Mining tours and paths in activity networks

Sofia Maria Nikolakaki
Boston University
smnikol@bu.edu

Alina Ene
Boston University
aene@bu.edu

Charalampous Mavroforakis
Boston University
cmav@bu.edu

Evimaria Terzi
Boston University
evimaria@bu.edu

ABSTRACT
The proliferation of online social networks and the spread of smart mobile devices enable the collection of information related to a multitude of users’ activities. These networks, where every node is associated with a type of action and a frequency, are usually referred to as activity networks. Examples of such networks include city road networks, where the nodes are intersections and the edges are road segments. Each node is associated with a number of geolocated actions that users of an online platform took in its vicinity. In these networks, we define a prize-collecting subgraph to be a connected set of nodes, which is compact, i.e., the nodes are close to each other, and exhibits high levels of activity.

The $k$-PCSUBGRAPHS problem we address in this paper is defined as follows: given an activity network and an integer $k$, identify $k$ non-overlapping and connected subgraphs of the network such that the nodes of each subgraph are close to each other, and the number of actions that they are associated with is high. Here, we define and study two new variants of the $k$-PCSUBGRAPHS problem, where the subgraphs of interest are tours and paths. Since both these problems are NP-hard, we provide approximate algorithms that solve them in time nearly linear to the number of edges. In our experiments, we use real activity networks obtained by combining actual city road networks and projecting on them user activity from Twitter and Flickr. Our experimental results demonstrate both the efficiency and the practical utility of our methods.

ACM Reference format:

1 INTRODUCTION
Recently, there has been a significant increase in the level of user engagement with online platforms like social networks or content-sharing websites. Simultaneous to this, and in part responsible, is the broad availability of smart devices. This phenomenon has given rise to a new breed of networks, the activity networks. The nodes in activity networks are enriched with extra information resulting from users’ engagement (typically a list of action types and the corresponding counts). For example, in city road networks the nodes are intersections and edges are road segments. These can be enriched with the geolocated posts, photos, or reviews that are being published in their vicinity. Another example is that of a social network; each user (node) can now be characterized by her aggregate activity in many disparate platforms. In these settings, we define a prize-collecting subgraph to be a connected set of nodes in the network, which is compact, i.e., the nodes are close to each other, and exhibits high levels of activity.

The $k$-PCSUBGRAPHS problem we address in this paper is defined as follows: given an activity network $G = (V, E)$, the edge cost and the node prize values, and an integer $k$, identify $k$ non-overlapping and connected subgraphs of $G$, such that each subgraph consists of nodes that are highly active and close to each other. The case where the subgraphs are required to be trees has been studied extensively in the past both for $k = 1$, known as the Prize Collecting Steiner Tree (PCST) [10, 11, 13] problem, and $k > 1$, known as the Prize Collecting Steiner Forest (k-PCSF) problem [12]. In this paper, we introduce and study two new variants of the $k$-PCSUBGRAPHS problem, where we require the subgraphs to be (1) tours, and (2) paths. We refer to these problems as the $k$-PCTOURS and the $k$-PCPATHS problem respectively.

To the best of our knowledge, we are the first to propose the $k$-PCTOURS and the $k$-PCPATHS problems. For these, only the variant of a single tour or path over points in a metric space has been studied in the past [1, 2, 4]. Here, we extend the definition for $k$ tours and paths to go beyond metric spaces and, based on the connection between the $k$-PCTOURS and the $k$-PCPATHS problems with the $k$-PCSF, we design efficient algorithms for solving them.

Our algorithms for both problems use an existing, nearly-linear time algorithm for $k$-PCSF [12] as a building block. This algorithm is linear in the number of edges and logarithmic to the number of nodes. As a result, and given that the number of subgraphs we seek to find is much smaller than the number of nodes, our algorithms also run in time nearly-linear to the number of edges in the network.

The $k$-PCTOURS and the $k$-PCPATHS problems arise in many applications, including city activity networks, which is also the main focus of this paper. In this application, tours and paths correspond to routes that users take as they travel in the city. For instance, as a tourist, it makes sense to know $k$ paths that are short enough and along which people have captured many beautiful pictures.

For our experiments, we compile real activity networks; we start with the road networks of three major cities in the US and project on them geolocated posts and photos from Twitter and Flickr respectively. We demonstrate the efficacy and the efficiency
of our algorithms in optimizing our objectives, and present some of the notable subgraphs that are uncovered.

Our paper is closely related to the work of Rozenshtein et al. [19], where they also analyze activity networks. The key difference from our work is that their aim is to find a single tree, and they do not consider subgraphs in the forms of tours or paths.

Contributions: The contributions of the paper can be summarized as follows:

- We propose an algorithmic framework for identifying multiple, high-activity subgraphs in activity networks. Moreover, we identify two new problems in this framework, namely the k-PCTours and the k-PCPaths problems.
- We design approximation algorithms as well as heuristic algorithms for both problems. Our algorithms are motivated by the connection between our problems and k-PCSF. In fact, we extend the algorithm for k-PCSF with additional routines, which often involve linear-time algorithms for doing dynamic-programming on trees. As a result, for the cases where the number of subgraphs is small, we get an almost linear-time algorithm for both problems.
- In our extensive experiments with data that record users’ social network activity in different US cities, we demonstrate the efficiency and the efficacy of our methods at discovering interesting routes that people take in their cities.

Roadmap: The rest of the paper is organized as follows: In Section 2 we describe our general problem definition and the two problem variants we consider. In section 3 we describe the algorithms we design to solve our problems. We experimentally evaluate our solutions in Section 4 and discuss related work in Section 5. We conclude the paper in Section 6.

2 PROBLEM DEFINITIONS

In this section, we provide the definitions and the notational conventions we use throughout the paper. We also formally define our problems and discuss their complexity.

2.1 Preliminaries

Throughout the paper, we assume that our input consists of a weighted undirected graph $G = (V, E, c, \pi)$, where $V$ is a set of $n$ nodes and $E$ is a set of $m$ edges. Each edge $e \in E$ is associated with a non-negative edge cost $c(e) \geq 0$, and each node $v \in V$ has a non-negative vertex prize $\pi(v) \geq 0$. One can think of the cost of an edge as a measure of the distance between the nodes it connects. The prize of a vertex is the intensity level of the activity observed in the corresponding vertex, e.g., the number of tweets, the number of photo uploads, etc. We refer to this graph as the activity network.

We also use the following conventions: Given $S \subseteq V$, we use $G[S]$ to denote the subgraph of $G$ induced by $S$. This is the subgraph with nodes $S$ and edges $E[S] \subseteq E$, such that the endpoints of every edge in $E[S]$ are two nodes in $S$. Thus, $G[S] = (S, E[S], c, \pi)$. Furthermore, we refer to the prize of $S$ as the sum of the prizes of the nodes in $S$. That is, $\pi(S) = \sum_{v \in S} \pi(v)$. Also, we use $\bar{S}$ to denote the complement of $S$ with respect to $V$, i.e., $\bar{S} = V \setminus S$. Finally, for $E' \subseteq E$, we refer to the sum of the costs of the edges in $E'$ as the cost of the set $E'$. That is, $c(E') = \sum_{e \in E'} c(e)$.

Throughout the paper we use the following definitions for tours and paths on a graph. A tour is a node sequence in which any two consecutive nodes are connected with an edge. A tour is closed, i.e., the last node is the same as the first one. A path is a sequence of edges that connect a sequence of nodes. A node can only appear in the path at most once. This definition is equivalent to the Hamiltonian path.

2.2 The k-PCSubgraphs problem

The general problem that we address in this paper can be defined as follows: given a graph $G = (V, E, c, \pi)$ and a positive integer $k$, identify $k$ connected and non-overlapping subgraphs of $G$, each of which includes nodes with high prize and low-cost edges. This is formally expressed as follows:

**Problem 1 (k-PCSubgraphs).** Given a graph $G = (V, E, c, \pi)$, a positive integer $k$ and $\lambda \in \mathbb{R}^+$, find a set of subgraphs $\mathcal{G} = \{G_1, \ldots, G_k\}$ with $G_i = (V_{G_i}, E_{G_i})$, such that

$$\sum_{e \in \bigcup_{i=1}^{k} E_{G_i}} c(e) + \lambda \sum_{v \in V \setminus \bigcup_{i=1}^{k} V_{G_i}} \pi(v)$$

is minimized.

In the above definition, $\lambda$ is a normalization coefficient that expresses our bias between the prizes and the costs. One can also think of $\lambda$ as being used to convert two quantities, which are defined in different scales, into the same units.

We study the following two instantiations of the k-PCSubgraphs problem: when the subgraphs are required to be tours, then we refer to the corresponding instantiation of Problem 1 as the k-PCTours problem. When the subgraphs are required to be paths, then we refer to the corresponding instantiation of Problem 1 as the k-PCPaths problem.

The k-PCTours problem is NP-hard, as it is NP-hard already when $k = 1$ [4]. As a result, we can only hope to approximate this problem in polynomial time. The k-PCPaths problem is also NP-hard [1, 4] and although it can be approximated for metric spaces it is inapproximable for general graphs [1]. Since we focus on general graphs, the algorithms we design for this problem cannot have bounded approximation factors.

2.3 The k-PCSF problem

A well known variant of Problem 1, restricts the subgraphs to be trees. When $k = 1$, this problem is known as the Prize Collecting Steiner Tree (PCST) problem. When $k > 1$, we will refer to the problem as the k-PCSF problem. The PCST problem admits a factor-2 approximation [10, 11, 13]. The most recent of these algorithms [11] runs in time $O(dm \log n)$, where $d$ is the bits of precision of the cost and prize values. In a later paper, Hedge et al. [12] show that a small modification of the initial algorithm they provided in [11] gives a factor-2 approximation for the k-PCSF problem. We refer to this algorithm as the kTrees algorithm and we use it as a building block for the rest of the paper.

3 ALGORITHMS

In this section, we describe our algorithms for the k-PCTours and the k-PCPaths problems. Since all these problems are NP-hard our algorithms only aim to provide approximate solutions.
Algorithm 1 The $k$Tour's algorithm.

**Input:** Graph $G = (V, E, c, \pi)$, $k \in \mathbb{Z}^+$, $\lambda \in \mathbb{R}^+$.  
**Output:** A set of $k$ tours $R$.

1. $\mathcal{F} \leftarrow \text{kTrees}(G, c, \pi, k, \lambda) \quad \triangleright$ Solves $k$-PCSF on $G$.  
2. $R \leftarrow \emptyset$  
3. for all trees $T_i$ in $\mathcal{F}$ do  
4. $G_i \leftarrow$ induced subgraph $G[V_{T_i}]$  
5. $\tau \leftarrow \text{FindTour}(G_i)$  
6. Add $\tau$ to $R$  
7. end for  
8. return $R$

Algorithm 2 The $\text{FindTour}$ algorithm.

**Input:** Graph $G = (V, E, c, \pi)$.

**Output:** A tour $\tau$ of $G$.

1. $G' \leftarrow \text{DoubleEdges}(G)$  
2. $\tau \leftarrow \text{CreateTour}(G') \quad \triangleright$ Finds Eulerian tour  
3. return $\tau$

3.1 Solving the $k$-PCTOURS problem

Here, we introduce a 4-approximation algorithm for the solution of the $k$-PCTOURS problem. We refer to this algorithm as $k$Tour's and present its outline in Algorithm 1.

$k$Tour's operates in two steps; first, it calls kTrees in order to find a set $\mathcal{F}$ of $k$ trees in $G$. Then, for each of these trees $T_i \in \mathcal{F}$, it isolates the subgraph of $G$ that includes all edges induced by the nodes of $T_i$, i.e., $G[V_{T_i}]$, and finds a tour that visits all of $T_i$'s nodes.

The method that designs a tour on a connected graph is summarized in Algorithm 2 and is called $\text{FindTour}$. It works by first transforming the input graph into an Eulerian one by doubling all its edges. Notice that, this allows visiting a node (forth) and returning (back), by using the original edge to go forth and its copy to come back. Whenever a tour picks both an edge and its copy, the cost of this specific edge will contribute twice to the total cost of the tour. Now the graph $G[V_{T_i}]$ has an Eulerian tour. Next, $\text{CreateTour}$ draws this tour $\tau$ by picking an arbitrary node in the Eulerian graph and executing a depth-first search while reporting the visited edges. The returned tour $\tau$ visits some of the nodes multiple times. The edges however are visited only once due to the edge doubling of the previous step. The running time of $k$Tour's is $O(dm \log n + m + n)$.

Our result for the approximation factor of $k$Tour's is based on the following lemma.

**Lemma 3.1.** Given a graph $G = (V, E, c, \pi)$, an integer $k > 0$ and $\lambda \in \mathbb{R}^+$, the cost of the optimal solution of $k$-PCSF is less than or equal to the cost of the optimal solution of $k$-PCTOURS. That is,

$$\text{Opt}_{k-\text{PCTOURS}} \leq \text{Opt}_{k-\text{PCTOURS}}$$

**Proof.** Let us denote with $R^* = \{\tau_1, \ldots, \tau_k\}$ the optimal solution for $k$-PCTOURS on $G$. Now, let us generate a solution $\mathcal{F} = \{T_1, \ldots, T_k\}$ for $k$-PCSF on $G$ in the following way: for each connected component $T_i \in R^*$, let $T_i$ be the minimum spanning tree on $T_i$. Then, $\pi(T_i) = \pi(\tau_i)$, while $c(T_i) < c(\tau_i)$. The cost of the optimal solution $\mathcal{F}^*$ for $k$-PCSF is $\text{cost}(\mathcal{F}^*) \leq \text{cost}(\mathcal{F}) < \text{cost}(R)$, where $\text{cost}()$ corresponds to Equation 1.

Given the above lemma we have the following result:

**Theorem 3.2.** $k$Tour's is a 4-approximation algorithm for the $k$-Tours problem.

**Proof.** Given a graph $G = (V, E, c, \pi)$, an integer $k > 0$ and $\lambda \in \mathbb{R}^+$, let $\mathcal{F} = \{T_1, \ldots, T_k\}$ be the solution returned by the kTrees algorithm, and $R = \{\tau_1, \ldots, \tau_k\}$ be the tours returned by kTour's on $G$. Notice that, by construction, $\tau_i$ is a tour visiting $V_{T_i}$. Also, $c(\tau_i) \leq 2c(T_i)$, for all $T_i \in \mathcal{F}$, because, in the worst case, $\tau_i$ traverses each edge in $E_{T_i}$ twice. Furthermore, $\pi(\tau_i) = \pi(T_i)$, because $\tau_i$ visits all the nodes of $T_i$. Consequently,

$$\text{cost}(R) = \lambda \pi(V) + \sum_{i=1}^{k} c(\tau_i) - \lambda \pi(\tau_i)$$

$$\leq \lambda \pi(V) + \sum_{i=1}^{k} 2c(T_i) - \lambda \pi(T_i)$$

$$\leq 2 \left( \lambda \pi(V) + \sum_{i=1}^{k} c(T_i) - \lambda \pi(T_i) \right)$$

$$\leq 4 \left( \lambda \pi(V) + \sum_{i=1}^{k} c(T_i) - \lambda \pi(T_i) \right)$$

$$= 4 \text{Opt}_{k-\text{PCSF}}$$

$$\leq 4 \text{Opt}_{k-\text{PCTOURS}}$$

where $\mathcal{F}^* = \{T_1^*, \ldots, T_k^*\}$ is the optimal solution for the $k$-Trees on $G$. The last step above is due to Lemma 3.1. Notice also that Equation (5) holds because $\pi(V) - \sum_{i=1}^{k} \pi(T_i) \geq 0$, and Equation (6) is because $k$Trees is a 2-approximation algorithm to $k$-PCSF [12].

3.2 Solving the $k$-PCPATHS problem

Our algorithm for the $k$-PCPATHS problem, kPaths, can be broken down into two main components. It starts by using kTrees to build a collection $\mathcal{F}$ of trees. Then, it executes $\text{FindPaths}$ (Algorithm 4) on $\mathcal{F}$ to select a set $\mathcal{P}$ of $k$ paths. The total complexity of kPaths is $O(dm \log n + kn + k^2)$. Before we discuss the details of this algorithm however, let us first present its building blocks.

**Solving k-PCPATHS on a binary tree:** Let's assume that the kTrees algorithm returns a single binary tree $T = (V_T, E_T)$. We will later discuss how to relax the restrictions both on the number as well as the type of the trees. Here, we show how one can find the $k$ best paths in $T$, i.e., the $k$ paths which together minimize the cost presented in Equation (1). To this end, we develop a dynamic-programming algorithm that runs in time $O(kn + k^2)$.

As a first step, let us arbitrarily pick a node $r \in V_T$ and let us use it as the root for $T$. We will use the notation $T_{v, r}$ to refer to the subtree of $T$ that is rooted at a particular node $v \in V_T$. Let us also define the set $\text{succ}(v) \subseteq V_T$ to be the set of nodes that are the immediate successors of $v$ in $T$. When $v$ has only two children, we will denote them as $v_l$ and $v_r$.

In general, paths in trees can be partitioned into two categories; those which use only ancestor-to-descendant edges and those that
The intuition of our algorithm is to iteratively compute the cost of the best vertical and the latter vertical paths, and we illustrate examples of such paths, using the tree in Figure 1. Specifically, for each node \( v \) in the tree, we need to pick how to split \( k \) between the left and right child of \( v \) respectively.

Next, \( H(v, k) \) considers \( k \) paths in the subtree \( T_v \), requiring that \( v \) participates in exactly one horizontal path. As a result, we need to pick how to split \( k \) between the left and right child of \( v \). Using \( D(v, k_1) \) and \( D(v, k_r) \), and constraining both \( k_1 \) and \( k_r \) to be positive, we make sure that both \( v_1 \) and \( v_r \) participate in exactly one vertical path. Otherwise, we would not be able to draw a horizontal path through \( v \).

\[
H(v, k) = \min_{k_1 + k_r = k} \left( D(v, k_1) + D(v, k_r) + c(v, v_1) + c(v, v_r) \right) 
\]

We use \( k_1 \) and \( k_r \) to denote the budget that we allocate to the left and right child of \( v \) respectively.

When computing \( B(v, k) \), we restrict \( v \) from participating in any of the \( k \) paths in the subtree \( T_v \).

\[
B(v, k) = \pi(v) + \min_{k_1 + k_r = k} \left( D(v, k_1) + D(v, k_r) \right) 
\]

Finally, \( L(v, k) \) decides between including \( v \) in one of the paths or not, independently of the type of the path.

\[
L(v, k) = \min \begin{cases} 
D(v, k) \\
H(v, k) \\
B(v, k) 
\end{cases} 
\]

We will refer to the algorithm that computes these recursive formulas for all the nodes of \( T \) as \( \text{FindPathsOnTree} \). To find the cost of the optimal selection of \( k \) paths on \( T \), we execute \( \text{FindPathsOnTree} \) and report \( L(r, k) \). We can retrieve the solution that optimizes this cost in \( O(k^2 \pi) \) time.

**Lemma 3.3.** \( \text{FindPathsOnTree} \) finds the optimal solution for the \( k\text{-PCPath} \) problem, when the input graph is a binary tree.

**Solving \( k\text{-PCPath} \) on a general tree:** We now show how to enable \( \text{FindPathsOnTree} \) to work for general trees. One could generalize the previous equations to work for any number of children per node. However, if this is implemented naively, it would result in a time complexity of \( O(\pi(k^2) \pi) \), which is not practical for large values of \( k \).
in an exponential running time. Instead, we choose to transform the tree to its binary equivalent; a relatively straightforward practice also seen in the works [8, 15]. We omit the description of this procedure due to space constraints. The new tree has at most twice as many nodes as the original tree and thus the runtime complexity of $\text{FindPathsOnTree}$ remains $O(k^2 n)$.

Notice that, we need to slightly modify our algorithm to ensure that, when run on the binarized version $B$ of an arbitrary tree $T$, $\text{FindPathsOnTree}$ will return the optimal valid path on $T$. For space reasons, we omit the details of these modifications.

**Solving $k$-PCPaths on a graph:** So far, we have assumed that there is only a single tree on which we will select our $k$ paths. This is equivalent to executing $k$Paths for $t = 1$. We consider this to be a distinct method, $k$Paths($t = 1$), whose purpose becomes clear in Section 4. However, the kTrees algorithm may also return a set $F$ of trees with $t = |F| \geq 1$, thus allowing us to be more flexible. In fact, we can ask kTrees to retrieve exactly $t = k$ trees, because, in the extreme case, we will assign one path per tree. Here, we show how to optimally split our budget of $k$ paths between the trees in $F$, so that we minimize the total cost of the solution. This can be achieved by another dynamic-programming recursion.

After running $\text{FindPathsOnTree}$ on each tree $T_i \in F$ for budget $k$, we know the optimal cost $L(T_i, k)$ of assigning $k$ paths at $T_i$, for all $0 \leq k \leq k$. The optimal allocation on $F$ can now be calculated using dynamic programming: let $T_1, \ldots, T_k$ be a random but fixed ordering on the trees in $F$, and $\text{Opt}(\ell, b)$ be the optimal allocation of $b \leq k$ paths on the first $\ell$ trees $T_1, \ldots, T_\ell$. Then $\text{Opt}(k, k)$ is the optimal solution to our budget assignment problem. This can be recursively computed as follows:

$$\text{Opt}(\ell, b) = \min_{0 \leq b \leq b} \text{Opt}(\ell - 1, b - b') + L(T_\ell, b') \quad (13)$$

We can compute $\text{Opt}(k, k)$ in time $O(k^2)$. The initial conditions of this dynamic program are the following: $\text{Opt}(T_i, 0) = \pi(T_i)$ and $\text{Opt}(T_i, i) = L(T_i, i)$, for $i = 1, \ldots, k$. We call this dynamic programming algorithm SplitBudget.

**Running time complexity of kPaths:** Overall, the complexity of kPaths can be split into two main parts; initially it runs kTrees to produce a set $F$ of trees, which takes $O(dm \log n)$. Then, it runs $\text{FindPaths}$ to find the paths on the trees. In $\text{FindPaths}$ for each $T_i \in F$ there is a call to $\text{Binaryize}$, which takes $O(n)$, and a call to $\text{FindPathsOnTree}$, which takes $O(kn)$. The complexity of repeating this for all the trees is $O(k^2 n)$. Finally, SplitBudget runs in $O(k^2)$. Therefore, the running time complexity of kPaths is $O(dm \log n + k^2 n + k^2)$.

We can reduce the complexity of kPaths to $O(dm \log n + kn + k^2)$ by making an interesting observation about $\text{FindPaths}$. Consider the computation of $D(v, i)$, see Equation (9). We note that, for a fixed node $v$, we can fill in all the entries $D(v, i)$ for $i$ ranging from 1 to $k$ in $O(k)$ time using the following observation. Consider the entry $D(v, i)$ and let $(l'_j, i'_j)$ be an optimal split for $i$. We claim that there is an optimal split $(j, f'_j)$ for $j = i + 1$ satisfying $f'_j = i'_j$ or $f'_j = i'_j + 1$. In other words, the better of the two splits $(l'_j + 1, i'_j)$, $(l'_j, i'_j + 1)$ is an optimal split for $i + 1$, so when going from $i$ to $i + 1$ we add a path, either to the left or to the right child, but without removing any other paths. This can be extended to all the values used in the DP, whose computation requires considering pairs of the form $(i_j, i_i)$. Thus, the new running time complexity of kPaths becomes $O(dm \log n + kn + k^2)$. Intuitively, let us assume that, without loss of generality, instead of only adding a path to the left child, the algorithm also removes an existing path $P_1$ and adds a new one $P_2$ to the solution. We claim that this solution is suboptimal. Indeed, the extra path $P_2$ increases the total cost more than $P_1$, or else it would have been picked by the algorithm at a previous step. As a result, no paths are removed when moving from a solution with $i$ paths to one with $i + 1$.

### 4 EXPERIMENTS

The purpose of this section is to apply our algorithms on real datasets, and explore both their performance and their efficiency. Specifically, (i) we evaluate the performance of our three methods, kTrees, kPaths($t = 1$) and kPaths, compared to a set of baseline methods for different values of $k$, (ii) we present a break-down of the runtime of our algorithms, and (iii) we provide a practical comparison using visual representations, thus showing their potential impact in the real world. For the formal definitions of these algorithms we refer the reader to Sections 3.1 and 3.2.

#### 4.1 Datasets

We generate six real-world activity networks by combining the road network of three cities with geo-located user activity on two online social platforms. Specifically, we focus on three major cities in the US (Boston, San Francisco and Austin), and two online platforms (Twitter and Flickr). Table 1 presents some statistics about the size and the activity levels in our datasets.

**Table 1:** A summary of the dataset statistics.

<table>
<thead>
<tr>
<th>City</th>
<th>#nodes</th>
<th>#edges</th>
<th>#tweets</th>
<th>#photos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>10 101</td>
<td>15 435</td>
<td>222 474</td>
<td>124 560</td>
</tr>
<tr>
<td>San Francisco</td>
<td>14 361</td>
<td>22 055</td>
<td>448 688</td>
<td>282 263</td>
</tr>
<tr>
<td>Austin</td>
<td>53 772</td>
<td>77 135</td>
<td>243 811</td>
<td>102 446</td>
</tr>
</tbody>
</table>

**Road network:** For each of the three cities of interest, we use OpenStreetMap to retrieve the corresponding city road network. The nodes in these networks represent road intersections, while the edges are road segments. Each edge has a positive weight that represents its length in kilometers. Given that we only select nodes within the city limits, these networks might have more than one connected component, e.g., a neighborhood might be connected to other neighborhoods through a road that exceeds the city’s limits. In this case, we only keep the largest connected component. Moreover, because of the nature of our application, we filter out the highways.

**User activity:** We use geo-located posts from Twitter (between October 2015 and January 2017) and geo-tagged photos from Flickr to capture the levels of activity at each node in our road networks. Specifically, we assign a post or a photo, respectively, to the node whose location is closest to the item. The node’s prize is then the

1 Corresponds to running the kPaths algorithm for a single tree, produced by kTrees for $k = 1$

2 http://wiki.openstreetmap.org/wiki

3 https://dev.twitter.com/rest/public

4 https://webscope.sandbox.yahoo.com
In order to evaluate the performance of the algorithms introduced in Section 3, we propose a variety of baseline methods that are summarized in Table 2. The primary difference between our approaches and the baseline algorithms lies in the method used to produce the trees required for the extraction of tours and paths. Note that the second step of all the baseline algorithms utilizes our own kTrees and FindPaths algorithms. As a result, we do not expect those algorithms to do considerably worse than ours, even though kTrees is a constant factor approximation algorithm, when trees are produced by kTrees, and FindPaths is optimal for a set of given trees. The baselines we consider are the greedy-based approach, Greedy, and the well-known density-based clustering algorithm, DBSCAN [7].

Greedy, is based on kTrees [11]. It iteratively extracts one tree from the input graph $G$ using kTrees for $k = 1$, adds it to the solution and then removes its nodes from $G$. It repeats this process until the solution includes $k$ trees. DBSCAN locates regions of high density separated from one another by regions of low density by using a nearest neighbor approach. The algorithm takes as input a set of points in space and the parameters MinPts and Eps, and returns an arbitrary number of dense clusters. To identify the parameter values for which the solution retrieves exactly $k$ clusters we perform a greedy search. Given that the output clusters are not necessarily connected, we impose connectivity by finding for each cluster the induced subgraphs, computing the minimum spanning tree for each of these subgraphs and selecting the cluster’s representative to be the subgraph that improves the objective value. Following, we describe each of our proposed baselines.

**Greedy, DBSCAN + kTrees**: These two algorithms are simple extensions of Greedy and DBSCAN, respectively, with the addition of applying the kTrees algorithm (Algorithm 1, lines 2-8) to their solutions.

**Greedy, DBSCAN + FindPaths**: These algorithms build $k$ trees using Greedy and DBSCAN, respectively, on top of which we run our FindPaths algorithm (Algorithm 4).

### 4.3 Experimental setup

Here, we discuss the details regarding the setup of the experimental evaluation and our choice of $\lambda$.

### Implementation details

A prerequisite for the algorithms presented in Section 3 is the kTrees algorithm. We use its C++ implementation that is available online\(^5\). The rest of the algorithms are implemented in Python. Experiments were run on a 64-bit MacBook Pro with an Intel Core i7 CPU at 2.6GHz and 16 GB RAM.

**Picking a fixed $\lambda$**: Recall that the purpose of parameter $\lambda$ in Equation (1) is to transform the prizes and the costs into comparable units. For instance, in our Twitter datasets, we utilize the distributions of the node prizes to identify that there are no more than 10 nodes with prize above 500 (number of tweets). The same cut-off for the edge cost histogram is 0.5 (km). As a result, we align the two quantities by picking $\lambda = 10^{-3}$. Similarly, for Flickr, we set $\lambda = 2 \times 10^{-3}$.

### 4.4 Performance evaluation

In this section, we showcase the performance of our algorithms compared to the baselines. To this end, we vary the parameter $k$ to take values in $\{1, 2, 3, \ldots, 20\}$ and compute the cost of the solution. We present the results of this evaluation in Figure 2. The figure comprises 12 panels; each row corresponds to a dataset and each column corresponds to a problem from Section 2. Furthermore, the first three rows consider user activity from Twitter, while the rest are on data from Flickr. As mentioned above, we use $\lambda = 0.001$ for the former and $\lambda = 0.002$ for the latter. Finally, recall that smaller values of cost imply a better solution. Let us now examine the performance of the algorithms, one problem (column) at a time.

**The k-PCTours problem**: For this problem, we compare the performance of kTrees to the baseline algorithms Greedy+kTrees and DBSCAN+kTrees for the k-PCTours problem, which is presented in the first column of Figure 2. Here, in the majority of the panels kTrees (which is the only algorithm with an approximation guarantee) clearly outperforms the other two algorithms. In addition, notice that for small values of $k$, the cost of Greedy+kTrees is close to kTrees because for these values of $k$ their solutions are similar. As $k$ increases however, kTrees prefers splitting highly active subgraphs rather than introducing new, less active ones, which is what Greedy+kTrees does. For this reason, the difference in their performance is more pronounced for larger values of $k$. Furthermore, observe that the cost of the solutions of DBSCAN+kTrees is not decreasing monotonically with $k$, contrary to both kTrees and Greedy+kTrees. Specifically, panels (a), (c), (g) and (i) illustrate that the cost, after reaching a minimum point for some value of $k$, starts increasing again.

Although this may seem counter-intuitive, this behavior is attributed to the way each algorithm functions. In particular, as $k$ grows, kTrees’ and Greedy+kTrees’ “favor” single-node trees with a relatively large prize, since expanding the subgraphs to their neighbors is too expensive, i.e., the edge cost would be bigger than the cost of leaving the nodes out of the solution. On the contrary, DBSCAN is driven by its input parameters and is oblivious to such trade-offs. Initially, the subgraphs it picks have smaller edge cost than the prizes of their nodes. At this stage, the cost of DBSCAN+kTrees decreases with $k$, until it reaches a minimum point. After that point, the cost of keeping the subgraphs connected outweighs their node prizes, and the total cost increases.

\(^5\)https://github.com/ludwigschmidt/cluster_approx
We now investigate the scalability of our proposed algorithms. To do so, we break each of the algorithms down to its components and report how the runtime is split between these components during the program execution.

Table 3 summarizes the results for this experiment. Each row corresponds to a different value of \( k \), while the columns represent the different components of our algorithms. For example, the running time for \( k \text{Tours} \) at \( k = 10 \) is equal to the runtime of \( \text{FindTour} \) at that row, plus the runtime of \( k \text{Trees} \) at that row, plus the runtime of the code outside these two main components. Note that due to the overhead of some processing occurring outside of the main components, the summations do not add up exactly as the final runtimes include these overheads. Overall, column 1 corresponds to the \( k \text{Trees} \) algorithm, columns 1–3 correspond to \( k \text{Tours} \), and columns 1 and 4–8 correspond to \( k \text{Paths} \). The values in the Table are clearly what one would expect, given the running time complexities. We observe that for the largest graph and input parameter \( k \) our methods do not run for more than 36 seconds.

We do not report the respective running times of the baseline algorithms, as their expected times can be derived from Table 3. Notice, that \( \text{Greedy+kTours} \) and \( \text{Greedy+FindPaths} \) require running \( k \text{Tours} \) \( k \) times and then perform \( k \text{Tours} \) and \( \text{FindPaths} \), respectively, for the appropriate value of \( k \). The same applies to \( \text{DBSCAN+kTours} \) and \( \text{DBSCAN+FindPaths} \), only now we need to run the \( \text{DBSCAN} \) algorithm at least once to produce \( k \) subgraphs, which takes \( O(n \log n) \), where \( n \) is the number of the nodes in the graph, and then perform \( k \text{Tours} \) and \( \text{FindPaths} \), respectively.

4.4 Qualitative comparisons

In this section, we aim to provide an intuition about how our algorithms behave in practice by using visualizations. Our focus is on three aspects: (i) the qualitative differences between our proposed approaches and the baseline algorithms, (ii) the purpose of suggesting the two methods \( k \text{Paths}(t = 1) \) and \( k \text{Paths} \), and (iii) the level of interpretation of the results produced by our algorithms.

In Figure 3, we demonstrate how the solutions of our proposed algorithms, namely \( k \text{Paths}(t = 1) \) and \( k \text{Paths} \), differ qualitatively from the ones of the baseline algorithms \( \text{DBSCAN+FindPaths} \) and \( \text{Greedy+FindPaths} \). Initially, we focus on the paths created by the baseline algorithms. On one hand, we observe in panels 3(c) and 3(g) that \( \text{DBSCAN+FindPaths} \) creates significantly shorter paths compared to the other algorithms, but these paths are high-valued with large prizes. On the other hand, for a smaller value of \( k \), such as in panel 3(d), \( \text{Greedy+FindPaths} \) tends to find some relatively larger paths, compared to the other algorithms, but as \( k \) increases it begins adding to the solution smaller paths, as shown in panel 3(h).

For even larger values of \( k \) we expect the greedy-based algorithm to output paths of singleton nodes because the algorithm lacks the ability of breaking large high-valued paths into smaller dense paths. Regarding the quality of the solutions of the proposed algorithms \( k \text{Paths}(t = 1) \) and \( k \text{Paths} \) we observe that both algorithms find very similar and high quality paths, so now we focus on their main difference using Figures 3 and 4.

The two algorithms follow a similar methodology and, consequently, result in solutions of similar cost. However, it becomes clear from the above Figures that the resulting paths can differ in practice, depending on the application. In particular, notice that in panel 3(a), one of the paths that \( k \text{Paths}(t = 1) \) returns reaches the

<table>
<thead>
<tr>
<th>( k )</th>
<th>( k \text{Trees} )</th>
<th>( \text{FindTour} )</th>
<th>( k \text{Tours} )</th>
<th>( \text{Binarize} )</th>
<th>( \text{FindPathsOnTree} )</th>
<th>( \text{SplitBudget} )</th>
<th>( \text{RetrievePath} )</th>
<th>( k \text{Paths} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0791</td>
<td>0.639</td>
<td>0.7182</td>
<td>0.0145</td>
<td>0.0633</td>
<td>0.0</td>
<td>0.0015</td>
<td>0.1586</td>
</tr>
<tr>
<td>5</td>
<td>0.0654</td>
<td>2.0756</td>
<td>2.141</td>
<td>0.0753</td>
<td>1.6741</td>
<td>0.0002</td>
<td>0.0182</td>
<td>1.8334</td>
</tr>
<tr>
<td>10</td>
<td>0.0655</td>
<td>3.9507</td>
<td>4.0163</td>
<td>0.1522</td>
<td>7.1594</td>
<td>0.001</td>
<td>0.0886</td>
<td>7.4668</td>
</tr>
<tr>
<td>15</td>
<td>0.0645</td>
<td>5.4639</td>
<td>5.5284</td>
<td>0.2519</td>
<td>18.4105</td>
<td>0.0027</td>
<td>0.0976</td>
<td>18.8273</td>
</tr>
<tr>
<td>20</td>
<td>0.0641</td>
<td>7.3068</td>
<td>7.3709</td>
<td>0.3631</td>
<td>35.3417</td>
<td>0.0054</td>
<td>0.1454</td>
<td>35.9198</td>
</tr>
</tbody>
</table>

The \( k \text{-PCPATHS} \) problem: The second column of Figure 2 shows the performance of our algorithms and the baselines for \( k \text{-PCPATHS} \) over the multiple datasets that we consider. We observe that, as expected, \( \text{kPaths}(t = 1) \) and \( k \text{Paths} \) outperform the baseline algorithms, and that \( \text{Greedy+FindPaths} \) has the worst performance. The \( \text{DBSCAN} \)-based baseline algorithm seems to perform better in the \( k \text{-PCPATHS} \) problem compared to its previous performance for the \( k \text{-PCTOURS} \) problem. Furthermore, in the case of \( \text{Flickr Austin} \) at panel 2(j), for \( k \leq 10 \) it seems to provide the best solution.

Discussion: We conclude with two more comments. First, the performance of \( k \text{Tours} \), \( \text{kPaths}(t = 1) \) and \( k \text{Paths} \) shows in all plots a sudden decrease in cost for some values of \( k \). This indicates that when transitioning from \( k \) to \( k - 1 \), the algorithm prefers merging two high-surplus components, i.e. two components whose total prize is larger when merged than the prize when kept separate. In a sense, these components have enough "budget" to pay for the edges that connect them. Splitting these merged components is reflected in the steep cost decrease, which implies that for a larger \( k \), the algorithm splits two high-surplus components.
5 RELATED WORK

Our work is related to a lot of existing work in theoretical computer science, activity-network mining and spatial data analysis. Below, we review this work and comment on how it relates to ours.

Prize-collecting trees, tours and paths: There is a lot of work in theoretical computer science on prize-collecting problems. In these problems the input is a graph \( G = (V, E) \), where every node is associated with a prize and each edge with a weight. The goal is to identify one subgraph of \( G \), say \( G' = (V', E') \), such that \( G' \) has a certain structure (e.g., it is a tree or a path) and the total weight of the edges in \( E' \) plus the prices of the nodes in \( V \setminus V' \) is minimized.

The most well-known among these problems is the prize-collecting Steiner Tree, where \( G' \) is required to be a tree. Using a primal-dual scheme, Goemans and Williamson [10] designed a 2-approximation algorithm for PCST with time complexity \( O(n^3 \log n) \). Later, the running time of this algorithm was reduced to \( O(n^2 \log n) \) [13] and more recently [11] to \( O(dm \log n) \), where \( d \) is the bits of precision of the edge costs and node prizes. The idea of restricting \( G' \) to being a tour or a path has also been considered in the past [1, 2, 4]. The key difference between all the above works and ours is that we focus on finding \( k \) instead of one subgraph in the given graph. Although we use ideas from the above problems to solve ours, both our formulations and algorithmic techniques are new.

The most relevant to ours is the work by Hegde et al. [12], which finds \( k \) trees instead of one. Hegde et al. modify their nearly-linear time algorithm for PCST [11] to provide a 2-approximation algorithm for this problem as well. In fact, our algorithms for \( k \)-PTOURS and \( k \)-PCPATHS use their algorithm as a subroutine.

Applications of PCST: Recently, Rozenstein et al. [19] presented an optimization problem for event detection in activity networks, which reduces to the PCST formulation. Although we also mine activity networks, our problem formulations are different. First, we detect \( k \) subgraphs (which can be routes and paths), while they only focus on detecting one single tree. Moreover, our algorithms run in time nearly-linear to \( m \), while they use the algorithm by Johnson et al. [13], which runs in time \( O(n^2 \log n) \).

Spatial Data Analysis. On a high level, our problem is also related to works on spatial scan statistics [6, 14, 16]. These works assume that data points are distributed on the Euclidean space based on some distribution. The goal is to detect sub-areas (usually
of pre-specified size or shape) that display significantly high density; this requirement is usually quantified in the form of a statistic. These works are clearly different from ours as we focus on activity networks rather than points on the 2-dimensional Euclidean space.

More recently, scan statistics have been extended to also work on graphs [3, 5, 9, 20, 23], and thus output connected subgraphs of the input graph with high value of the scan statistic. Although some of these works impose constraints on the output graph (e.g., size, connectivity), none of them has the flexibility of asking for subgraphs of a particular structure. We believe that this is crucial when analyzing spatial data that correspond to routes taken by city dwellers. As a result, the algorithmic problems underlying this work are totally different from the ones we consider in this paper.

Finally, one can view all density-based clustering methods [7, 17, 18, 21, 22] as related to our work. After all, they identify areas of unusually high density of points. However, these methods do not consider an underlying graph structure and, as a result, one cannot impose any graph-based constraints on their output. Consequently, our algorithmic techniques are also different.

6 CONCLUSIONS

In this paper, we introduced the general $k$-PCSUBGRAPHS problem in activity networks, where the goal is to find a set of connected and non-overlapping subgraphs with nodes exhibiting high levels of activity. More specifically, we focused on two of its variants, namely the $k$-PC TOURS and the $k$-PC PATHS problems. Using existing ideas from theoretical computer science, we gave almost linear-time algorithms for solving all these problems and discussed their approximation guarantees. In our experimental evaluation, we used activity networks consisting of an underlying road network of three cities: Boston, San Francisco and Austin. Each node of the network,
which corresponded to an intersection, was associated with the number of tweets or photo uploads made by social-network users in the vicinity of the intersection. The application of our algorithms to such datasets demonstrated the efficiency of our algorithms and their ability to extract interesting patterns of human activity in urban settings.

**Reproducibility:** All of our code and data will be available for research purposes with the camera-ready version of this paper.
REFERENCES


