolo.bolzoni,sven.helmer}@unibz.it

**Abstract.** We develop an approach for solving rooted orienteering problems with category constraints as found in tourist trip planning and logistics. It is based on expanding partial solutions in a systematic way, prioritizing promising ones, which reduces the search space we have to traverse during the search. The category constraints help in reducing the space we have to explore even further. We implement an algorithm that computes the optimal solution and also illustrate how our approach can be turned into an anytime approximation algorithm, yielding much faster run times and guaranteeing lower bounds on the quality of the solution found. We demonstrate the effectiveness of our algorithms by comparing them to the state-of-the-art approach and an optimal algorithm based on dynamic programming, showing that our technique clearly outperforms these methods.

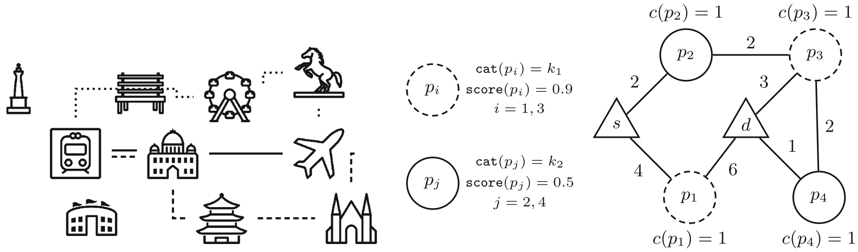
## 1 Introduction

Imagine a user arriving at the train station of a city, depicted on the left-hand side of Fig. 1(a), and needing to be at the airport, located on the right-hand side of the figure, five hours later[1]. They do not want to immediately go to the airport, but have a look at the city first. If the user takes the shortest route from the train station to the airport, represented by the solid line, they are only able to see a basilica on the way and still have a lot of spare time when arriving at the airport. According to the ratings of a tourist guide, the basilica, a pagoda, and a cathedral are the top points of interest (POIs) that can be visited en route to the airport within five hours (see dashed line). While this makes much better use of the time, the user may not be in the mood to visit these POIs, as they may be tired from traveling and would like a more relaxing route. In this instance, the dotted line, connecting a park, a ferris wheel, and a statue, is much more appropriate. As this example shows, the goal is to find a trip with the best points of interest that makes good use of the available time while at the same time considering the preferences of a user.

Finding itineraries for domains such as tourist trip planning and logistics often involves solving an orienteering problem. This is because those tasks are

[1] The icons are from icons8.com, used under Creative Commons License CC BY-ND 3.0. To view a copy, visit https://creativecommons.org/licenses/by-nd/3.0/.


M. Gertz et al. (Eds.): SSTD 2017, LNCS 10411, pp. 24–42, 2017.
DOI: 10.1007/978-3-319-64367-0_2




(a) Motivational example[1] (b) Itinerary with four POIs

**Fig. 1.** Example scenarios

not about determining the shortest path, but the most attractive or the one covering the most needy customers while satisfying a strict time constraint. We focus on a variant that assumes that every point of interest (POI) or customer has a category. This categorization helps a user in expressing preferences, e.g. a tourist may only be interested in certain types of venues, such as museums, galleries, and cafes, while certain vehicles may only be able to serve particular customers.

In general, orienteering is an NP-hard problem, and adding categories does not change this fact [2]. We propose an approach based on a best-first strategy to explore the search space, meaning that we first expand the partial solutions that show the greatest potential. We do so with the help of a function approximating the attractiveness of POIs that can still be added. Similar to admissible heuristics in an A*-search, this function needs to satisfy certain properties. Additionally, we are able to prune partial solutions that cannot possibly result in an optimal route.

Even though this technique will speed up the search for an optimal solution by pruning unpromising partial solutions, in the worst case it still has an exponential run time. Therefore, we describe how to transform our method into much more efficient approximation algorithms, creating different variants with important properties concerning the quality of the generated solution and the run time. In summary, we make the following contributions: we show how to apply a best-first strategy to the problem of orienteering with category constraints; we turn the optimal algorithm into different approximation algorithms proving lower bounds for the quality of a solution or upper bounds for the run time; in the experimental evaluation we compare our approach to state-of-the-art algorithms, demonstrating its effectiveness. For some scenarios we improved the quality of the solutions by about 10% with a run time of just five seconds (with the main competitor taking up to 50 s and producing worse solutions).

## 2 Related Work

Introduced by Tsiligrides in [16], the orienteering problem (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 [6,10] as well as dynamic programming techniques [9,12] 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 [7,16] and (partial) path improvement [1,3,13]. Meta-heuristics, such as genetic algorithms [15], neural networks [17], and ant colony optimization [8] have also been tested. For a recent overview on orienteering algorithms, see [5]. 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 [11], while the generalized orienteering version maximizes the number of visited points of interest (POIs) given a fixed time budget. However, as there are no scores, no trade-offs between scores and distances are considered.

Adapting an existing algorithm for OP would be a natural starting point for developing an approximation algorithm considering categories. 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 [4]. 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 [14]. It has been adapted by us for OPs on road networks with category constraints in the form of the CLuster Itinerary Planning algorithm (CLIP) [2], where we use agglomerative hierarchical clustering and multidimensional knapsack to separate routing from POIs selection.

## 3 Problem Formalization

We assume a set $\mathbf{P}$ of points of interest (POIs) $p_i, 1 \leq i \leq n$. 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 edges, $e_l \in \mathbf{E} = \{(x, y) \mid x, y \in \mathbf{P} \cup \{s, d\}\}$ connect 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 $c(p_i)$ that denotes its visiting time. 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 $p_i$, $\mathtt{cat}(p_i)$ denotes the category $p_i$ belongs to and $\mathtt{score}(p_i)$ its score or reward, 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 $t_{\max}$.

**Definition 1** (Itinerary). *An itinerary $\mathcal{I}$ starts from a starting point $s$ and finishes at a destination point $d$ ($s$ and $d$ can be identical). It includes an ordered sequence of connected nodes $\mathcal{I} = \langle s, p_{i_1}, p_{i_2}, \ldots, p_{i_q}, d\rangle$, 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 $\mathcal{I}$, $\texttt{cost}(\mathcal{I}) = c(s, p_{i_1}) + c(p_{i_1}) + \sum_{j=2}^{q}(c(p_{i_{j-1}}, p_{i_j}) + c(p_{i_j})) + c(p_{i_q}, d)$, and its* score *to be the sum of the scores of the individual POIs visited, $\texttt{score}(\mathcal{I}) = \sum_{j=1}^{q} \texttt{score}(p_{i_j})$.*

*Example 1.* Figure 1(b) shows an example with four POIs, $p_1, p_2, p_3$, and $p_4$, along with their distances, visiting times, scores, and categories. We simplify the graph slightly to keep it readable: all POIs of the same category have the same score and we also omit some edges. Three example itineraries with one, two, and three POIs, respectively, are: $\mathcal{I}_1 = \langle s, p_1, d\rangle$, $\mathcal{I}_2 = \langle s, p_2, p_3, d\rangle$, and $\mathcal{I}_3 = \langle s, p_2, p_3, p_4, d\rangle$. Their costs and scores are as follows:

- $\mathcal{I}_1 = \langle s, p_1, d\rangle$: $\texttt{cost}(\mathcal{I}_1) = 4 + 1 + 6 = 11$, $\texttt{score}(\mathcal{I}_1) = 0.9$;
- $\mathcal{I}_2 = \langle s, p_2, p_3, d\rangle$: $\texttt{cost}(\mathcal{I}_2) = 2 + 1 + 2 + 1 + 3 = 9$, $\texttt{score}(\mathcal{I}_2) = 0.5 + 0.9 = 1.4$;
- $\mathcal{I}_3 = \langle s, p_2, p_3, p_4, d\rangle$: $\texttt{cost}(\mathcal{I}_3) = 2 + 1 + 2 + 1 + 2 + 1 + 1 = 10$, $\texttt{score}(\mathcal{I}_3) = 0.5 + 0.9 + 0.5 = 1.9$.

Given traveling and visiting times as well as scores (obtained from sources such as tourist offices or web sites and possibly combined with user preferences), we need to build an itinerary starting at $s$ and ending at $d$ from a subset $P$ of $\mathbf{P}$ with duration smaller than $t_{\max}$ and maximum cumulative score. 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. The category constraints are directly determined via user preferences. We are now ready 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 $p_i \in \mathbf{P}$, each with $\texttt{score}(p_i)$, visiting times $c(p_i), 1 \leq i \leq n$, 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, $\texttt{score}(\mathcal{I})$, is maximized;*
- *no more than $\max_{k_j}$ POIs are used for category $k_j$;*
- *the time constraint is met,* i.e., $\texttt{cost}(\mathcal{I}) \leq t_{max}$.

*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 $t_{\max}$.

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.

## 4 Best-First Search Strategy

We roughly follow a best-first strategy, meaning that we keep all solutions generated so far in a priority queue sorted by their potential score. A solution is represented by the set of POIs it contains. In each step, we take the solution with the highest potential score from the queue, expand it with POIs that have not been visited yet, and re-insert the expanded solutions back into the queue. An important difference to the classic best-first search is that in our case it is not straightforward to identify the goal state, as we do not know the score of the optimal itinerary a priori. Consequently, we have to run the algorithm until there are no solutions left in the queue that have a higher potential score than the best found so far (which then becomes the result we return). Applying this approach in a straightforward fashion is not practical as OPMPC is an intractable problem. Therefore, after presenting an algorithm that computes the optimal solution, we turn it into an approximation algorithm.

### 4.1 Potential Score

An important aspect of our search strategy is the computation of the potential score of a partial solution, which has to be an upper bound of the score of the fully expanded solution based on this partial solution. Basically, this follows the principle of admissible heuristics found in the A*-algorithm, with the difference that we never underestimate the value, since we consider scores and not costs.

**Definition 3** (Potential score). *Given a (partial) itinerary $\mathcal{I} = \langle s, p_1, p_2, \ldots, p_i, d\rangle$, its potential score $\mathtt{pot}(\mathcal{I})$ is defined as follows:*

$$\mathtt{pot}(\mathcal{I}) = \mathtt{score}(\mathcal{I}) + \mathtt{extra}(\mathcal{I})$$

*where $\mathtt{score}(\mathcal{I}) = \sum_{j=1}^{i} \mathtt{score}(p_j)$ and $\mathtt{extra}(\mathcal{I})$ is a heuristic never underestimating the additional score that is still possible for $\mathcal{I}$.*

For efficiency reasons, we need to find an approximation for $\mathtt{extra}(\mathcal{I})$ that is as small as possible without underestimating it. We do so by getting a rough estimate of the top-scoring POIs that still fit into the partial itinerary $\mathcal{I}$. In a first step, we discard all POIs in the set of remaining POIs, $p_k \in \mathbf{P} \setminus \mathcal{I}$, that are too costly for traveling from $p_i$ via $p_k$ to $d$, as $\mathtt{cost}(\mathcal{I}) - c(p_i, d) + c(p_i, p_k) + c(p_k) + c(p_k, d) > t_{\max}$. We call this set of remaining POIs $\mathbf{P}_{rem(\mathcal{I})}$. Having reduced the set of POIs, we turn to category constraints for a more accurate approximation. For each category we know how many POIs still fit into the itinerary without violating the category constraints. As previously defined,

$\max_{k_j}$ is the maximum value of POIs that can be chosen for category $k_j$. Let $u_{k_j} = |\{p_i \in \mathcal{I} | \mathtt{cat}(p_i) = k_j\}|$ be the number of POIs of category $k_j$ currently found in $\mathcal{I}$, then we know that we can only fit $\max_{k_j} - u_{k_j}$ more POIs of category $k_j$ into $\mathcal{I}$. For each category $k_j$, we sort the POIs in $\mathbf{P}_{rem(\mathcal{I})}$ according to their score in descending order and add up the scores of the top $\max_{k_j} - u_{k_j}$. This comprises $\mathtt{extra}(\mathcal{I})$.

### 4.2 Our Algorithm

Algorithm 1 shows our approach in pseudocode. For the moment disregard the parts marked in gray: these are optimizations that will be discussed in the following section. Partial solutions are kept in a double-ended priority queue (also called deque). The deque is initialized with the empty solution $e$ just containing $s$ and $d$ and a score of 0. We take the solution with the highest potential score out of the deque and expand it, one by one, with the POIs that have not been visited yet. In order to figure out whether a solution violates the time constraint $t_{\max}$, we have to compute the (shortest) length of the itinerary of a solution. As every partial solution contains only a limited number of POIs, we compute this length in a brute-force way. We discard any expanded solutions that violate the time constraint or any of the category constraints and put all valid expansions back into the deque. If an expanded solution is better than the current best solution, we update it. We continue until the potential score of the solution taken from the deque is smaller than the best found so far (which then becomes the answer we return).

### 4.3 Further Optimizations

The score of the current best solution helps us to prune earlier and more aggressively, because only partial solutions with a better potential score have to be kept. We initialize the best solution found so far with an itinerary found quickly using a greedy algorithm, which works as follows. Given a partial itinerary $\mathcal{I} = \langle s, p_1, p_2, \ldots, p_i, d \rangle$ (starting with the empty itinerary), the greedy strategy adds to this path a POI $p \in \mathbf{P}_{rem(\mathcal{I})}$ such that its $\mathtt{utility}(p) = \frac{\mathtt{score}(p)}{c(p_i,p)+c(p)+c(p,d)}$ is maximal and no constraints are violated. We repeat this until no further POIs can be added to the itinerary.

We can also use a greedy approach to improve the current best solution while the algorithm is running (see lines 15 to 17). When expanding a partial solution taken from the deque, we also complete it with the greedy strategy. This improves the current best solution more rapidly, leading to more aggressive pruning, and is especially useful in earlier stages of the algorithm, as there is still a lot of unexplored search space. We may want to stop the greedy expansion in later stages, due to it being less effective. The threshold $g \in [0, 1]$ defines when the algorithm is in an *early stage*, it checks each expanded solution $z$: whenever the ratio ${}^{z.\mathtt{cost}}/{}_{t_{\max}} \leq g$, it does a greedy expansion. We investigate this parameter in more detail in Sect. 7 on the experimental evaluation.

**Algorithm 1.** $b \leftarrow$ `MainLoop`$(q, G)$

**Input:** query $q$ (consisting of $s$, $d$, $t_{\text{max}}$, and category constraints), graph $G$, set $\mathbf{P}$ of all POIs

**Output:** best solution $b$

1 $Q$ priority deque for partial solutions<br>
2 $e$.`cost` $\leftarrow c(q.s, q.d)$<br>
3 $e$.`extra` $\leftarrow$ `Extra`$(q, e, G, \mathbf{P})$<br>
4 $b \leftarrow$ `ExtendGreedily`$(q, e)$<br>
5 $Q$.`push`$(e)$<br>
6 **while not** $Q$.`empty`() **do**<br>
7     $s \leftarrow Q$.`popMaximum`()<br>
8     **foreach** $z \in$ `Expand`$(s, G)$ **do**<br>
9         **if** *$z$ violates any category constraint* **then continue**<br>
10         $z \leftarrow$ `ComputePathAndScore`$(z)$<br>
11         **if** $z$.`cost` $> q.t_{max}$ **then continue**<br>
12         $z$.`extra` $\leftarrow$ `Extra`$(q, z, G, \mathbf{P})$<br>
13         **if** $z$.`score` $> b$.`score` **then** $b \leftarrow z$<br>
14         $Q$.`push`$(z)$<br>
15         **if** *at an early stage* **then**<br>
16             $y \leftarrow$ `ExtendGreedily`$(q, z)$<br>
17             **if** $y$.`score` $> b$.`score` **then** $b \leftarrow y$<br>
18     **while** $b$.`score` $> Q$.`minimum`().`score` $+ Q$.`minimum`().`extra` **do**<br>
19         $Q$.`popMinimum`()<br>
20 **return** $b$

## 5 Approximation Algorithms

Even though pruning and further optimizations speed up the search, in the worst case we still face exponential run time. After all, OPMPC is an NP-hard problem. Here we present variations of Algorithm 1 that turn it into an approximation algorithm. In principle, we can guarantee an upper bound on the run time or a lower bound on the score.

### 5.1 Bounding the Score

In line 18 of Algorithm 1 we remove (partial) solutions from the end of the deque that have a potential score smaller than the best solution $b$ found so far. Since solutions are sorted by an overestimation of their score, we never accidentally get rid of a partial solution that could be expanded into the optimal one. However, we could prune more aggressively by introducing a *cut factor* $c$ ($c \geq 1$) and checking whether $c \cdot b$.`score` $> Q$.`minimum`().`score` $+ Q$.`minimum`().`extra` (setting $c$ to 1 results in the original algorithm). The larger $c$, the more partial solutions will be ignored. However, by doing so we also risk losing the optimal and other good solutions.

Nevertheless, we are sure to get a solution that guarantees at least $\alpha = 1/c$ of the score of the optimal solution.

### 5.2 Bounding the Run Time

While the approach in the previous section makes sure that we always get a certain amount of the optimal score, we are not able to bound its run time. All we know is that it is faster, since we prune more partial solutions. By modifying Algorithm 1 in a different way, we can obtain a run time bound. Instead of allowing the queue to become arbitrarily long, we limit its length to a maximum of $l_{\max}$ entries. In this way, we put a limit on the number of partial solutions that can be expanded. Before pushing an expanded solution $z$ back into the deque in line 14, we check whether there is still space. If there is not enough space, we remove the partial solution with the lowest potential score (this could also be $z$). While we cannot determine the quality of the answer a priori, we know the value for $\alpha$ upon completion of the algorithm if we keep track of the largest potential score of all the partial solutions we discarded. Assume that $z_{\max}$ has the largest potential score among the discarded solutions. Then we know that $\alpha = b/\texttt{pot}(z_{max})$, where $b$ is the optimal solution.

Another, more direct, way of controlling the run time is to set an explicit limit $r$ for it. After the algorithm uses up the allocated time, it returns the best solution found so far. For example, we can check between line 7 and 8 whether there is any time left. If not, we jump out of the while-loop. Again, it is not possible to prescribe a value for $\alpha$ beforehand, but we can determine its value when the algorithm finishes: in this case $z_{\max}$ is the partial solution with the largest potential score that is still in the deque when we stop.

## 6 Properties and Bounds

In the following we prove that conservative pruning (i.e. cut factor $= 1$) allows us to find the optimal solution. Additionally, we prove the bounds for the score and the run time of the approximation algorithms.

### 6.1 Correctness of Pruning

Let $\mathcal{I}_c$ be a *complete itinerary*, i.e., no further POIs can be added to $\mathcal{I}_c$ without violating a constraint. We have to prove that for every complete itinerary $\mathcal{I}_c$, $\texttt{pot}(\mathcal{I}_c) = \texttt{score}(\mathcal{I}_c) \leq \texttt{pot}(\mathcal{I}_p)$, where $\mathcal{I}_p$ is a partial itinerary of $\mathcal{I}_c$ ($\mathcal{I}_p \subset \mathcal{I}_c$). $\mathcal{I}_p \subset \mathcal{I}_c$ iff the POIs $p_1, p_2, \ldots, p_p$ in $\mathcal{I}_p$ are a subset of the POIs $p_1, p_2, \ldots, p_c$ in $\mathcal{I}_c$. The potential score of $\mathcal{I}_c$ is equal to its score, since $\texttt{extra}(\mathcal{I}_c) = 0$ (no more POIs can be added). We can actually show a stronger statement that includes the one above.

**Lemma 1.** *As we expand solutions, their potential score decreases: given two itineraries $\mathcal{I}_1$ and $\mathcal{I}_2$ with $\mathcal{I}_1 \subset \mathcal{I}_2$, it follows that $\texttt{pot}(\mathcal{I}_2) \leq \texttt{pot}(\mathcal{I}_1)$.*

*Proof.* We prove the lemma by structural induction.

**Induction step ($n \to n+1$):** Given two itineraries $\mathcal{I}_n$ with POIs $p_1, p_2, \ldots, p_n$ and $\mathcal{I}_{n+1}$ with POIs $p_1, p_2, \ldots, p_{n+1}$, we have to show that $\mathtt{pot}(\mathcal{I}_{n+1}) \leq \mathtt{pot}(\mathcal{I}_n)$

$$\begin{aligned}
&\Leftrightarrow \mathtt{score}(\mathcal{I}_{n+1}) + \mathtt{extra}(\mathcal{I}_{n+1}) \leq \mathtt{score}(\mathcal{I}_n) + \mathtt{extra}(\mathcal{I}_n)\\
&\Leftrightarrow \mathtt{score}(\mathcal{I}_n) + \mathtt{score}(p_{n+1}) + \mathtt{extra}(\mathcal{I}_{n+1}) \leq \mathtt{score}(\mathcal{I}_n) + \mathtt{extra}(\mathcal{I}_n)\\
&\Leftrightarrow \mathtt{score}(p_{n+1}) + \mathtt{extra}(\mathcal{I}_{n+1}) \leq \mathtt{extra}(\mathcal{I}_n)
\end{aligned}$$

Assuming that we can still add (up to) $k$ POIs to $\mathcal{I}_n$, $\mathtt{extra}(\mathcal{I}_n)$ is computed by taking the top-k POIs from $\mathbf{P}_{rem(\mathcal{I}_n)}$, whereas $\mathtt{extra}(\mathcal{I}_{n+1})$ is computed by taking the top-(k-1) POIs from $\mathbf{P}_{rem(\mathcal{I}_{n+1})}$. We have just expanded $\mathcal{I}_n$ by adding $p_{n+1}$ to it, so we know that $\mathbf{P}_{rem(\mathcal{I}_{n+1})} \cup \{p_{n+1}\} \subseteq \mathbf{P}_{rem(\mathcal{I}_n)}$, as $p_{n+1}$ is missing from $\mathbf{P}_{rem(\mathcal{I}_{n+1})}$ (compared to $\mathbf{P}_{rem(\mathcal{I}_n)}$) and $\mathbf{P}_{rem(\mathcal{I}_{n+1})} \cup \{p_{n+1}\}$ can at most reach $\mathbf{P}_{rem(\mathcal{I}_n)}$, because the remaining time in itinerary $\mathcal{I}_{n+1}$ is less than the remaining time in itinerary $\mathcal{I}_n$. Therefore, the choice we get when picking top-scoring POIs we pick for the left-hand side of the inequality is more restricted compared to the right-hand side.

**Base case (empty itinerary):** The empty itinerary $\mathcal{I}_0$ contains no POIs, therefore $\mathtt{pot}(\mathcal{I}_0) = \mathtt{extra}(\mathcal{I}_0)$ and we choose the top-scoring POIs from the largest possible set $\mathbf{P}_{rem(\mathcal{I}_0)}$, which means that $\mathcal{I}_0$ has the largest possible potential score.

### 6.2 Lower Bounding the Score

Using the approximation technique from Sect. 5.1, we can guarantee a lower bound for the score. At any given time, when discarding a partial solution $\mathcal{I}_p$, we know that its potential score $\mathtt{pot}(\mathcal{I}_p)$ is at most a factor $c$ greater than the best solution $b$ found so far: $c \cdot b \geq \mathtt{pot}(\mathcal{I}_p)$. If there is no further change for $b$, the ratio of achieving $\alpha = {}^1/_c$ of the best score holds, as $\mathtt{pot}(\mathcal{I}_p)$ never underestimates the score of a full expansion of $\mathcal{I}_p$. If $b$ is replaced by a new best score $b'$, we then know that the previously discarded partial solutions are actually closer to the optimal solution, as $b'$ can only be greater than $b$. In fact, the cut factor of the previously discarded partial solutions improves to $c' = {}^b/_{b'} \cdot c$. Additionally, the lemma in the previous section states that the potential score of an expanded solution can never go up, which means that we are also on the safe side here: we cannot lose a complete solution that is less than a factor $\alpha$ of the optimal solution by discarding $\mathcal{I}_p$.

Even when using the approximation techniques bounding the run time shown in Sect. 5.2, which do not allow us to set a lower bound for the score a priori, we can draw conclusions about the factor $\alpha$ after the algorithm finishes. When running our algorithm with a limited queue length, we keep track of the solution with the largest potential score $\mathtt{pot}(\mathcal{I}_{\max})$ that we have discarded. When execution stops, we can now compare $\mathtt{pot}(\mathcal{I}_{\max})$ to the best solution $b$ that is returned, knowing that $\alpha$ has to be at least ${}^b/_{\mathtt{pot}(\mathcal{I}_{\max})}$. Using an explicit time limit $r$, the largest (implicitly) discarded $\mathtt{pot}(\mathcal{I}_{\max})$ can be found at the head of the queue and we can compute $\alpha$ as indicated above. We can run our algorithm in an iterative fashion: once the time limit $r$ has been reached, but a user is not satisfied with the current level of quality, they can run the algorithm a while longer, checking $\alpha$ again. In this way, the quality of the solution and the run time can be balanced on the fly.

### 6.3 Upper Bounding the Run Time

When setting an explicit time limit $r$, bounding the run time is straightforward. In case of a limited queue length, drawing conclusions about the run time is more complicated. The worst case for a best-first search algorithm is a scenario in which all the POIs have the same score, the same cost, and a similar distance from $s$, $d$ and each other. In this case we first expand the empty partial solution generating all itineraries of length one. In the next step, we expand all these itineraries to those of length two, then all of those to those of length three, and so on, meaning that no pruning is taking place. We now give an upper bound for the number of generated itineraries and then illustrate the effect of a bounded queue length on this number.

We have $m$ categories $k_j$, $1 \leq j \leq m$, each with the constraint $max_{k_j}$ and the set $K_{k_j} = \{p_i | p_i \in P, cat(p_i) = k_j\}$ containing the POIs belonging to this category. The set of relevant POIs $R$ is equal to $\bigcup_{j=1, max_{k_j}>0}^{m} K_{k_j}$, we denote its cardinality by $|R|$ (assuming that $R$ only contains POIs actually reachable from $s$ and $d$ in time $t_{\max}$). For the first POI we have $|R|$ choices, for the next one $|R|-1$, and so on until we have created paths of length $\lambda = \sum_{j=1, max_{k_j}>0}^{m} max_{k_j}$. We cannot possibly have itineraries containing more than $\lambda$ POIs, as this would mean violating at least one of the category constraints. So, in a first step this gives us $\prod_{i=0}^{\lambda-1}(|R|-i)$ different itineraries. This is still an overestimation, though, as it does not consider the $t_{max}$ constraint, nor does it consider that once an itinerary has reached $max_{k_j}$ POIs for a category $k_j$, the set $K_{k_j}$ can be discarded completely when extending the itinerary further. Introducing a maximum queue length of $l_{\max}$, we arrive at the final number $\prod_{i=0}^{\lambda-1} \min((|R| - i), l_{\max})$.

## 7 Experimental Evaluation

We conducted our experiments on a Linux machine (Arch Linux, kernel version 4.8.4) with an i7-4800MQ CPU running at 2.7 GHz and 24 GB of RAM (of which 15 GB were actually used). The algorithms were written in C++, compiled with `gcc` 6.2 using `-O2`. The priority deque was implemented with an interval heap, the graph representing the city map with an adjacency list. For comparison with the CLIP we used black-box testing, configuring it with the default settings for cluster size and relaxed linear programming solver as suggested in [2]. The greedy strategy expands an empty itinerary containing just $s$ and $d$.

We averaged 25 different test runs for every data point in the following plots. For each test run we selected a start and end node randomly, such that $c(s, d) < t_{\max}$. When comparing different algorithms, we generated one set of queries and re-used it for every algorithm. Unless specified otherwise, as default values for the parameters we use two hours for $t_{\max}$, the largest number of categories possible for a graph (four or nine), a category constraint of two for each category, a cut ratio of 1.2, a greedy expansion threshold of 1, an unbounded queue length, and unlimited run time.

### 7.1 Data Sets

We used several different data sets in our evaluation. The first one was an artificial one consisting of a grid with 10,000 nodes ($100 \times 100$) with a total of 3,000 POIs in four different categories. The distance between neighboring nodes in the grid was 60 s. The POIs were randomly scattered in the grid, had a visiting time between three minutes and one hour and a score between 1 and 100.

The second artificial network is a spider network having the same number of nodes and POIs as the grid, but the edges are placed as a 100-sided polygon with 100 levels. The edges between levels are 100 s, and the central, and smallest, polygon has sides of 8 s. The other levels become larger and larger as in a Euclidean plane.

The two real-world data sets were a map of the Italian city of Bolzano with a total of 1,830 POIs in nine categories and a map of San Francisco with a total of 1,983 POIs in four categories. Here the length of the edges is computed assuming a walking speed of $1 \frac{m}{s}$.

### 7.2 Effects of Parameters

First of all, we illustrate the effects of the different parameters on the performance of our approximation algorithms. Namely, these are the values for the greedy expansion threshold, the cut ratio, the queue length limit, and the run time limit. We also show how the number of categories selected by a user influences our performance.

**Greedy Expansion Threshold.** Figure 2 shows the impact of the greedy expansion threshold on the performance of our algorithm (here on a grid network; the results for the other networks look very similar).

As can be seen in Fig. 2 (left), the effect on the score is minimal (for comparison a pure greedy strategy is also shown). However, the improvements in terms of run time are significant (Fig. 2, right). Running a greedy expansion during the execution of the algorithm updates the best solution found so far much faster, resulting in better pruning. On average, it was always worth running a greedy expansion right up to the point when the algorithm finishes, since it can be done very quickly. In fact, toward the end of the execution this can be done even faster, as only one or two more POIs can be added to an itinerary.

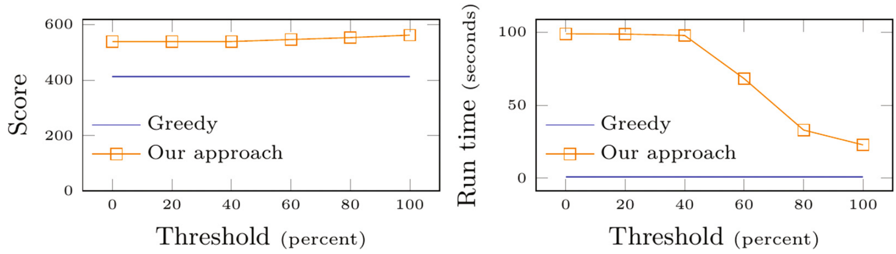
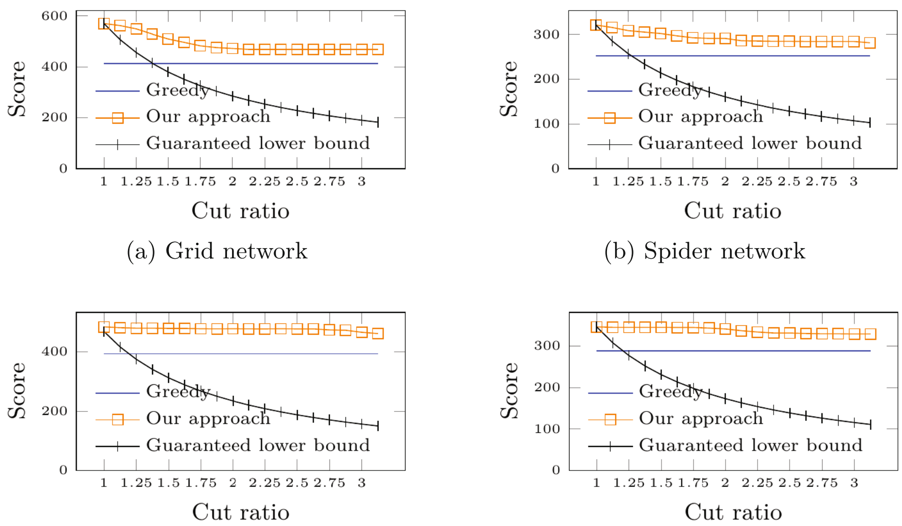



**Fig. 2.** Varying the greedy expansion threshold

**Cut Ratio.** Next, we investigate the impact of the cut ratio. Increasing the cut ratio allows us to prune more aggressively, albeit at the price of sometimes losing the optimal solution. In Fig. 3 we depict the results for varying the cut ratio from 1 (optimal case) to 2.5 (guaranteeing at least 0.4 of the optimal score). As expected, the score of the found solutions goes down when increasing the cut ratio. This is the case for all data sets. Nevertheless, it does so much slower than the guaranteed lower bound for the score and it also stays well above the score for the solutions found by the simple greedy algorithm. This is especially true for the real-world data sets (Figs. 3(c) and (d)). We found that our algorithm finds good solutions early (more on this in Sect. 7.2 about setting a fixed run time limit).




(a) Grid network

(b) Spider network

(c) Bolzano network ($t_{\max} = 3$h)

(d) San Francisco network ($t_{\max} = 3$h)

**Fig. 3.** Increasing the cut ratio

The impact of the cut ratio on the run time is clearly noticeable (see Fig. 4). For some data sets, namely the grid and San Francisco networks, increasing the cut ratio brings down the run time significantly. Grid networks, including the artificial grid and San Francisco, whose map is composed of many grid-like structures, have a very long run time for small cut ratios, while spider networks do not exhibit this behavior. (Bolzano, which has grown organically over centuries around an old city center, shares some of the features of a spider network.) The different edge lengths make the central part of the spider network easier to explore, whereas the external parts are more isolated. As a consequence, the computation of $\texttt{extra}(\mathcal{I})$ is more precise, meaning that we get a more accurate picture about the potential scores of the yet unexpanded solutions that are still in the queue, making it easier to prune partial solutions. In a grid-like network, there are lots of alternatives, making it more difficult to compute $\texttt{extra}(\mathcal{I})$ accurately, so the potential scores of some unexpanded solutions may still be high, and we need more time to check them.

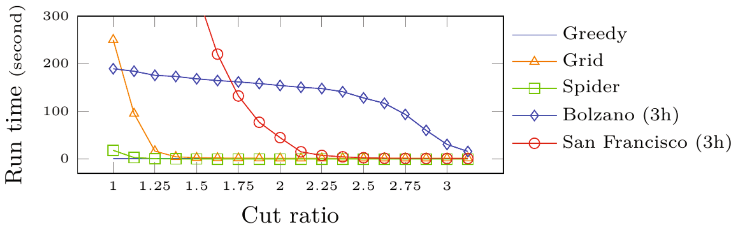



**Fig. 4.** Effect of the cut ratio on run time

**Queue Length.** Now we look at the impact of bounding the queue length to shorten the run time of our algorithm. The left-hand column of Fig. 5, labeled (a), shows the results for the grid network, the right-hand column, labeled (b), those for the spider network. The cut ratio is set to 1 to measure only the impact of the queue length limit.[2] The size of the queue on the x-axis is measured relatively to the number of POIs in the graph. Having a very short queue lowers the score slightly. Please note that the score is expressed as a ratio of the optimal one and that the y-axis does not start at 0 (greedy scored 0.73 for the grid and 0.78 for the spider network). Unsurprisingly, the run time goes up with increasing queue length and it will eventually reach the run time of the unbounded queue (represented by horizontal lines).

**Run Time Limit.** In Fig. 6 we show results for setting an explicit run time limit. The orange line at the top of of each plot represents the optimal score. We can see that even for very short run time limits we get results for every

[2] Due to the cut ratio, we did not run this experiment for the real-world data sets, as it would have taken too much time.

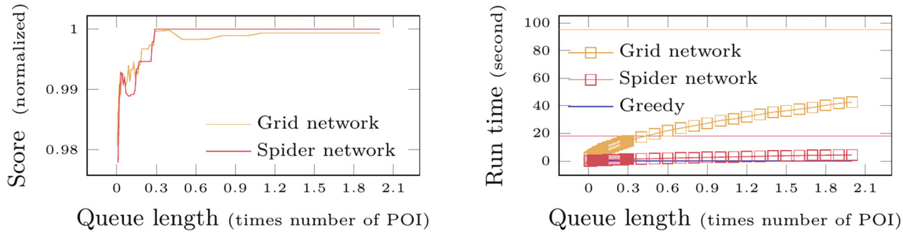



**Fig. 5.** Limiting the queue length

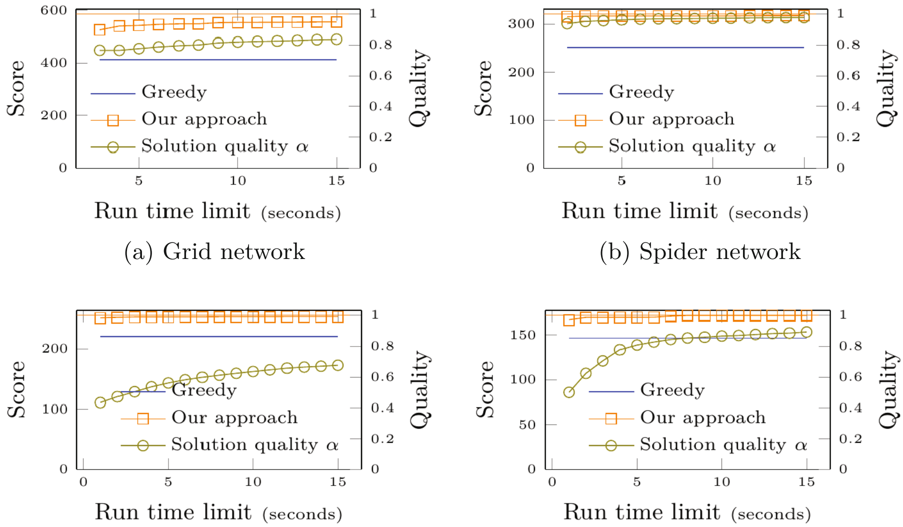


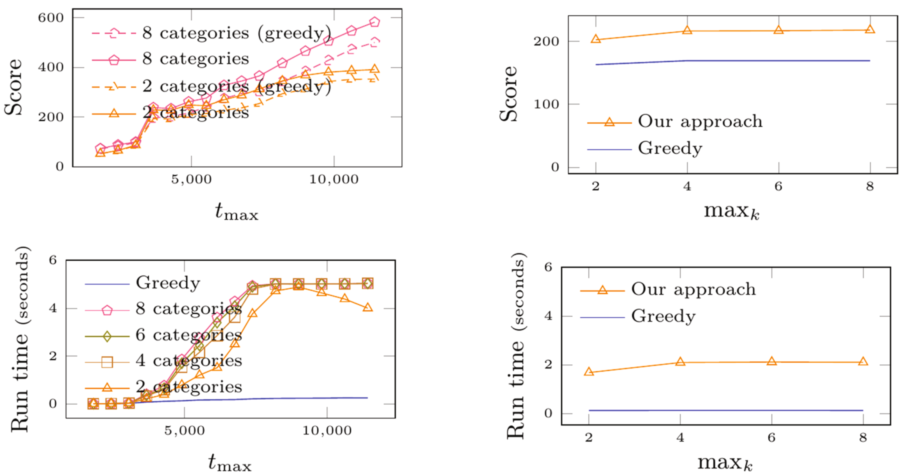

**Fig. 6.** Limiting the run time (Color figure online)

data set that are very close to the optimal score. What this means is that our algorithm finds very good solutions early in the search process and then, if we let it continue, spends the remaining time basically verifying that there are no significantly better solutions. We also clearly outperform the greedy heuristic in terms of score (the greedy algorithm takes between 0.2 and 1.5 s to find a solution).

When limiting the run time, we cannot guarantee the quality of the attained score a priori. Nevertheless, when the algorithm stops running, together with the score it returns the factor $\alpha$, telling us the ratio of the optimal score we have reached at least (for details, see Sect. 6.2). We have plotted the lower bound $\alpha$ in Fig. 6, please note that for this measure the scale on the right-hand side of the y-axis is used. Looking at Fig. 6, we see that the results for $\alpha$ for two of the data sets, the spider network in Fig. 6(b) and the Bolzano network in Fig. 6(c) are somewhat different, which comes as a surprise, as we mentioned that the Bolzano network shares some of the properties of a spider network in Sect. 7.2. However, there are other effects at work here, too. For the Bolzano network the larger number of categories is responsible for the lower guarantee. This makes it more difficult to compute $\texttt{extra}(\mathcal{I})$ accurately, making it more difficult to discard potential unexpanded solutions. (We go into more details about the number of categories in the following section.) We would like to point out that the found solutions are close to the optimal for every data set, only the lower bounds for the score are affected differently.

In summary, we can obtain a solution and its quality from our algorithm at any time, enabling us to balance the quality of the solution and the run time on the fly.

**Impact of Categories.** For the final parameter, we illustrate the impact of categories on the run time. While our approximation algorithms scale well in terms of the itinerary length ($t_{\max}$), as we will see in Sect. 7.3, this is not independent of the number of categories. Figure 7(a), left column, shows the effects of increasing the number of categories. By putting a limit on the run time, we can keep the execution time of the algorithm low, while still generating very good solutions. In the upper diagram, we only show the plots for two and eight categories to keep it uncluttered. In the lower diagram, the run time for two categories actually goes down for longer itineraries. At the end of visiting two POIs of two categories each, we actually start having more and more spare time, which makes the problem easier to solve.




(a) Varying number of categories (b) Varying $\max_k$ constraints

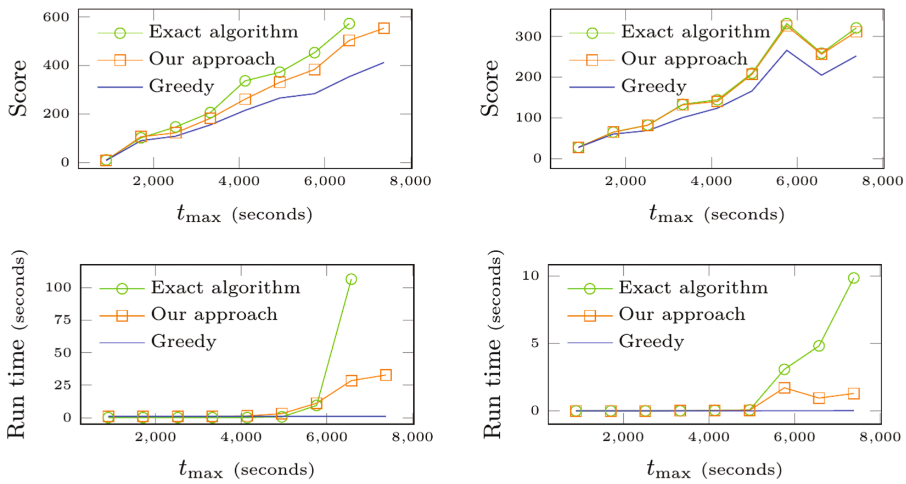
**Fig. 7.** Impact of categories

Figure 7(b), right column, shows the effect of increasing the maximum constraint of categories. Again, we quickly generate good solutions in a scalable way, due to the explicit limit on the run time.

### 7.3 Comparison with Competitors

Here we compare our algorithm to the state-of-the-art. We distinguish two different cases: (a) the artificial networks (grid and spider) and (b) the real-world data sets (Bolzano and San Francisco). For the real-world data sets we also run more realistic queries with a larger time constraint $t_{\max}$ and for that reason do not compare it to the optimal algorithm (as its run time explodes for long itineraries).

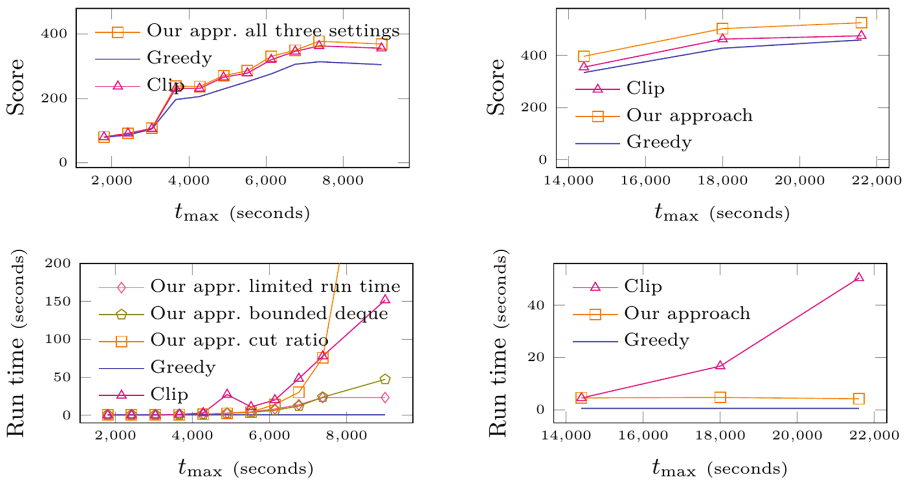
**Grid and Spider Networks.** Figure 8 illustrates the results of running our algorithm with a cut factor of 1.2 against the optimal algorithm (the left-hand column, labeled (a), shows the results for the grid network, the right-hand column, labeled (b), shows those for the spider network). In terms of the score our approach outperforms the greedy approach and comes close to the optimal solution, while at the same time having a much better run-time performance than the optimal algorithm. The final measurements for large values of $t_{\max}$ are missing, since we aborted runs taking longer than ten minutes to complete.




(a) Grid network (b) Spider network

**Fig. 8.** Comparison with the exact algorithm

**Real-World Data Sets.** We now move to the real-world data sets. Figure 9(a), found in the left-hand column, compares three different variants of our algorithm (cut factor, bounded queue, and run time limit) with the state-of-the-art algorithm for orienteering with categories, CLIP on a map of Bolzano. In terms of the score (upper part of Fig. 9(a)) the three variants of our algorithm are almost identical: merely one measurement resulted in a difference two digits after the decimal point (thus we only depict one curve). Our algorithm outperforms both, CLIP and the greedy heuristic, the latter by a large margin. When we look at the run time (lower part of Fig. 9(a)), we see huge differences, though. The only competitive algorithms are the ones limiting the run time in some form, either directly or by limiting the queue length.




(a) Bolzano network (b) San Francisco network

**Fig. 9.** Comparison with CLIP

In the final experiment we show the full strength of our approach for running queries with a large time constraint $t_{\max}$. Here we run a blended version of our algorithm, combining a run time limit of five seconds with a queue length of half the number of total POIs in the graph. At first glance, combining a run time limit with a fixed queue length seems redundant. However, from a certain run time limit onwards, the queue length may become quite large and we may want to restrict it.

Figure 9(b) shows results for the San Francisco data set, comparing CLIP to our blended variant. In terms of the score, our approach outperforms both, CLIP and the greedy heuristic. We even get a better score than CLIP with a shorter run time, demonstrating that our technique scales much better than CLIP. In fact, as we were running the experiments on weaker hardware compared to the findings in [2], we could not replicate the ten-hour itineraries for CLIP shown in [2] in reasonable time. This clearly shows that our algorithm is much more suitable for deployment on mobile devices, which have limited computational resources. We also illustrate that limiting the queue length (to a certain extent) does not have adverse effects on the run-time-limited variant.

## 8 Conclusion and Future Work

We have developed an effective and efficient approximation algorithm for solving the orienteering problem with category constraints (OPMPC) by applying a best-first strategy, blending it with greedy search, and then limiting its run time. One major advantage of our technique over CLIP, the state-of-the-art approach for OPMPC, is the fact that our technique is an anytime algorithm, which immediately generates solutions with quality guarantees, as it keeps track of potential scores. Consequently, we can run our algorithm for a fixed time or until a certain quality level has been reached, whichever comes first.

Nevertheless, we still see some room for improvement. Profiling our algorithm, we noticed that we spend a considerable amount of time (about 40%) calculating the potential score. If we were able to do this more efficiently, maybe by parallelizing the task, we would be able to create an even more efficient algorithm. For longer itineraries it may also be interesting to move to more efficient techniques for computing itineraries from sets of POIs. On a more general level, our approach could also be viable for other orienteering variants, such as the team orienteering problem or orienteering with time windows.

## References

1. Blum, A., Chawla, S., Karger, D.R., Lane, T., Meyerson, A., Minkoff, M.: Approximation algorithms for orienteering and discounted-reward TSP. SIAM J. Comput. **37**(2), 653–670 (2007)
2. Bolzoni, P., Helmer, S., Wellenzohn, K., Gamper, J., Andritsos, P.: Efficient itinerary planning with category constraints. In: SIGSPATIAL/GIS 2014, Dallas, Texas, pp. 203–212 (2014)
3. Chekuri, C., Korula, N., Pál, M.: Improved algorithms for orienteering and related problems. In: SODA 2008, pp. 661–670 (2008)
4. Chekuri, C., Pál, M.: A recursive greedy algorithm for walks in directed graphs. In: FOCS 2005, pp. 245–253 (2005)
5. Gavalas, D., Konstantopoulos, C., Mastakas, K., Pantziou, G.: A survey on algorithmic approaches for solving tourist trip design problems. J. Heuristics **20**(3), 291–328 (2014)
6. Gendreau, M., Laporte, G., Semet, F.: A branch-and-cut algorithm for the undirected selective traveling salesman problem. Networks **32**(4), 263–273 (1998)
7. Keller, C.: Algorithms to solve the orienteering problem: a comparison. Eur. J. OR **41**, 224–231 (1989)
8. Liang, Y.-C., Kulturel-Konak, S., Smith, A.: Meta heuristics for the orienteering problem. In: CEC 2002, pp. 384–389 (2002)
9. Lu, E.H.-C., Lin, C.-Y., Tseng, V.S.: Trip-mine: an efficient trip planning approach with travel time constraints. In: MDM 2011, pp. 152–161 (2011)

10. Ramesh, R., Yoon, Y.-S., Karwan, M.H.: An optimal algorithm for the orienteering tour problem. Inf. J. Comput. **4**(2), 155–165 (1992)
11. Rice, M.N., Tsotras, V.J.: Parameterized algorithms for generalized traveling salesman problems in road networks. In: SIGSPATIAL/GIS 2013, Orlando, Florida, pp. 114–123 (2013)
12. Righini, G., Salani, M.: Decremental state space relaxation strategies and initialization heuristics for solving the orienteering problem with time windows with dynamic programming. Comput. OR **36**(4), 1191–1203 (2009)
13. Sevkli, Z., Sevilgen, F.E.: Variable neighborhood search for the orienteering problem. In: Levi, A., Savaş, E., Yenigün, H., Balcısoy, S., Saygın, Y. (eds.) ISCIS 2006. LNCS, vol. 4263, pp. 134–143. Springer, Heidelberg (2006). doi:10.1007/11902140_16
14. Singh, A., Krause, A., Guestrin, C., Kaiser, W.J., Batalin, M.A.: Efficient planning of informative paths for multiple robots. In: IJCAI 2007, pp. 2204–2211 (2007)
15. Tasgetiren, F., Smith, A.: A genetic algorithm for the orienteering problem. In: IEEE Congress on Evolutionary Computation (2000)
16. Tsiligrides, T.A.: Heuristic methods applied to orienteering. J. Oper. Res. Soc. **35**(9), 797–809 (1984)
17. Wang, Q., Sun, X., Golden, B.L., Jia, J.: Using artificial neural networks to solve the orienteering problem. Ann. OR **61**, 111–120 (1995)

http://www.springer.com/978-3-319-64366-3

Advances in Spatial and Temporal Databases
15th International Symposium, SSTD 2017, Arlington, VA, USA, August 21 – 23, 2017, Proceedings
Gertz, M.; Renz, M.; Zhou, X.; Hoel, E.; Ku, W.-S.; Voisard, A.; Zhang, C.; Chen, H.; Tang, L.; Huang, Y.; LU, C.-T.; Ravada, S. (Eds.)
2017, XIV, 454 p. 206 illus., Softcover
ISBN: 978-3-319-64366-3