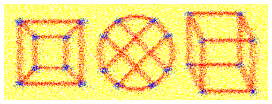
Finding the most relevant fragments in networks


Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 14, no. 2, pp. 307–336 (2010)

Finding the Most Relevant Fragments in Networks

Kevin Buchin[1] *Sergio Cabello*[2] *Joachim Gudmundsson*[3]
Maarten Löffler[4] *Jun Luo*[5] *Günter Rote*[6] *Rodrigo I. Silveira*[7]
Bettina Speckmann[1] *Thomas Wolle*[3]

[1]Department of Mathematics and Computer Science,
TU Eindhoven, The Netherlands
[2]Institute of Mathematics, Physics and Mechanics, and Faculty of
Mathematics and Physics, University of Ljubljana, Slovenia
[3]NICTA Sydney, Locked Bag 9013, Alexandria NSW 1435, Australia
[4]Computer Science Department, University of California, Irvine, USA
[5]Shenzhen Institute of Advanced Technology,
Chinese Academy of Sciences, China
[6]Institut für Informatik, Freie Universität Berlin, Germany
[7]Departament de Matemàtica Aplicada II, Universitat Politècnica de
Catalunya, Barcelona, Spain

Abstract

We study a point pattern detection problem on networks, motivated by applications in geographical analysis, such as crime hotspot detection. Given a network N (a connected graph with non-negative edge lengths) together with a set of sites, which lie on the edges or vertices of N, we look for a connected subnetwork F of N of small total length that contains many sites. The edges of F can form parts of the edges of N.

We consider different variants of this problem where N is either a general graph or restricted to a tree, and the subnetwork F that we are looking for is either a simple path or a tree. We give polynomial-time algorithms, NP-hardness and NP-completeness proofs, approximation algorithms, and also fixed-parameter tractable algorithms.

Submitted:	Reviewed:	Revised:	Accepted:
October 2009	January 2010	March 2010	April 2010
	Final: May 2010	Published: June 2010	
Article type: Regular paper		Communicated by: D. Wagner	

E-mail addresses: kbuchin@win.tue.nl (Kevin Buchin) sergio.cabello@fmf.uni-lj.si (Sergio Cabello) joachim.gudmundsson@nicta.com.au (Joachim Gudmundsson) mloffler@uci.edu (Maarten Löffler) jun.luo@sub.siat.ac.cn (Jun Luo) rote@inf.fu-berlin.de (Günter Rote) rodrigo.silveira@upc .edu (Rodrigo I. Silveira) speckman@win.tue.nl (Bettina Speckmann) thomas.wolle@nicta.com.au (Thomas Wolle)

1 Introduction

Consider the following scenario: You are given a detailed map of the road network of an area together with the exact locations of all crimes committed during the last year. Your job is to determine the area of the network with the greatest concentration of crimes. To do so, you will want to find many crimes that are somehow "close". But finding crimes whose locations are close with respect to the Euclidean distance might not give you the right answer—the crimes need to be close with respect to the road network. In other words, you need to find a comparatively "small" fragment of the network which contains the locations of many crimes. This is usually referred to as a crime *hotspot*.

The problem of detecting crime hotspots has received a lot of attention in recent years (see for example [10, 23, 29, 30, 32]). Crime hotspots are relevant to both crime prevention practitioners and police managers: They allow local authorities to understand what areas need most urgent attention, and they can be used by police agencies to plan better patrolling strategies.

Most problems of this type have been almost exclusively considered in the fields of geographic data mining [24] and geographical analysis [26, 27]. Many different variants of the problem have been studied. The data set can be a point set (each point indicating the location of a crime) or a crime rate aggregated into regions such as police beats or census tracts. Even though both provide useful information, for the purpose of finding hotspots, the precise locations of the crimes are required. Existing methods also differ in the shape of the hotspot. For example, a well-known technique, the "Spatial and Temporal Analysis of Crime", outputs areas of higher crime rate as standard deviational ellipses [19]. However, in urban areas, most human activities, including the criminal ones, are georeferenced to the street network, and any measure of proximity should take the network connectivity and network distances into account, rather than using the Euclidean distance.

Crime hotspot detection is just one application example where this type of spatial data analysis is performed, but many others exist. For example, instead of crimes, one could analyze traffic accident locations, with the goal of finding a comparatively "small" part of the network which contains the locations of many accidents (see for example [20, 25]). A more cheerful scenario that leads to the same algorithmic problem concerns a tour operator that wants to build the perfect bus tour. She knows the road network and she knows where the touristic sights are on this network. Now she wants to find the part of the road network that contains the largest number of sights.

In this paper we address the problem of finding hotspots in networks from an algorithmic point of view. The precise algorithmic problem that we consider is defined in the following.

Formal problem statement. A *network* N is a connected graph with non-negative edge lengths. We view the edges as curves of given lengths, and the network is the union of all edges and vertices, considered as a metric space. Thus, an edge uv of length c is (isometric to) an interval of length c, and it

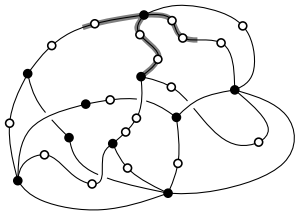


Figure 1: A network with sites; a fragment is highlighted in gray.

contains a point at distance ℓ from u and at distance $c - \ell$ from v, for any ℓ in the interval $0 \leq \ell \leq c$. A *fragment* F of a network N is a connected subgraph of N: The edges of F are contained in edges of N (they are either edges of N or parts of edges of N). The *length* of a fragment F is the sum of its edge lengths. Together with N, we are given a set S of *sites*, which are located on the edges or vertices of N. Generally, we are looking for a fragment of small length that should contain many sites (for an example see Figure 1). More formally, we consider the following problem:

> We are given a network N with m edges, a set S of n sites on N, and a positive real value d. Find a fragment F of N (from a particular class of graphs) of length at most d that contains the maximum number of sites.

Not surprisingly, the most general problem where N is a graph and the fragment F is a graph, a tree, or even a path, is NP-complete (proofs are provided in Section 4). Hence we try to understand how much the problem needs to be simplified to allow for efficient algorithms. For example, the simplest case when N is a path can trivially be solved in $O(n + m)$ time by sweeping a path of length d along N. Exact and efficient algorithms for special (simple) cases are also interesting from a practical point of view, since they often form a foundation for effective heuristics that solve the general case. In addition, we investigate under which realistic input assumptions the general problem becomes tractable.

1.1 Notation

We consider various variants where N is either a tree or a graph and F is either a simple path or a tree. (Note that if F is allowed to be a general graph then the optimal solution will always be a fragment F which is a tree). We denote each variant by the pair of symbols NF, where N and F is one of four codes: **G** stands for a general graph, **T** for a tree, and **P** for a simple path (without repeated vertices). For example, **GP** denotes the instance of the problem where N is a general graph and F is a simple path. All paths considered in this article are *simple*: they do not repeat vertices or edges.

Throughout the paper we assume that the sites are given in sorted order along the edges of N, otherwise sorting the sites would force a lower bound of $\Omega(n \log n)$ for the time complexity of our algorithms.

N / F	**Graph**	**Tree**	**Simple Path (P)**
Graph	Same as **GT**	NP-complete / 4-apx	APX-hard
Tree	–	$O(mn + n^2)$	$O(n + m)$
Path	–	–	$O(n + m)$

Table 1: Summary of the main results obtained for the different variants. The leftmost column shows options for the network N, whereas the top row shows the options for the fragment F.

1.2 Results

Recall that N is a network with m edges and that there are n sites on N. We are looking for a fragment of length at most d which contains the maximal number of sites. A summary of the main results is shown in Table 1.

We first present those variants of the problem that allow for polynomial-time solutions. We show that if N is a tree, efficient algorithms exist. In particular, in Section 2 we consider **TP**: N is a tree and F is a path. In this case we can find the most relevant fragment in $O(n+m)$ time and $O(n+m)$ space. We can also find *all* relevant fragments (that is, all fragments of length at most d that contain a given number k of sites) in $O(m + n + f \log n)$ time where f is the number of relevant fragments. Alternatively, using a different data structure, all relevant fragments can be found in $O(m + n \log n + f)$ time. In Section 3 we discuss **TT**: both N and F are trees. Here we can find the most relevant fragment in $O(mn + n^2)$ time.

Section 4 shows that the variants where N is a graph and F is either a tree or a path are NP-complete. In addition, we also present constant-factor approximation algorithms when F is a tree, and inapproximability results when F is a path.

In Section 5 we study several input assumptions under which efficient algorithms exist for the general problem when N is a graph. For the case in which the network N has bounded treewidth, we give algorithms for **GT** and **GP** that run in $O((m + n)n^2)$ time. If we assume a bound on the maximum vertex degree and on the length of the smallest edge in N—both these assumptions are satisfied in typical street networks—problems **GP** and **GT** can be solved in polynomial time.

1.3 Related work

Spatial analysis has been studied intensively in GIS for decades [14] and it has been used in many other areas such as sociology, epidemiology, and marketing [38]. Many spatial phenomena are constrained to network spaces, especially when they involve human activities. For example, car accidents tend to happen only on roads and gas stations are also usually located along roads. There is an ample body of work concerning spatial network analysis and network restricted clustering [1, 36, 37, 39]. Like many spatial analysis methods, most

spatial network analysis uses statistical methods such as the network K-function method [36]. As already mentioned, the problem of finding crime hotspots has received a lot of attention itself [10, 23, 29, 30, 32]. A large part of the existing methods look for hotspots of a particular shape (like an ellipse). Others instead output a *crime map*, dividing the map into a grid and showing the different crime intensities at every grid cell [29]. Although popular in practice, these methods in general do not provide guarantees on the output quality or running time.

On the more algorithmic side, the problems studied in this paper are related to the *orienteering problem* [17] (also known as *bank robber problem* [4]), as well as to the well-known k-MST and k-TSP problems. In the graph version of the orienteering problem one is given a graph with lengths on edges and rewards on nodes, and the goal is to find a path in the graph that maximizes the reward collected, subject to a hard limit on the length of the path. Many variants of the orienteering problem have been studied [2, 4, 7, 11, 12]. Even though most of them look for a path, versions where the subgraph sought is a cycle or tree have also received some attention (see for example [2]). The main difference between the problem considered in this paper and the standard (unrooted) orienteering problem is that due to the motivation of our problem from spatial analysis, we are interested only in paths that do not repeat edges. Moreover, we consider various combinations of types of graphs for N and F, that cannot be handled with standard orienteering algorithms.

There is a close connection between the orienteering problem and the k-TSP and k-MST problems. The former consists in finding a tour of minimum cost that visits at least k vertices, whereas the latter looks for a minimum cost tree that spans at least k vertices. Moreover, the orienteering problem is in some sense dual to the k-TSP, and approximation algorithms for k-TSP can be easily extended to the (unrooted) orienteering problem. Regarding the k-MST problem, the main difference with our problem is that the sites in S do not need to be vertices of N, and that k is not given.

2 TP: N is a tree and F is a path

In this section we assume that the network N is a tree T. We first show in Section 2.1 that we can in fact assume that T is a rooted tree where each internal vertex has two children. Here we also introduce the notation used in this section and state a useful lemma. In Section 2.2 we show how to find the most relevant fragment in linear time and space and in Section 2.3 we explain how to report all relevant fragments.

2.1 Preliminaries

We assume for simplicity of exposition that no site lies on a vertex of T. Our approach is based on dynamic programming, and sites at vertices produce some extra cases that have to be considered. However, they are no fundamental

problem. Select an arbitrary vertex of T as a root, denoted by v_{root}. We transform the input tree into a tree where each internal vertex v has precisely two children, denoted by v_ℓ, v_r (see Figure 2): a vertex with $t \geq 3$ children can be replaced by a path of $t-1$ degree-three vertices with zero-length edges between them. Vertices with a single child can be eliminated by simply merging the two incident edges. A fragment in the original network corresponds to a fragment of the same length in the new network, and vice versa.

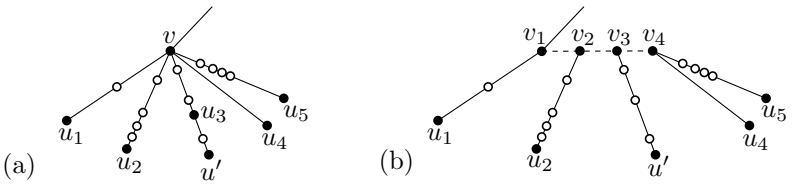


Figure 2: Transforming the input tree (a) into a rooted tree where each internal vertex has two children (b). The dashed edges in (b) have length zero.

We preprocess T so that the distance $d_T(v, v')$ can be obtained in constant time for any query pair of vertices v, v' in T. This can be done in linear time by building a data structure for lowest common ancestor queries [6] and storing for each vertex its distance from the root.

For any pair of sites a, b in the tree T, let $\pi_T(a, b)$ denote the unique path in T that connects them. Let $n(F)$ denote the number of sites of S contained in a fragment F of T, and in particular, let $n(uv)$ denote the number of sites of S along the edge uv. For each vertex v of T, let $T(v)$ denote the subtree of T rooted at v, and let $p(v)$ be the maximum number of sites from S contained in any path from v to a leaf of $T(v)$. For any edge vu, where v is the parent of u, let $T(vu)$ be the subtree consisting of $T(u)$ plus the edge vu, and let $p(vu) = n(vu) + p(u)$ be the maximum number of sites from S contained in any path from v to a leaf of $T(vu)$. The following bounds will be useful to analyze our algorithms.

Lemma 1

$$\sum_{\substack{u \in V(T) \\ u \text{ not a leaf}}} \min\{p(uu_r), p(uu_\ell)\} \leq n$$

and

$$\sum_{\substack{u \in V(T) \\ u \text{ not a leaf}}} n(T(uu_r)) \cdot n(T(uu_\ell)) \leq n^2.$$

Proof: We first prove the first formula. Define for each vertex $v \in V(T)$ the value

$$\sigma(v) = \sum_{\substack{u \in V(T(v)) \\ u \text{ not a leaf}}} \min\{p(uu_r), p(uu_\ell)\}.$$

We claim that $\sigma(v)+p(v)=n(T(v))$ for any $v\in V(T)$. This claim implies that $\sigma(v_{\text{root}})\leq n$, and hence the result follows. The claim is proved by induction on the size of the subtrees. The claim holds for any leaf v because $\sigma(v)=0$. Consider an interior vertex v and assume without loss of generality that $p(vv_r)\geq p(vv_\ell)$. Then $p(v)=p(vv_r)=p(v_r)+n(vv_r)$ and $\min\{p(vv_r),p(vv_\ell)\}=p(v_\ell)+n(vv_\ell)$. Hence, we can use the induction hypothesis on $\sigma(v_r),\sigma(v_\ell)$ to conclude

$$\begin{aligned}\sigma(v)+p(v)&=\min\{p(vv_r),p(vv_\ell)\}+\sigma(v_r)+\sigma(v_\ell)+p(v_r)+n(vv_r)\\&=p(v_\ell)+n(vv_\ell)+\sigma(v_r)+\sigma(v_\ell)+p(v_r)+n(vv_r)\\&=n(vv_r)+n(vv_\ell)+n(T(v_r))+n(T(v_\ell))\\&=n(T(v)).\end{aligned}$$

This finishes the proof of the first formula. To prove the second formula, consider for each node $v\in V(T)$ the value

$$\pi(v)=\sum_{\substack{u\in V(T(v))\\ u\text{ not a leaf}}} n(T(uu_r))\cdot n(T(uu_\ell)).$$

We claim that $\pi(v)\leq n(T(v))^2/2$ for any $v\in V(T)$, which implies the result. The claim is also proved by induction on on the size of the subtrees. The result clearly holds when v is a a leaf because $\pi(v)=0$. For an internal vertex v we can use the induction hypothesis to argue

$$\begin{aligned}\pi(v)&=n(T(vv_r))\cdot n(T(vv_\ell))+\pi(v_r)+\pi(v_\ell)\\&\leq n(T(vv_r))\cdot n(T(vv_\ell))+n(T(v_r))^2/2+n(T(v_\ell))^2/2\\&\leq n(T(vv_r))\cdot n(T(vv_\ell))+n(T(vv_r))^2/2+n(T(vv_\ell))^2/2\\&=\left(n(T(vv_r))+n(T(vv_\ell))\right)^2/2=n(T(v))^2/2.\end{aligned}$$

This finishes the proof of the second formula. □

2.2 Finding the most relevant path

In this section we use dynamic programming to find a path in T of total length at most d that covers the maximum number of sites of S. The approach requires linear time and space.

For each interior vertex v we compute lists $P(v),P(vv_r),P(vv_\ell)$ (see Figure 3). The list $P(v)$ has $p(v)$ elements. The j^{th} element is (a pointer to) a site $s\in S$ with the property that the path $\pi_T(v,s)$ is a path of minimum length among the paths contained in $T(v)$ that start in v and contain j sites of S. Analogously, the list $P(vv_\ell)$ has $p(vv_\ell)$ elements, storing the minimum-length paths in $T(vv_\ell)$ that have one endpoint in v, and similarly for $P(vv_r)$.

We compute these lists recursively in a bottom-up manner. These lists are extended by adding elements at the front. Thus, we store each list as an *extensible array*, but we store the elements in reverse order: the j^{th} element of a list of length m is stored in array position $A[m-j]$. Standard techniques

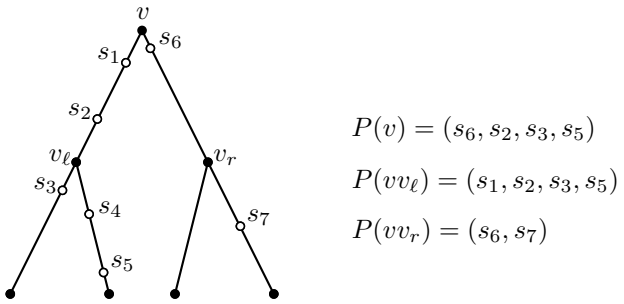


Figure 3: Example showing the lists used by the algorithm for **TP**, for a vertex v.

can be used to implement such arrays with constant access time and amortized constant time for extending them by one element [13, Section 17.4]. The total space is linear in the total number of added elements. The arrays are reused for different lists to achieve overall linear time and space.

We process the tree bottom-up and maintain a value $k_{\max}$ that equals the number of sites of S in the best path of length d so far. Initially $k_{\max} = 1$. When v is a leaf, we allocate an empty list $P(v)$ and set $p(v) = 0$. Consider an internal vertex v. Its two children v_r, v_ℓ have already been processed. We aim for a time bound of $O(n(vv_r) + n(vv_\ell) + \min\{p(vv_r), p(vv_\ell)\})$ for processing v.

(i) We construct $P(vv_r)$ and $P(vv_\ell)$. $P(vv_r)$ is obtained by adding the ordered sequence (from v to v_r) of $n(vv_r)$ sites of S on the edge vv_r to the beginning of the list $P(v_r)$. The list $P(v_r)$ is destroyed in this operation. We construct $P(vv_\ell)$ similarly, and the total amortized running time is $O(1 + n(vv_r) + n(vv_\ell))$.

(ii) We find the best path contained in $T(v)$ that intersects vv_r but not vv_ℓ. We look for a path containing more than $k_{\max}$ sites of S by simultaneously scanning $P(vv_r)$ with a shifted copy of itself. Formally, we start with $j = 1$, and while $j \le n(vv_r)$ and $j + k_{\max} \le p(vv_r)$ do:

   (a) if the distance between the j^{th} site of $P(vv_r)$ and the $(j + k_{\max})^{\text{th}}$ site of $P(vv_r)$ is at most d, then we increment $k_{\max}$ by one.
   (b) otherwise, we increment j by one.

   The same approach can be used to find the best path among those contained in $T(v)$ and intersecting vv_ℓ but not vv_r. To bound the running time, note that case (b) happens at most $n(vv_r) + n(vv_\ell)$ times, and that each time that case (a) occurs, the value $k_{\max}$ is incremented by one. Therefore, this task takes $O(1 + \Delta + n(vv_r) + n(vv_\ell))$ time, where Δ is the increment in the value of $k_{\max}$.

(iii) We find the best path in $T(v)$ that intersects both vv_r and vv_ℓ. The idea is as above: we simultaneously scan the lists $P(vv_\ell)$ and $P(vv_r)$, looking

for a path with $k_{\max}+1$ sites and incrementing $k_{\max}$ whenever we find such a path. We first look for the best path from an element of $P(vv_\ell)$ to the first element f of $P(vv_r)$: while $k_{\max} \le p(vv_\ell)$ and the distance between the $(k_{\max})^{\text{th}}$ site of $P(vv_\ell)$ and f is at most d,

(a0) we increment $k_{\max}$ by one.

Now, starting with $j = \min(k_{\max}, p(vv_\ell))$, we will consider paths between the j^{th} element of $P(vv_\ell)$ and the $(k_{\max}-j+1)^{\text{st}}$ element of $P(vv_r)$. Such paths contain $k_{\max}+1$ sites. While $j \ge 1$ and $k_{\max}-j+1 \le p(vv_r)$ do:

(a) if the distance between the j^{th} element of $P(vv_\ell)$ and the $(k_{\max}-j+1)^{\text{st}}$ element of $P(vv_r)$ is at most d, then we increment $k_{\max}$ by one.

(b) otherwise, we decrement j by one.

Case (b) happens at most $\min\{p(vv_r), p(vv_\ell)\}$ times, and each time that case (a0) or (a) occurs, the value $k_{\max}$ is incremented by one. Therefore, this task takes $O(1+\Delta+\min\{p(vv_r), p(vv_\ell)\})$ time, where Δ is the increment in the value $k_{\max}$.

The operations of steps (ii) and (iii) together have now taken care of all paths in $T(v)$ that are not contained in one of the subtrees $T(v_\ell)$ or $T(v_r)$.

(iv) Finally, we compute $P(v)$ by taking the elementwise minimum of the two lists $P(vv_\ell)$ and $P(vv_r)$. Assume without loss of generality that $p(vv_\ell) \le p(vv_r)$; then we will reuse the list $P(v_r)$ to represent the list $P(v)$. For each $j = 1, \ldots, p(vv_\ell)$, the j^{th} element of $P(v)$ is simply the minimum of the j^{th} element of $P(vv_r)$ and the j^{th} element of $P(vv_\ell)$. The elements beyond the $p(vv_\ell)^{\text{th}}$ element are left unchanged. This pairwise comparison of the two lists takes $O(1+\min\{p(vv_r), p(vv_\ell)\})$ time.

After processing each vertex v of T, we have computed the optimum value $k_{\max}$. Of course, the pair of sites defining the optimum path can be retrieved if we remember the relevant pair of sites each time we increment $k_{\max}$. At each vertex v we spend $O(1+\Delta(v)+n(vv_r)+n(vv_\ell)+\min\{p(vv_r), p(vv_\ell)\})$ time, where $\Delta(v)$ is the increment that $k_{\max}$ takes when processing vertex v. The sum of $\Delta(v)$ over all vertices v is the final value of $k_{\max}-1$, and therefore is bounded by n. The sum of $n(vv_r)+n(vv_\ell)$ over all vertices v is n, since each site is counted once in the sum. The sum of $\min\{p(vv_r), p(vv_\ell)\}$ over all vertices v is $O(n)$ because of Lemma 1. The total number of elements added to the lists is n, and hence the storage requirement is $O(m+n)$. (There is an $O(1)$ storage overhead for each of the $m-1$ lists.) We summarize.

Theorem 1 *Given a tree-network with m vertices, a set S of n sites along its edges, and a value d, we can find in $O(n+m)$ time and $O(n+m)$ space a path fragment that has length at most d and contains the maximum number of sites from S.*

2.3 Finding all relevant fragments

By extending the ideas of the previous section, we can report *all* (combinatorially distinct) paths of length at most d in a tree T with a given number k of sites. That is, we want the set $\mathcal{P}$ of all pairs (a, b) of sites for which the path $\pi_T(a, b)$ has length at most d and contains exactly k sites. As a preprocessing step, we compute for all sites their distance from the root v_{root} in linear time. Note that if a set of sites in $T(v)$ or $T(vv_r)$ is sorted by their distance from the root v_{root} it is also sorted by their distance from v.

Our approach reuses many ideas from Section 2.2. We also use dynamic programming that processes the tree bottom-up. For each interior vertex v we have lists $L(v)$, $L(vv_r)$, and $L(vv_\ell)$ with $p(v)$, $p(vv_r)$, and $p(vv_\ell)$ elements, respectively. We refer to these lists as *first-level lists*. In the following, to avoid repetitions, let $\star$ be a generic symbol to denote v, vv_r, or vv_ℓ. The j^{th} element of the list $L(\star)$ is (a pointer to) a sorted list containing all sites s in $T(\star)$ such that $S \cap \pi_T(v, s)$ has j sites, where the sorting key of a site is its distance from the root. We refer to such sorted lists as *second-level lists*. Hence, in a second-level list, the sites are sorted by their distance from the root v_{root} and from v. Using as key the distance from the root is more convenient than the distance form v, since the second-level lists are merged and recombined into other lists $L(v)$ for different vertices v.

Like in Section 2.2, each first-level list $L(\star)$ is stored in an extensible array in reverse order, so that we can append elements to the list $L(\star)$ at the front in amortized constant time, and the list uses linear space in the number of stored elements. Each second-level list is stored in a linear-space data structure such that the operations Creation, FindMin, and FindNext take $O(1)$ time, and merging two lists with y and z elements, $y \leq z$, takes $O(y \log(1 + z/y))$ time. This can be obtained with a height-balanced binary tree [9]. (We could also use finger trees [18] as an alternative representation.) Note that different lists in $L(\star)$ store disjoint sets of sites, and hence all the second-level lists of $L(\star)$ together store $|S \cap T(\star)|$ sites and use $O(|S \cap T(\star)|)$ space. We will use the following result to bound the time complexity of our algorithm.

Lemma 2 *A sequence of merges of second-level sorted lists resulting in a second-level sorted list with x sites takes $O(x \log x)$ time.*

Proof: Suppose that merging two second-level sorted lists with y and z elements, where $y \leq z$, into a list of length $x = y + z$ takes at most $C \cdot y \log_2(1 + z/y) = C \cdot y \log_2(x/y)$ time, for some constant C. We show by induction on x that the total time for all merges is at most $C \cdot x \log_2 x$. If the final second-level sorted list is obtained by merging two second-level sorted lists with y and z elements, where $y \leq z$ and $x = y + z$, then total time is bounded by

$$\begin{aligned} C(y \log(x/y)) + C(y \log y) + C(z \log z) &= C(y \log x + z \log z) \\ &\leq C((y + z) \log x) = C(x \log x) \end{aligned}$$

□

We process the tree bottom-up, reporting the pairs of $\mathcal{P}$ as we find them. When v is a leaf, we allocate an empty list $L(v)$ and set $p(v) = 0$. Consider an internal vertex v. Its two children v_r and v_ℓ have already been processed. We next process v to find the set $\mathcal{P}(v)$ of all pairs $(a, b) \in \mathcal{P}$ such that a is in $vv_r \cup vv_\ell$ and b is in $T(v)$. When $n(vv_r) > 0$, let $s_1, \ldots s_{n(vv_r)}$ denote the sites along the edge vv_r ordered from v to v_r. Without taking into account the time used for merging second-level lists, we aim for a time bound of $O(1 + n(vv_r) + n(vv_\ell) + \min\{p(vv_r), p(vv_\ell)\} + |\mathcal{P}(v)|\})$ for processing v. The time used for merging second-level lists will be bounded globally using Lemma 2.

(i) We construct $L(vv_\ell)$ and $L(vv_r)$. If $n(vv_r) = 0$, then $L(vv_r) = L(v_r)$. Otherwise, we obtain $L(vv_r)$ as follows: for $j = n(vv_r), \ldots, 1$, we create a new second-level list that contains the site s_j and append it at the front of the list $L(v_r)$. The list $L(v_r)$ is destroyed in this operation. We construct $L(vv_\ell)$ similarly, and the total amortized running time is $O(1 + n(vv_r) + n(vv_\ell))$.

(ii) We find the pairs $(a, b) \in \mathcal{P}(v)$ such that a is in vv_r and $\pi_T(a, b)$ is contained in $T(vv_r)$. For $j = 1, \ldots, \min\{n(vv_r), p(vv_r) - k + 1\}$, we find the pairs (s_j, b) such that b is a site from the second-level $(j + k - 1)^{\text{st}}$ list of $L(vv_r)$ and at distance at most d from a, and report them. This takes $O(1 + \Delta)$ time, where Δ is the number of reported pairs, if we iteratively access the sites of the second-level $(j + k - 1)^{\text{st}}$ sorted list in sorted order until we find an element whose distance from s_j is larger than d.

The same approach can be used to find the pairs $(a, b) \in \mathcal{P}$ such that a is in vv_ℓ and $\pi_T(a, b)$ is contained in $T(vv_\ell)$. Therefore, this task takes $O(1 + \Delta + n(vv_r) + n(vv_\ell))$ time, where Δ is the number of reported pairs.

(iii) We find the pairs $(a, b) \in \mathcal{P}(v)$ such that $\pi_T(a, b)$ intersects both vv_r and vv_ℓ. The idea is as follows: for each appropriate value of j, we have to consider the pairs (a, b) where a is a site from the second-level j^{th} list of $L(vv_r)$ and b is a site from the second-level $(k - j)^{\text{th}}$ list of $L(vv_\ell)$, because $\pi_T(a, b)$ contains exactly k sites, and report the ones where the distance between a and b is at most d. However, the second-level lists are sorted by the distance from v, and hence for any a we can obtain the different candidate b's by increasing distance from a. Formally, we start with $j = \min\{k, p(vv_r)\}$, and while $j \geq 1$ and $k - j \leq p(vv_\ell)$ do:

(a) we take a to be the first element in the second-level j^{th} sorted list of $L(vv_r)$.

(b) we repeat the following, until a is not defined or no pair is reported in the iteration:

(b1) we find the pairs (a, b) such that b is a site from the second-level $(k - j)^{\text{th}}$ list of $L(vv_\ell)$ at distance at most d from a, and report them. This takes $O(1 + \Delta)$ time, where Δ is the number

of reported pairs, if we iteratively access the sites of the second-level $(k-j)^{\text{th}}$ sorted list of $L(vv_\ell)$ until we find an element whose distance from a is larger than d.

(b2) update a to be the successor of the current a in the second-level j^{th} sorted list of $L(vv_r)$.

(c) we decrement j by one.

For each j, we spend $O(1+\Delta)$ time, where Δ is the number of reported pairs. Since there are at most $\min\{k, p(vv_r), p(vv_\ell)\}$ possible values for j, this task takes $O(1+\Delta+\min\{p(vv_r), p(vv_\ell)\})$ time.

The operations of steps (ii) and (iii) together have reported all pairs in $\mathcal{P}(v)$, and each pair is reported exactly once.

(iv) Finally, we compute $L(v)$ by joining the information in the two first-level lists $L(vv_\ell)$ and $L(vv_r)$. Assume without loss of generality that $p(vv_\ell) \leq p(vv_r)$; then we will reuse the list $L(v_r)$ to represent the list $L(v)$. For each $j = 1, \ldots, p(vv_\ell)$, the j^{th} element of $L(v)$ is obtained by merging the second-level sorted lists stored in the j^{th} position of $L(vv_r)$ and the j^{th} position of $P(vv_\ell)$. The second-level lists beyond the $p(vv_\ell)^{\text{th}}$ position are left unchanged. This step takes $O(1+\min\{p(vv_r), p(vv_\ell)\})$ time, plus the time used to merge the second-level lists.

After processing the last vertex v of T, we have reported all of $\mathcal{P}$ because each pair in $\mathcal{P}$ is in $\mathcal{P}(v)$ for some vertex v. Without counting the time for merging second-level lists, at vertex v we have spent $O(1+n(vv_r)+n(vv_\ell)+\min\{p(vv_r), p(vv_\ell)\})$ time plus $O(1)$ time per reported pair. Noting that each pair of $\mathcal{P}$ is reported once, that the sum of $n(vv_r)+n(vv_\ell)$ over all nodes v is n, and using Lemma 1, the sum over all vertices gives $O(m+n+|\mathcal{P}|)$ time.

It remains to bound the time used for merging second-level lists. Note that after processing the root v_{root} each site appears exactly in one of the second-level lists of $P(v_{\text{root}})$. Hence, all the second-level lists of $P(v_{\text{root}})$ contain exactly n sites. By Lemma 2, we can bound by $O(n \log n)$ the time spent for all the merges performed during the algorithm. We summarize.

Theorem 2 *Given a tree-network T with m vertices, a set S of n sites along its edges, a value d, and a value k, we can report the set $\mathcal{P}$ of pairs (a,b) of sites for which the path $\pi_T(a,b)$ has length at most d and contains exactly k sites, in $O(m+n\log n+|\mathcal{P}|)$ time.*

It can happen that the sites in a path-fragment of length d are contained in a larger set of sites that can still be covered by a path of length d. Our algorithm will report all these fragments, and not just the *maximal* path-fragments (unless $k = k_{\max}$).

A different representation of the second-level lists gives a different time bound. Note that with the second-level lists we only perform two non-trivial operations: merging, and accessing the elements iteratively from the element with

minimum key until some condition is violated. Consider the scenario where we use Fibonacci heaps to implement such second-level list. A Fibonacci heap supports Creation, Insertion, FindMin, and Merge in $O(1)$ time, and Deletion in $O(\log n)$ amortized time. With this data structure, we can access iteratively the element with minimum key and its x successors in $O(x \log n)$ time: we repeat x times FindMin and Delete, and at the end we insert all the deleted elements back. If we implement the second-level lists using Fibonacci heaps, we obtain a multiplicative overhead of $O(\log n)$ time per reported pair. However, the merges of lists over the whole algorithm take $O(n)$ time, because in each vertex v we make $\min\{p(vv_r), p(vv_\ell)\}$ merges, which takes $O(\min\{p(vv_r), p(vv_\ell)\})$ amortized time. We thus obtain the following result.

Theorem 3 *Given a tree-network T with m vertices, a set S of n sites along its edges, a value d, and a value k, we can report the set $\mathcal{P}$ of pairs (a, b) of sites for which the path $\pi_T(a, b)$ has length at most d and contains exactly k sites, in $O(m + n + |\mathcal{P}| \log n)$ time.*

Note that the algorithm remains essentially unchanged if instead of storing lists of sites, we would store lists of lengths from v to those sites. In the next section we adopt this approach, since storing the sites explicitly becomes too costly for trees.

3 TT: Both N and F are trees

In this section we again assume that the input network is a tree T. We use the transformation described in Section 2.1 and can hence assume that T is a rooted tree where each internal vertex v has precisely two children. We also use the notation introduced in Section 2.1.

Our approach is based on dynamic programming, and processes the vertices of T bottom-up. For each internal vertex v we compute a list $L(v)$, and with the help of $L(v)$ we are able to compute the optimal solution where v is the highest vertex in T. (Note that the approach described in this section differs slightly from the one explained in Section 2.2.) The j^{th} entry, $L(v)[j]$, of $L(v)$ stores the length of the smallest tree fragment of $T(v)$ containing v and covering j sites of S. If there is no such tree fragment we set $L(v)[j] = \infty$. We also set $L(v)[0] = 0$ to simplify some formulas below. For each leaf v, the tree $T(v)$ contains no sites of S, and $L(v)$ will be empty. When all the leaves have been processed we continue bottom-up. Consider an interior vertex v for which the lists $L(v_r), L(v_\ell)$ of its children v_r, v_ℓ have already been computed. We compute $L(v)$ as follows:

(i) For each child u of v we build a list $L(vu)$ from $L(u)$ with the following property: The j^{th} entry of $L(vu)$ stores the length of the smallest tree fragment of $T(vu)$ containing v and covering j sites. The list is constructed as follows. Consider the sites $s_1, s_2, \ldots, s_{n(vu)}$ along the edge vu ordered from v to u. For $j = 1, \ldots, n(vu)$, set the j^{th} entry of $L(vu)$ to the distance

between v and s_j. Then, for $j = n(vu), \ldots, n(T(vu))$ we set the j^{th} entry of $L(vu)$ to be $|vu| + L(u)[j - n(vu)]$, where $|vu|$ denotes the length of the edge vu.

The total time to compute the lists $L(vv_r), L(vv_\ell)$ is $O(n(T(v))) = O(n)$.

(ii) The lists $L(vv_r)$ and $L(vv_\ell)$ are used to construct $L(v)$, as follows. For each integer $j = 1, \ldots, n(T(v))$ we set

$$L(v)[j] = \min\{\, L(vv_r)[a] + L(vv_\ell)[b] \;\big|\; 0 \le a \le n(T(vv_r)),\, 0 \le b \le n(T(vv_\ell)),\, a + b = j \,\}.$$

This procedure constructs the list $L(v)$ using time

$$O\big(n(T(vv_r)) + n(T(vv_\ell)) + n(T(vv_r)) \cdot n(T(vv_\ell))\big) = O(n + n(T(vv_r)) \cdot n(T(vv_\ell))).$$

Each vertex v of T is processed once and requires $O(n+n(T(vv_r))\cdot n(T(vv_\ell)))$ time. The sum of $O(n)$ over all vertices is $O(mn)$. The sum of $n(T(vv_r)) \cdot n(T(vv_\ell))$ over all vertices is $O(n^2)$, by Lemma 1. Hence, we can construct the lists $L(v)$ for all vertices v of T in $O(mn + n^2)$ time.

We describe now how to find the most relevant tree fragment of length at most d in T. First, we compute the most relevant tree fragment that does not contain any vertex of T, and therefore is a path. This can be done in $O(n + m)$ time by finding optimal solutions contained in each edge of T. Next, for each vertex v, we use $L(v)$ to find the most relevant tree fragment that has v as highest vertex. Taking the best among these solutions gives the optimal solution. If a tree fragment has v as highest vertex, then it is contained in $T(v_{\text{parent}}v)$, where v_{parent} denotes the parent of v. (We can handle the case $v = v_{\text{root}}$ by adding a dummy parent to v_{root}.) Let $s_1, \ldots, s_{n(v_{\text{parent}}v)}$ be the sites of S on the edge vv_{parent}, ordered from v to v_{parent}. We construct a list $M(v)$, where the j^{th} entry stores the length of the smallest tree fragment of $T(v_{\text{parent}}v)$ that has v as highest vertex and contains j sites of S, using:

$$M(v)[j] = \big\{\, L(v)[a] + |vs_b| \;\big|\; 0 \le a \le n(T(v)),\, 0 \le b \le n(vv_{\text{parent}}),\, a + b = j \,\big\}.$$

Constructing $M(v)$ takes $O(1 + n(T(v)) \cdot n(vv_{\text{parent}})) = O(1 + n \cdot n(vv_{\text{parent}}))$ time for a vertex v of T, which sums up to $O(m + n^2)$ time over all vertices v of T. The largest number of sites contained in a tree fragment with v as highest vertex is given then by the unique index j_v satisfying $M(v)[j_v] \le d$ and $M(v)[j_v + 1] > d$.

Theorem 4 *Given a tree-network with m vertices, a set S of n sites along its edges, and a value d, we can find in $O(mn + n^2)$ time using $O(n)$ space a tree fragment that has length at most d and contains the maximum number of sites from S.*

The number of tree-fragments with prescribed length and maximum number of sites may be exponential in n and m. For example, consider a network that is a star on $n+1 > 3$ vertices where each edge has length one, and place a site in each node of the network with degree one; so there are n sites. For even n, the number of fragments with length $n/2$ and maximum number of sites is $\binom{n}{n/2}$, which is exponential in n. Hence, we did not study efficient algorithms to report *all* optimal solutions.

4 Hardness and approximation results if N is a graph

4.1 GP: N is a graph and F is a path

We begin by showing that for both versions the decision version of the problem is NP-complete.

Theorem 5 *Given a graph-network with a set of sites, it is NP-complete to determine, for a given length d and a given number of sites k, if there is a path-fragment F of length at most d that contains k sites.*

Proof: We will use the Hamiltonian path problem on graphs of degree at most three: Given a graph $H = (V, E)$ of maximal degree 3, decide whether there is a path in H that visits each vertex H exactly once. This problem is NP-complete; see for example [15, 28]. If we consider H as a network where each edge has unit length and add a site on each vertex, then there is a path-fragment of length $|V| - 1$ that contains $|V|$ vertices if and only if H has a Hamiltonian path. The result follows. □

The reduction to Hamiltonian path used in the proof of Theorem 5 also gives us the following corollary, by setting d very large (i.e., much larger than the total length of all edges in N).

Corollary 1 *Given a graph-network with a set of sites, it is NP-hard to approximate the length d of a path-fragment F that covers $k_{\max}$ sites, where $k_{\max}$ is the number of sites contained in an optimal path-fragment of length d.*

For **GP**, where the path cannot use vertices more than once, we prove NP-hardness also for approximating the number of sites k.

Theorem 6 *Given a graph-network with a set of sites, it is NP-hard to approximate within a constant factor the maximum number of sites k contained in a path-fragment F of length at most d.*

Proof: The reduction is from the *longest path* problem. The input is a graph $H = (V, E)$, and the goal is to compute a path of maximum length. This problem was shown to be hard to approximate within a constant factor by Karger et al. [21].

The reduction takes the input graph H and places one site on each edge. That graph and set of sites is a valid input for **GP**. It is easy to verify that any algorithm for **GP** that approximates k for a given length d can be used to get an approximate solution to the longest path problem, by using $d = L$, where L is the sum of all the edge lengths in H. □

4.2 GT: N is a graph and F is a tree

We begin by showing that the decision version of **GT** is NP-complete. The reduction is from the k-minimum spanning tree (k-MST) problem: Given a graph G with non-negative edge weights and two values k and d, decide whether there is a connected tree spanning k vertices of G of total weight at most d. Ravi *et al.* [31] showed that the k-MST problem is NP-complete.

Theorem 7 *Given a graph-network with a set of sites, it is NP-complete to determine, for a given length d and a given number of sites k, if there is a tree-fragment F of length at most d that contains k sites.*

Proof: We know that **GT** is in NP since a given tree can easily be verified to be a valid solution in polynomial time.

For the reduction, we have to reduce an instance of k-MST to a **GT** instance that has a solution of length at most d if and only if the k-MST problem has a solution of length at most d. As input to the k-MST problem we are given a graph G, a positive integer k and a positive value d. For each vertex in G we add a site on it.

Assume we have an algorithm that solves the **GT** decision problem, i.e., it returns 'yes' if there is a subtree of G of length at most d that contains k sites, otherwise it returns 'no'. Clearly, 'yes' is returned if and only if there is a connected tree containing k vertices of G of total length at most d. Since this problem is known to be NP-complete [31] and the reduction to **GT** requires only linear time, the theorem follows. □

We now describe a constant-factor approximation algorithm for the dual problem of minimizing the length of the fragment d that contains a given number of sites. The algorithm uses a polynomial-time 2-approximation algorithm for the minimum k-Spanning Tree (k-MST) problem by Garg [16].

Assume that we are given a network $N = (V, E)$, the set S and a number d as an input for **GT**. Construct a graph G from N by replacing every site s in N with a vertex v_s and a path of length 0 containing $|V|$ vertices connected to v_s. The construction is illustrated in Figure 4.

We will use a 2-approximation algorithm for the k-MST problem. Given a graph H and a positive integer k, this algorithm returns a subtree T of H containing k vertices, whose total weight is at most two times the weight of a k-MST of H.

Now, run the 2-approximation algorithm for $(k \cdot (|V| + 1))$-MST on G. Let T be the spanning tree returned by the approximation algorithm and let S' be the set of vertices of T. Return the subtree of T induced by the vertices $\{v_s \in S' \mid s \in S\} \cup (S' \cap V)$.

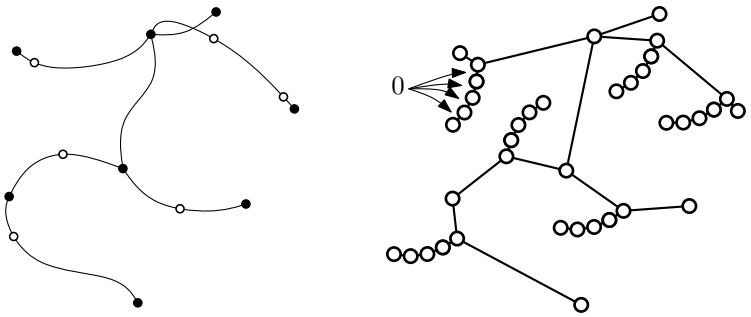


Figure 4: (a) A network with vertices and sites, with geometric curve lengths as edge lengths. (b) The resulting abstract graph, with only vertices, and edges that can have any length. The lengths of the new edges are 0, while the other edges have the same lengths as in (a).

Theorem 8 *The above algorithm is a* 2*-approximation algorithm for* **GT***, where the length* d *of the fragment is approximated.*

Proof: If there exists a fragment spanning k sites in N of length d_N then there exists a $(k(|V|+1))$-MST of G of length at most d_N.

On the other hand, suppose there exists a $k(|V|+1)$-MST of G of length d_G. Then the fragment must cover at least k sites; otherwise, the maximum number of vertices in S would be $(k-1)(|V|+1)+|V|$ which is less than $k(|V|+1)$, a contradiction. Therefore there exists a fragment spanning k sites in N of length at most d_G, thus $d_N = d_G$.

Since both problems are equivalent, using a 2-approximation algorithm for k-MST, the theorem follows. □

To approximate the number of sites k, we can directly use an approximation algorithm for a variant of the orienteering problem called *tree-orienteering*, where the network sought is a tree. Arkin *et al.* [2] propose a 5-approximation algorithm for this problem, based on reusing an approximation algorithm for k-MST. Using the currently best result for k-MST by Garg [16], the factor obtained by the algorithm of Arkin *et al.* automatically improves to 4.

Theorem 9 *There exists a polynomial-time* 4*-approximation algorithm for* **GT***, where the number* k *of sites contained is approximated.*

Proof: Since the input to the tree-orienteering problem is a weighted graph G with a prize on each vertex, the input to **GT** must be adapted. Consider the input $N = (V, E)$, S, and d for **GT**. We reduce to the orienteering problem by setting $G = (V', E')$ to be the graph with vertex set $V' = V \cup S$ and replacing each edge (u, v) in E by a path going trough the vertices corresponding to the points of S on (u, v). Conceptually, for each vertex in G corresponding to a point in S we set the prize to be 1, otherwise it is 0. If there is a tree of length

d for the tree-orienteering problem with k prizes, then it corresponds to a tree for our problem containing k sites of S, thus a c-approximation algorithm for the tree-orienteering problem will also be a c-approximation algorithm for **GT**.

Note that most orienteering algorithms do not allow zero-prizes, hence instead of using 0 and 1, we can use 1 and a sufficiently large constant (or equivalently, we can replicate each site sufficiently often). □

5 GP and GT: Exact algorithms

While the general problem considered in this paper is NP-hard, in many applications we have additional information and/or restrictions on the network and the fragment, which make polynomial-time solutions possible. Here we discuss two such scenarios. In Section 5.1 we consider networks N of bounded treewidth and in Section 5.2 we bound the maximum vertex degree of N as well as the length of the smallest edge in N. This second case can be particularly useful in practice. Both cases lead to fixed-parameter tractable algorithms.

5.1 Networks of bounded treewidth

The notions of treewidth and tree-decomposition (introduced by Robertson and Seymour, see e.g. [33, 34]) have proven to be algorithmically very useful (see e.g. [8]). A *tree decomposition* is a mapping of a graph into a tree and the *treewidth* of a graph measures the number of graph vertices mapped onto any tree node in an optimal tree decomposition. It is NP-hard to determine the treewidth of a graph, but many problems on graphs are solvable in polynomial time if the treewidth of the input graph is bounded (see e.g. [8]). First we describe an algorithm for **GT** on a network N of which the treewidth is bounded by a constant. Later we explain the adaptations needed to solve **GP** under a similar setting.

Formally, a *tree-decomposition* of a network $N = (V, E)$ is a pair (T, X) with $T = (I, F)$ a tree, and $X = \{X_i \mid i \in I\}$ a family of subsets of V, called *bags*, one for each node of T, such that

- $\bigcup_{i \in I} X_i = V$;
- for all edges $vw \in E$ there exists an $i \in I$ with $\{v, w\} \subseteq X_i$;
- for all $i, j, k \in I$: if j is on the path in T from i to k, then $X_i \cap X_k \subseteq X_j$.

The *width* of a tree-decomposition $((I, F), \{X_i \mid i \in I\})$ is $\max_{i \in I} |X_i| - 1$. The *treewidth* $tw(N)$ of a network N is the minimum width over all tree-decompositions of N. A tree-decomposition (T, X) is *nice*, if T is rooted and binary, and the nodes are of four types:

- *Leaf nodes* i are leaves of T and have $|X_i| = 1$.
- *Introduce nodes* i have one child j with $X_i = X_j \cup \{v\}$ for some vertex $v \in V$.

- *Forget nodes* i have one child j with $X_i = X_j \setminus \{v\}$ for some vertex $v \in V$.
- *Join nodes* i have two children j_1, j_2 with $X_i = X_{j_1} = X_{j_2}$

The advantage of using a nice tree-decomposition is that, often, developing and describing algorithms is easier. Converting a tree-decomposition into a nice tree-decomposition of the same width can be done in linear time [22]. However, computing a tree-decomposition of N with width $tw(N)$ is NP-hard [3].

We construct a network N' from N by putting the sites of S as vertices of N' on its edges. Putting an additional vertex on an edge is called a *subdivision*. The length of the edges in N' is fixed in a straight-forward manner. N' has $|V| + n$ vertices and $m + n$ edges. We refer to a vertex of N' that originated from N or S as *network-vertex* or *site-vertex*, respectively.

We assume that we are given a nice tree-decomposition (T, X) of N' of width $tw(N)$. Such a tree-decomposition exists, because subdivisions do not affect the treewidth. To each bag i of T, we associate a table containing certain information. This table represents partial solutions for the subnetwork $N'_i \subseteq N'$ induced by the vertices contained in the bags of the subtree of T rooted at i. More specifically, we will keep track of forests in N'_i, of their lengths and of the number of site-vertices they contain. Such a forest might have vertices in common with X_i. These vertices are represented by an *interface*, which is a set of disjoint subsets of X_i. An interface of a forest tells us which vertices of X_i are involved in the forest, and it also tells us which vertices belong to the same tree of that forest.

Our algorithm employs dynamic programming on (T, X). We start at the leaves, and for an internal node i of T, we compute the table of i using the tables of the children of i. For that, we combine the information of compatible interfaces from the children of i. The resulting running time is exponential in the treewidth, but polynomial in the size of the input.

Theorem 10 *Let t_0 be a constant. Given a graph-network N' with m' edges whose treewidth is bounded by t_0, a set S of n sites along its edges, and a value d, we can find in $O((m' + n)\, n^2)$ time a tree-fragment that has length at most d and contains the maximum number of sites from S.*

Proof: Let i be a node of T. Note that $|X_i| \leq tw(N) + 1$, which is assumed to be bounded by a constant. Recall we have defined an *interface* f of X_i as a set of disjoint subsets of X_i:

$$f = \{Z_1, Z_2, ..., Z_{|f|} \mid \forall j Z_j \subseteq X_i \wedge \forall j_1 \neq j_2 : Z_{j_1} \cap Z_{j_2} = \emptyset\}$$

Since $|X_i|$ is bounded by a constant, the number of interfaces of X_i is also bounded by a constant. The table that we associate with node i contains an entry for every non-empty interface f of X_i.

We say that a forest F' of N'_i is *compliant* with the interface f when

- a vertex in X_i is used in F' if and only if this vertex is in $\bigcup_{j=1}^{|f|} Z_j$; and

- any two vertices in $\bigcup_{j=1}^{|f|} Z_j$ belong to the same tree of F' if and only if there exists a set $Z \in f$ that contains these two vertices.

Note that each forest is compliant with a unique interface.

In the table entry for interface f, we store a subtable that has $n+1$ entries—one entry for every possible number s of site-vertices ($s \in \{0, ..., n\}$). In the subtable entry for s, we store the length of a shortest forest F' of N_i' (we might also store pointers to reconstruct the forest itself) that is compliant with f and covers exactly s site-vertices. The size of such a table for X_i is $O(n)$.

Before we look at how these tables are computed for each node, we describe some operations that we do at each node along the way: As soon as an interface contains at least two sets and one of them is empty, we can delete the entire entry for that interface, because an empty set indicates that there is a tree in the forest which cannot be connected anymore. When an entry stores a length that is greater than d, we disregard that entry completely, because the corresponding forest is already too long. Whenever we consider an interface with only one element, we are looking at a tree. At the end, we will find the solution to our problem in the table entries for the root of T, whose interfaces specify trees. During the dynamic programming, we do the following specific procedures for each type of node i.

Leaf nodes: For leaves it is easy, because there is just one vertex $v \in X_i$ and hence there are just two interfaces $\{\{v\}\}$ and $\{\}$, depending on whether we choose v to be a forest.

Introduce nodes: Let i 'introduce' a vertex v, and let j be the child of i. Consider any interface f' of j. When constructing forests of N_i', we have the choice whether or not to use v in such a forest. If we do not use v, then any forest stored at interface f' of j is also a forest for interface f' of i. If we use v, we may connect it to existing forests by using some edges $e_1, e_2, ...$ between v and vertices in $\bigcup_{Z \in f'} Z$. Note that there are at most $t_0 = O(1)$ such edges. For each of these possibilities applied to f', we obtain an interface f of i. Now, the subtables in the table of f' will be used to create entries in the subtables in the table of f. The length of the edges $e_1, e_2, ...$ (if any are used) plus the entry for s' site-vertices for interface f' contribute to the entry for s site-vertices for interface f. Note that $s \in \{s', s' + 1\}$, depending on whether v is a network-vertex or site-vertex. Among all forests that contribute to an entry for s site-vertices in the subtable for interface f, we keep track of the one with smallest length.

Forget nodes: Let i 'forget' a vertex v, and let j be the child of i. We can convert entries for interfaces f' of node j into entries for interfaces f of node i by simply deleting v from the sets of the interface f'. Among all forests that contribute to an entry s for f, we keep track of the one with the smallest length.

Join nodes: Let i be a join node with children j_1 and j_2. Let f_1 and f_2 be interfaces of j_1 and j_2, respectively. If $\bigcup_{Z_1 \in f_1} Z_1 = \bigcup_{Z_2 \in f_2} Z_2$, then we compute the interface f that represents the subgraph which is the union of the two forests corresponding to f_1 and f_2. This interface f for node i is computed as follows. First, we compute $f = f_1 \cup f_2$, and then we repeatedly replace all pairs Z_1 and

Z_2 in f with $Z_1 \cap Z_2 \neq \emptyset$ and $Z_1 \neq Z_2$ by $Z_1 \cup Z_2$, until no such pairs exist anymore.

An entry for s_1 site-vertices for f_1, combined with an entry for s_2 site-vertices for f_2, contributes to the entry for s site-vertices for f; here $s = s_1 + s_2 - x$, where x is the number of site-vertices that occur in both f_1 and f_2, because they have been counted twice. Just like for other nodes, among all forests that contribute to an entry s for f, we keep track of the one with the smallest length.

The correctness of the algorithm relies on the dynamic programming approach and the procedures described above. It follows that passing and computing information from one node to the next is done correctly. Note, however, that at introduce or join nodes, we might temporarily connect trees in such a way that the result is no longer a forest. For instance, we might connect a vertex v introduced at node i to a forest in such a way that the resulting subgraph G_1 contains a cycle. This subgraph G_1 has a certain interface and a certain number of site-vertices, and therefore, it contributes to the corresponding entry in the corresponding subtable. Now consider any edge in the cycle of G_1, and consider the subgraph G_2 that results from G_1 by removing this edge. Also G_2 will be considered when processing node i. Note that G_1 and G_2 have the same interface and the same number of site-vertices. Hence, G_2 contributes to the same entry as G_1, but the total length of G_2 is smaller than that of G_1. Therefore, by keeping track of the minimum length in the entries of the subtables, we ensure that we will indeed store the length of a forest. At each node i, we store tables that represent information on forests of N'_i. An interface that contains only one set of vertices is an interface of a tree. And hence, for each node, we can determine the shortest subtree with maximum number of site-vertices.

For the running time, we observe that at each introduce and forget node we spend $O(n)$ time in total for all values of site-vertices and for all interfaces, since the size and number of interfaces is constant. At a join node, we combine any entry for s_1 with any entry for s_2, which gives $O(n^2)$ time. Now, the theorem follows from the fact that a tree-decomposition of N' has $O(|V(N')|)$ nodes. □

Solving GP. The previous method can be modified to find the shortest path-fragment in N. To do this we need to keep track of *pathsets*, sets of disjoint paths in N'_i, of their cumulative lengths and of the number of site-vertices they contain. Similar to a forest, also a pathset might have vertices in common with X_i, which are represented by an interface. However, we need to extend interfaces f to also reflect which vertices in $Z \in f$ are at the endings of the path represented by Z. With interfaces like this, we can make sure to combine only subsolutions that represent pathsets.

Theorem 11 *Let t_0 be a constant. Given a graph-network with m' edges whose treewidth is bounded by t_0, a set S of n sites along its edges, and a value d, we can find in $O((m'+n)\,n^2)$ time a path-fragment that has length at most d and contains the maximum number of sites from S.*

Proof: The proof is very similar to the proof of Theorem 10. We only address crucial differences here. Let f be an interface of node i as defined above, and

let $Z \in f$ represent a path $P = v_1 v_2 v_3 \dots v_{k-2} v_{k-1} v_k$. Considering P as a string of vertices, we define the X_i-*prefix* of P to be the longest prefix of P, such that every vertex in the X_i-prefix is contained in X_i. Let the X_i-*suffix* be defined in an analogous way. Note that either of X_i-prefix and X_i-suffix can be empty. Now, we *extend the definition of interface*, such that we associate the X_i-prefix $v_1 v_2$... and the X_i-suffix ...$v_{k-1} v_k$ of P as superscripts to Z. Since $|X_i|$ is bounded by a constant, also the number of possible interfaces of X_i is bounded by a constant. For two interfaces to be equal, their elements must be the same which means they also have to agree on their superscripts. Instead of considering a forest, we maintain information corresponding to pathsets.

Leaf nodes: Handling a leaf node i with $X_i = \{v\}$ during the dynamic programming is straightforward and results in two interfaces: $\{\{v\}^{v,v}\}$ and $\{\,\}$, where v is the X_i-prefix and X_i-suffix of $\{v\}$.

Introduce nodes: For an introduce node i ('introducing' a vertex v) with child j and an interface f' of j, we have the following options to obtain an interface f of i. We can connect v to no other path which gives rise to a new element $\{v\}^{v,v} \in f$. Or we may connect v to an endpoint of one path represented by $Z \in f$, which makes v an endpoint of that path, and hence v has to appear in the X_i-prefix or X_i-suffix of Z. Or we can connect v to two endpoints of two different paths represented by $Z_1, Z_2 \in f$, which results in one (longer) path represented by Z. However, we may only do this if we do not create a vertex of degree three or more in Z, which can be ensured, since we know the X_i-prefixes and X_i-suffixes of Z_1 and Z_2. We must not connect v to three or more vertices as this would not result in a path.

Forget nodes: When 'forgetting' a vertex v at a forget node i, we simply delete v from the elements of interfaces of i. Furthermore, if v occurs in an X_i-prefix, we delete v and all successors of v in that X_i-prefix. In a similar way we delete v and all its predecessors in X_i-suffixes.

Join nodes: At a join node i with children j_1 and j_2, let us consider interfaces f_1 and f_2 of j_1 and j_2, respectively. We now have to consider also X_i-prefixes and X_i-suffixes. To see if and how we can combine f_1 and f_2 to an interface f of node i, we do the following. First, we compute $f = f_1 \cup f_2$. And then we repeatedly look at all pairs $Z_1, Z_2 \in f$ ($Z_1 \neq Z_2$) and test (to be described below) if the paths represented by Z_1 and Z_2 can be connected without creating cycles or vertices of degree three or more. If that is the case, we replace Z_1 and Z_2 by $Z_1 \cup Z_2$ in f, and we compute the X_i-prefix and X_i-suffix of Z using the X_i-prefix and X_i-suffix of Z_1 and Z_2. We repeat this until no such pairs exist anymore.

It remains to describe how to do the test to join the paths represented by Z_1 and Z_2. Let P_1 and P_2 be the paths represented by Z_1 and Z_2. If $Z_1 \cap Z_2 = \emptyset$, then P_1 and P_2 cannot be connected to form a single path, because they have no vertex in common; otherwise, we have that $Z_1 \cap Z_2 \neq \emptyset$. Now, if there exist a suffix s of the X_i-suffix of Z_1 and a prefix p of the X_i-prefix of Z_2 with $s = p$ (here s and p are strings), then the ending of P_1 overlaps with the beginning of P_2, which is for our method a necessary condition for the paths to

be connected. Clearly, when interpreting s and p as sets of vertices, we have that $s = p \subseteq Z_1 \cap Z_2$. For P_1 and P_2 to be connected, we also have to ensure that $s = p = Z_1 \cap Z_2$, i.e. $Z_1 \cap Z_2$ has to be exactly the set of vertices where P_1 and P_2 overlap. That is because if there is a vertex $v \in Z_1 \cap Z_2$ with $v \notin s = p$, then we could traverse the vertices starting from v to s on P_1, further along $s = p$ and back to v via P_2. If v is the other ending (other than s and p) of P_1 and P_2, then we have a cycle. If v is internal to P_1 or P_2, then we have a cycle and a vertex of degree at least three. Summarising, only if $s = p$ (where s and p are interpreted as strings) and if $s = p = Z_1 \cap Z_2$ (where s and p are interpreted as vertex sets), we can connect P_1 and P_2 to form a single path. To check all possibilities how two paths could be connected, the above test has to be performed in four different combinations: s is a suffix of the X_i-suffix of Z_1 and p is a prefix of the X_i-prefix of Z_2 (as described above); s is a suffix of the X_i-suffix of Z_2 and p is a prefix of the X_i-prefix of Z_1; s is a suffix of the X_i-suffix of Z_1 and p is a prefix of the reversed X_i-suffix of Z_2; s is a suffix of the reversed X_i-prefix of Z_2 and p is a prefix of the X_i-prefix of Z_1. Hence, only after these tests we know whether combining f_1 and f_2 results in a cycle or a vertex of degree three or more. And if not, we combine f_1 and f_2 to an interface f representing a pathset of N_i'.

The remainder of the proof is analogous to the proof of Theorem 10. In particular, we obtain the solution by considering the paths of length at most d corresponding to interfaces with one set at the root, and selecting the one that covers most site-vertices. □

5.2 Limiting vertex degree and edge length

Real-world road networks are unlikely to contain high degree vertices or very short edges (with respect to the length d of the fragment). Let D be the maximum vertex degree of N, and let s be the length of the shortest edge in N. If we assume that both D and the fraction $f = d/s$ are bounded by a constant, then we can solve **GP** and **GT** in time polynomial in n and m.

To solve **GP** when f and D are small, we can simply enumerate all possible paths, and then choose one that is optimal. The optimal path consists of one partial edge of N, then a sequence of complete edges, and then another partial edge. We call the part consisting of complete edges the *skeleton* of the path, see Figure 5. Let $P(f, D)$ denote the number of skeleton paths that can start at any given vertex of N. Any skeleton path can consist of at most f edges, because the shortest edge has length s and the fragment can have length at most d. At any vertex except the first, the path has at most $D-1$ possible ways to proceed. Therefore, in the worst case $P(f, D) = D \cdot (D-1)^{f-1}$. The total number of skeleton paths is now at most $m \cdot P(f, D)$, since there are m vertices to start with. We compute all of these, and for each skeleton look for the best path that has that skeleton, and report the best solution.

For $i = 1, \ldots, m \cdot P(f, D)$, let E_i denote the set of edges adjacent to the two endpoints of the i^{th} skeleton path. Furthermore, let m_i denote the number of edges in E_i and let n_i denote the number of sites on the edges in E_i. To find

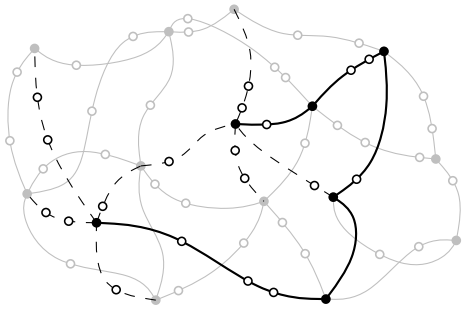


Figure 5: A skeleton path is shown thick. The dashed edges show the possible edges (and corresponding sites) that can still be used to complete the skeleton path to a real path.

the best path using a given skeleton, we have to append two partial edges to its endpoints that cover the largest amount of sites, while their length remains bounded by d minus the length of the skeleton. To be able to do this, we precompute for each edge two lists with the distance to the k^{th} site on the edge, as seen from one endpoint. This takes linear time in total. Then, for a given skeleton, we guess an adjacent edge to both of its endpoints, and then find the best combination of partial edges on those two edges. Note that both edges may be the same edge, in which case the two partial edges can overlap, but when this is the case we can simply take the whole edge. There are D^2 choices for the adjacent edges per skeleton, as illustrated in Figure 5. However, instead of trying each of the D^2 choices, we can directly find the best partial edge using the algorithm from Section 2 in $O(m_i + n_i)$ time.

Now, observe that m_i is at most $2D$. Furthermore, we can bound $\sum_i n_i$. Every edge uv of N is adjacent to the at most $2 \cdot P(f, D)$ skeleton paths that start at u or v, and therefore:

$$\sum_i n_i \leq n \cdot 2 \cdot P(f, D).$$

The total running time now becomes:

$$O(\sum_i (m_i + n_i)) \leq O(\sum_i (D + n_i)) \leq O(D \cdot m \cdot P(f, D) + n \cdot P(f, D))$$

$$= O(m \cdot D^2 \cdot (D-1)^{f-1} + n \cdot D \cdot (D-1)^{f-1}).$$

Theorem 12 *On graphs with degree at most D and smallest edge length s, **GP** can be solved in $O(m \cdot D^2 \cdot (D-1)^{d/s-1} + n \cdot D \cdot (D-1)^{d/s-1})$ time.*

We can use a similar approach for **GT**. A solution again consists of a number of complete edges of N and a number of partial edges. The complete edges are

all connected and now form a *skeleton tree*. An example is shown in Figure 6. Let $T(f, D)$ denote the number of skeleton trees that can contain any given vertex v of N and have degree at most $D-1$ at v. Note that we can make this assumption because every tree has at least one vertex of degree at most $D-1$, and we encounter this vertex as v at some point. We could even count only the number of trees that have degree 1 at v, which would improve the complexity slightly. We do not do that because later we need to reuse $T(f, D)$ in the analysis, and the description would become more complicated, which does not seem worth the small improvement. As with skeleton paths, any skeleton tree can consist of at most f edges.

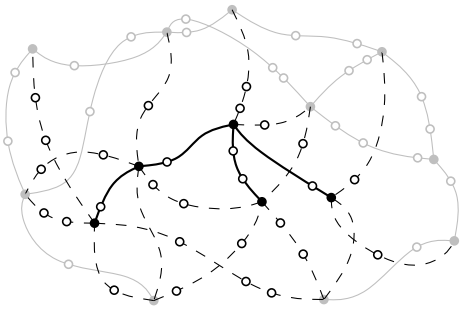


Figure 6: A skeleton tree is shown thick. The dashed edges show the possible edges (and corresponding sites) that can still be used to complete the skeleton tree to a real tree. Some dashed edges have both endpoints on the skeleton tree; these need to be preprocessed before we can solve the problem.

The number of skeleton trees containing a given root vertex with degree at most $D-1$ at the root can be bounded by the number of $(D-1)$-ary trees, which is known to be

$$\binom{(D-1)k}{k-1}/k = \frac{\binom{(D-1)k}{k}}{(D-2)k+1} \approx \frac{(D-1)^{(D-1)k+1/2}}{(D-2)^{(D-2)k+3/2}}/\sqrt{k^3} \leq e^k \cdot (D-1)^k,$$

if they contain k vertices (and $k-1$ edges), cf. [5], see also [35] for an elementary proof. (The approximation uses Stirling's formula, and $e \approx 2.718$ is Euler's constant.) We conclude that $T(f, D) \leq (e(D-1))^{f+1}$. The total number of skeleton trees is now at most $m \cdot T(f, D)$. Again, for each skeleton we compute the best tree that has that skeleton, and report the best solution.

For $i = 1, \ldots, m \cdot T(f, D)$, let E_i denote the set of edges adjacent to some vertex of the i^{th} skeleton tree. Again, let m_i denote the number of edges in E_i and let n_i denote the number of sites on the edges in E_i. Now, all edges that are adjacent to a given skeleton tree might be used partially in a solution. Some of these edges are connected to the skeleton by only one endpoint, and some by both endpoints, see Figure 6. Therefore, as a preprocessing step, we compute for each edge e in N three lists. The first list stores, for each integer

k, the shortest possible path, starting at the left[1] endpoint of e, within e, that contains k sites (in practice it is enough to store the distance to the k^{th} site from the left). The second list stores the same information, but starting from the right endpoint. The third list stores the shortest pair of paths within e, starting at the left and right endpoints of e, which contains k sites. We can easily compute all of these lists in quadratic time.

With this information, we can solve the problem for a given skeleton tree by considering the correct lists for all adjacent edges (depending on which endpoints are in the skeleton tree). We need to find the best combination of partial edges, which can be done in $O(m_i n_i + n_i^2)$ time with the algorithm from Section 3.

Now, observe that m_i is at most $f \cdot D$. Furthermore, we can bound $\sum_i n_i$. Every edge uv of N is adjacent to the at most $2 \cdot T(f, D)$ skeleton trees that have u or v as a leaf, and therefore:

$$\sum_i n_i \leq n \cdot 2 \cdot T(f, D).$$

The total running time now becomes:

$$\begin{aligned} O(\sum_i (m_i n_i + n_i^2)) &\leq \sum_i O(fDn_i + n_i^2 + 1) \\ &\leq \sum_i O(fDn_i) + \sum_i O(n_i^2) + O(m \cdot T(f, D)) \\ &= O(fDn \cdot T(f, D) + (n \cdot T(f, D))^2 + m \cdot T(f, D)) \\ &= O(m \cdot T(f, D) + n^2 \cdot T(f, D)^2) \\ &= O(m(e(D-1))^{f+1} + n^2 \cdot (e(D-1))^{2f+2}). \end{aligned}$$

Theorem 13 *On graphs with degree at most D and smallest edge length s, **GT** can be solved in $O(m(e(D-1))^{d/s+1} + n^2 \cdot (e(D-1))^{2d/s+2})$ time.*

6 Conclusions and open problems

In this paper we studied a point pattern identification problem motivated from hotspot detection. Not surprisingly, the most general versions of the problem, where the network is a graph, were shown NP-complete. Thus we focused on finding cases for which polynomial-time solutions are possible. We showed that if the network N is a tree, efficient algorithms exist to solve the problem. In particular, we gave a linear-time algorithm for the **TP** variant and a simpler $O(mn + n^2)$-time algorithm for **TT**. Furthermore, we gave exact polynomial-time algorithms for networks N of bounded treewidth and for the realistic case in which the maximum degree of the vertices and the minimum edge length in N are bounded.

[1] For ease of explanation we assume here that the endpoints of each edge are arbitrarily defined as 'left' or 'right'.

Various extensions of this work are possible and some open problems remain. First of all, can we give an effective heuristic for the general problem based on our exact and efficient solutions to special cases? For example, we could consider to test various spanning trees of an input network and overlay the solutions to arrive at a global solution. Judging the quality of such an approach requires an extensive experimental evaluation.

On the more theoretical side, how does the problem change if we associate a weight with each site? Our current algorithms for **TP** and **TT** cannot solve the weighted versions. Another question, again inspired by geographic analysis: How about a setting where there are two types of sites, for example cars and accidents? Then we would be interested in a short fragment where the ratio between cars and accidents is small—a question which is related to so-called ratio-clustering.

Acknowledgements

This research was initiated during the GADGET Workshop on Geometric Algorithms and Spatial Data Mining, held in October 2007 and funded by the Netherlands Organisation for Scientific Research (NWO) under BRICKS/FOCUS grant number 642.065.503. The authors thank the other participants, namely Mark de Berg, Maike Buchin, Matya Katz, Marc van Kreveld and Patrick Laube for helpful discussions and for providing a stimulating working environment. Furthermore, we thank the anonymous referees for their careful comments and suggestions to improve the text.

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

K. Buchin and J. Luo were supported by Netherlands Organisation for Scientific Research (NWO) under BRICKS/FOCUS grant number 642.065.503. S. Cabello was supported in part by the Slovenian Research Agency, program P1-0297. M. Löffler and R. I. Silveira were supported by the Netherlands Organisation for Scientific Research (NWO) under the project GOGO. B. Speckmann was supported by the Netherlands Organisation for Scientific Research (NWO) under project no. 639.022.707.

References

[1] K. Aerts, C. Lathuy, T. Steenberghen, and I. Thomas. Spatial clustering of traffic accidents using distances along the network. In *Proc. 19th Workshop of the International Cooperation on Theories and Concepts in Traffic Safety*, 2006.

[2] E. M. Arkin, J. S. B. Mitchell, and G. Narasimhan. Resource-constrained geometric network optimization. In *Proc. 14th Ann. Symposium on Computational Geometry*, pages 307–316, 1998.

[3] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8:277–284, 1987.

[4] B. Awerbuch, Y. Azar, A. Blum, and S. Vempala. New approximation guarantees for minimum-weight k-trees and prize-collecting salesmen. *SIAM Journal on Computing*, 28(1):254–262, 1998.

[5] L. W. Beineke and R. R. Pippert. The number of labeled dissections of a k-ball. *Math. Ann.*, 191:87–98, 1971.

[6] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[7] A. Blum, S. Chawla, D. R. Karger, T. Lane, A. Meyerson, and M. Minkoff. Approximation algorithms for orienteering and discounted-reward TSP. *SIAM Journal on Computing*, 37(2):653–670, 2007.

[8] H. Bodlaender. Treewidth: Structure and algorithms. In *Proc. 14th Colloquium on Structural Information and Communication Complexity*, number 4474 in LNCS, pages 11–25, 2007.

[9] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *J. ACM*, 26(2):211–226, 1979.

[10] M. Celik, S. Shekhar, B. George, J. P. Rogers, and J. A. Shine. Discovering and quantifying mean streets: A summary of results. Technical Report 07-025, University of Minnesota, Computer Science and Engineering, 2007.

[11] C. Chekuri, N. Korula, and M. Pál. Improved algorithms for orienteering and related problems. In *Proc. 19th ACM-SIAM Symp. Discrete Algorithms*, pages 661–670, 2008.

[12] K. Chen and S. Har-Peled. The orienteering problem in the plane revisited. In *Proc. 22nd Ann. Symposium on Computational Geometry*, pages 247–254, 2006.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition, 2001.

[14] S. Fotheringham and P. Rogerson. *Spatial Analysis and GIS*. Taylor and Francis, London, 1994.

[15] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Comp. Sci.*, 1:237–267, 1976.

[16] N. Garg. Saving an epsilon: a 2-approximation for the k-MST problem in graphs. In *Proc. 37th ACM Symposium on Theory of Computing*, pages 396–402, 2005.

[17] B. Golden, L. Levy, and R. Vohra. The orienteering problem. *Naval Research Logistics*, 34:307–318, 1987.

[18] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[19] Illinois Criminal Justice Information Authority. *STAC User Manual*, 1996.

[20] A. P. Jones, I. H. Langford, and G. Bentham. The application of K-function analysis to the geographical distribution of road traffic accident outcomes in Norfolk, England. *Social Science & Medicine*, 42(6):879 – 885, 1996.

[21] D. Karger, R. Motwani, and G. D. S. Ramkumar. On approximating the longest path in a graph. *Algorithmica*, 18(1):82–98, 1997.

[22] T. Kloks. *Treewidth*. PhD thesis, Utrecht University, 1993.

[23] N. Levine. Crime mapping and the Crimestat program. *Geographical Analysis*, 38:41–56, 2005.

[24] H. Miller and J. Han, editors. *Geographic Data Mining and Knowledge Discovery*. CRC Press, 2001.

[25] H. J. Miller and S.-L. Shaw. *Geographic Information Systems for Transportation: Principles and Applications*. Oxford University Press, 2nd edition, 2001.

[26] A. Okabe, K. Okunuki, and S. Shiode. SANET: a toolbox for spatial analysis on a network. *Geographical Analysis*, 38(1):57–66, 2006.

[27] D. O'Sullivan and D. Unwin. *Geographic Information Analysis*. Wiley, 2002.

[28] C. H. Papadimitriou and U. V. Vazirani. On two geometric problems related to the traveling salesman problem. *Journal of Algorithms*, 5(2):231–246, 1984.

[29] J. H. Ratcliffe. The hotspot matrix: A framework for the spatio-temporal targeting of crime reduction. *Police Practice and Research*, 5:05–23, 2004.

[30] J. H. Ratcliffe and M. J. McCullagh. Aoristic crime analysis. *International Journal of Geographical Information Science*, 12:751–764, 1998.

[31] R. Ravi, R. Sundaram, M. V. Marathe, D. J. Rosenkrantz, and S. S. Ravi. Spanning trees - short or small. *SIAM Journal on Discrete Mathematics*, 9(2):178–200, 1996.

[32] T. Rich. Crime mapping and analysis by community organizations in hartford, connecticut. *National Institute of Justice: Research in Brief*, pages 1–11, 2001.

[33] N. Robertson and P. D. Seymour. Graph minors. I. Excluding a forest. *Journal of Combinatorial Theory, Series B*, 35:39–61, 1983.

[34] N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986.

[35] G. Rote. Binary trees having a given number of nodes with 0, 1, and 2 children. *Séminaire Lotharingien de Combinatoire*, B38b, 1997. Comment on a paper by H. Prodinger, 6 pages.

[36] P. G. Spooner, I. D. Lunt, A. Okabe, and S. Shiode. Spatial analysis of roadside Acacia populations on a road network using the network k-function. *Landscape Ecology*, 19:491–499, 2004.

[37] T. Steenberghen, T. Dufays, I. Thomas, and B. Flahaut. Intra-urban location and clustering of road accidents using GIS: a Belgian example. *International Journal of Geographical Information Science*, 18:169–181, 2004.

[38] J. Stillwell and G. Clarke. *Applied GIS and Spatial Analysis*. Wiley, 2005.

[39] I. Yamada and J. Thill. Local indicators of network-constrained clusters in spatial point patterns. *Geographical Analysis*, 39:268–292, 2007.