Detecting Hotspots in Geographic Networks

K. Buchin[1], S. Cabello[2], J. Gudmundsson[3], M. Löffler[1], J. Luo[4], G. Rote[5], R. I. Silveira[1], B. Speckmann[6], and T. Wolle[3]

[1] Department of Information and Computing Sciences, Utrecht University, The Netherlands.
{buchin,loffler,rodrigo}@cs.uu.nl
[2] Department of Mathematics, Inst. for Math., Physics and Mechanics, Ljubljana, Slovenia.
sergio.cabello@imfm.uni-lj.si
[3] NICTA Sydney, Australia.
{joachim.gudmundsson,thomas.wolle}@nicta.com.au
[4] Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, China.
jun.luo@sub.siat.ac.cn
[5] Institut für Informatik, Freie Universität Berlin, Germany.
rote@inf.fu-berlin.de
[6] Department of Mathematics and Computer Science, TU Eindhoven, The Netherlands.
speckman@win.tue.nl

Abstract

We study a point pattern detection problem on networks, motivated by geographical analysis tasks, such as crime hotspot detection. Given a network N (for example, a street, train, or highway network) together with a set of sites which are located on the network (for example, accident locations or crime scenes), we want to find a connected subnetwork F of N of small total length that contains many sites. That is, we are searching for a subnetwork F that spans a cluster of sites which are close with respect to the network distance.

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, a path with self-intersections at vertices, or a tree. Many of these variants are NP-hard, that is, polynomial-time solutions are very unlikely to exist. Hence we focus on exact algorithms for special cases and efficient algorithms for the general case under realistic input assumptions.

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" part 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 (Celik et al., 2007; Levine, 2005; Ratcliffe, 2004; Ratcliffe and McCullagh, 1998; Rich, 2001)). 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 (Miller and Han, 2001) and geographical analysis (Okabe et al., 2006; O'Sullivan and Unwin, 2002). 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 (Illinois Criminal Justice Information Authority, 1996). 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.

In this paper we address the problem of finding hotspots in networks from an algorithmic point of view. We model the problem as follows. The input network N is a connected graph with positive edge lengths. The connected subnetwork F of N which we are searching for is a *fragment* of N, that 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* (locations of interest), 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 Fig. 1). These sites then form a cluster with respect to the network distance. More formally, we consider the following problem:

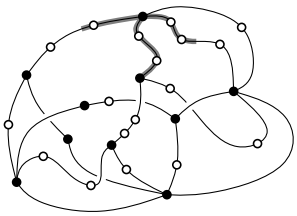

Fig. 1. A network with sites (white circles), nodes (black circles), edges and a fragment (gray).

Most Relevant Fragment. 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, can easily be shown to be NP-complete, that is, polynomial-time solutions are very unlikely to exist. Hence we investigate under which realistic input assumptions the general problem becomes tractable. Furthermore, we present exact and efficient algorithms for special (simple) cases which we believe to be interesting also from a practical point of view, since they can form a solid foundation for effective heuristics that solve the general case.

Notation. We consider various variants where N is either a tree or a graph and F is either a simple path, a path with self-intersections at vertices (but no duplicate edges), 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, **PS** for a simple path (without repeated vertices), and **PI** for a path with possible self-intersections at vertices. For example, **TPS** denotes the instance of the problem where N is a tree and F is a simple path.

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.

Results. Recall that we are given a network N with m edges together with n sites on N. We are looking for a fragment of length at most d which contains the maximal number of sites. 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. We first discuss two more challenging variants where N is a tree. Here optimal and efficient solutions based on dynamic programming exist. In particular, in Section 2 we consider **TPS**: N is a tree and F is a simple path. In this case

we can find the most relevant fragment in $O(n + m)$ time and $O(n + m)$ space. 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.

In Section 4 we study several realistic input assumptions under which efficient algorithms exist for the general problem when N is a graph. If we assume a bound on the maximum vertex degree and on the length of the smallest edge in N—both assumptions are satisfied in general street networks—problems **GP** and **GT** can be solved in polynomial time. The same holds for networks N of bounded treewidth.

Related work. Spatial analysis has been studied intensively in GIS for decades (Fotheringham and Rogerson, 1994) and it has been used in many other areas such as sociology, epidemiology, and marketing (Stillwell and Clarke, 2005). 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 (Aerts et al., 2006; Spooner et al., 2004; Steenberghen et al., 2004; Yamada and Thill, 2007). Like many spatial analysis methods, most spatial network analysis uses statistical methods such as the network K-function method (Spooner et al., 2004). As already mentioned, the problem of finding crime hotspots has received a lot of attention itself (Celik et al., 2007; Levine, 2005; Ratcliffe, 2004; Ratcliffe and McCullagh, 1998; Rich, 2001). 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 (Ratcliffe, 2004). 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* (Golden et al., 1987) (also known as *bank robber problem* (Awerbuch et al., 1998)), as well as to the 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 (Arkin et al., 1998; Awerbuch et al., 1998; Blum et al., 2007; Chekuri et al., 2008; Chen and Har-Peled., 2006). 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 (Arkin et al., 1998)).

2 TPs: Looking for a Simple Path *F* in a Tree *N*

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 then show how to find the most relevant fragment in linear time and space.

2.1 Preliminaries

We assume for simplicity of exposition that no site lies on a vertex of T. Sites at vertices complicate the algorithm a little but 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 Fig. 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.

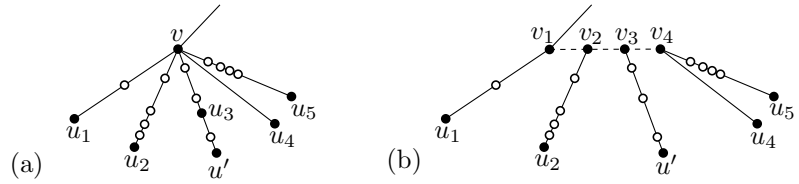

Fig. 2. Transforming the input tree (a) into a rooted tree where each internal vertex has two children (b). The dashed edges in (b) have length zero.

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 (Bender et al., 2005) 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. For each vertex v of T, let $T(v)$ denote the subtree of T rooted at v, and let $m(v)$ 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 $m(vu)$ be the maximum number of sites from S contained in any path from v to a leaf of $T(vu)$. Let $n(F)$ denote the number of sites of S contained in a fragment F of T and let $n(e)$ denote the number of sites of S along the

edge e. Note that $m(vu) = n(vu) + m(u)$. The following bounds, whose proof we omit for lack of space, will be useful to analyze our algorithms.

Lemma 1.

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

and

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

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

The only way in which the length of these lists changes is 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 can be used to implement such arrays with constant access time and amortized constant time for extending them by one element (Cormen et al., 2001, 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 $m(v) = 0$. Consider an internal vertex v. Its two children v_r, v_ℓ have already been processed. We aim for a time bound of $O(n(vv_\ell) + n(vv_r) + \min\{m(vv_r) + m(vv_\ell)\})$ for processing v.

(i) We construct $P(vv_\ell)$ and $P(vv_r)$. $P(vv_\ell)$ is obtained by adding the ordered sequence of $n(vv_\ell)$ sites of S on the edge vv_ℓ to the beginning of the list $P(v_\ell)$. The list $P(v_\ell)$ is destroyed in this operation.

We construct $P(vv_r)$ similarly, and the total amortized running time is $O(1 + n(vv_\ell) + n(vv_r))$.

(ii) We find the best path contained in $T(v)$ that intersects vv_r but not vv_ℓ. We look for a path of length $k_{\max} + 1$ by simultaneously scanning $P(vv_r)$ with a shifted copy of itself. Formally, we start with $j = 1$, and while $j \leq n(vv_r)$ and $j + k_{\max} \leq m(vv_r)$ 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 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. Assume without loss of generality that $m(vv_\ell) \leq m(vv_r)$; the other case would be symmetric. Start with $j = m(vv_\ell)$, and while $j \geq 1$ and $k_{\max} - j + 1 \leq m(vv_r)$ 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 $\min\{m(vv_r), m(vv_\ell)\}$ times, and each time that case (a) occurs, the value $k_{\max}$ is incremented by one. Therefore, this task takes $O(1 + \Delta + \min\{m(vv_r) + m(vv_\ell)\})$ 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 $P(v)$, as follows. Assume without loss of generality that $m(vv_\ell) \leq m(vv_r)$; the other case is symmetric. We will re-use the list $P(v_r)$ to represent the list $P(v)$. For each $j = 1, \ldots, m(vv_\ell)$, 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 $m(vv_\ell)^{\text{th}}$ element are left unchanged. This pairwise comparison of the two lists takes $O(1 + \min\{m(vv_r), m(vv_\ell)\})$ 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 $O(1+\Delta(v)+n(vv_r)+n(vv_\ell)+\min\{m(vv_r)+m(vv_\ell)\})$ 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 $k_{\max}$, 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 $\min\{m(vv_r) + m(vv_\ell)\}$ over all vertices v is $O(n)$ because of Lemma 1. 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.*

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 jth entry, $L(v)[j]$, of $L(v)$ stores the length of the smallest tree fragment of $T(v)$ containing v and covering j points of S. If there is no such tree fragment of $T(v)$ covering j points then 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 jth 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 points $s_1, s_2, \ldots, s_{n(vu)}$ along the edge vu ordered from v to u. For $j = 1, \ldots, n(vu)$, add the jth entry to $L(vu)$ containing the distance between v and s_j. Then, for $j = n(vu), \ldots, n(T(vu))$ we set the jth 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 $O(n^2)$ (see 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 solution 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 points of S on the edge vv_{parent}, ordered from v to v_{parent}. We construct a list $M(v)$, where the jth entry stores the length of the smallest tree fragment of $T(v_{\text{parent}}v)$ that has v as highest vertex and contains j points 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 $O(n(T(v)) \cdot n(vv_{\text{parent}}) = O(n \cdot n(vv_{\text{parent}}))$ time for a vertex v of T, which sums up to $O(n^2)$ 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 2. *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.*

4 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 4.1 we bound the maximum vertex degree of N as well as the length of the smallest edge in N and in Section 4.2 we consider networks N of bounded treewidth. In both cases we describe fixed-parameter tractable algorithms.

4.1 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 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 the best one. 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. The skeleton can consist of at most f edges. The number of skeleton paths of at most f edges is $O(m \cdot D^f)$, since we can start at any vertex, and at each new vertex we have at most D choices of how to proceed. We compute all of these, and then look for the best path that has that skeleton.

To find 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 pre-compute for each edge two lists with the distance to the k-th point 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. Finding the best partial edges takes time linear in the number of sites on those edges, which is in the worst case $O(D^2 n)$. Multiplying this time by the number of skeleton paths we obtain the following result.

Theorem 3. *On graphs with degree at most D and smallest edge length s,* **GP** *can be solved in $O(nm \cdot D^{d/s+2})$ 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 form a *skeleton tree*. We enumerate all skeleton trees of length at most d. A skeleton tree can have at most f edges. The number of skeleton trees containing a given "root" vertex can be bounded by the number of D-ary trees, which is known to be

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

if they contain k vertices (and $k-1$ edges), cf. (Beineke and Pippert, 1971), see also (Rote, 1997) for an elementary proof. (The approximation uses Stirling's formula, and $e \approx 2.718$ is Euler's constant.) Summing this over all possible sizes $k = 1, \ldots, f+1$, we conclude that there are no more than $O(m \cdot (eD)^{f+1})$ skeleton trees.

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. 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 endpoint of e, within e, that contains k sites (so just 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, starting at the left and right endpoints of the edge, that 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 $O(mn+n^2)$ time with an algorithm very similar to that in Section 3. Since we do this for each skeleton tree, the total running time is $O((m^2n+mn^2) \cdot (eD)^{f+1})$.

Theorem 4. *On graphs with degree at most D and smallest edge length s,* **GT** *can be solved in $O((m+n)mn \cdot (eD)^{d/s+1})$ time.*

Note. The running times for a single path or tree in the above proofs are overly pessimistic, since they allow that *all* m edges and *all* n sites enter the calculation. By a more thorough examination of the edges and sites that are actually relevant for each path or tree, the running time of Theorem 3 can be improved to $O(n \cdot D^{d/s} + m \cdot D^{d/s+1})$, and the running time of Theorem 4 can be improved to $O(m(eD)^{d/s+1} + n^2 \cdot (eD)^{2d/s+2})$. Details will be given in the full paper.

4.2 Networks of bounded treewidth

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. (Bodlaender, 2007)). Here we sketch an algorithm for **GT** on a network N of bounded treewidth.

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 $\{v, w\} \in E$ 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 $X_i = X_{j_1} = X_{j_2}$

Using nice tree decompositions often makes it easier to develop and describe algorithms for graphs of bounded tree-width. Any tree decomposition can be converted into a nice tree decomposition of the same width in linear time (Kloks, 1993).

We construct a network N' from N by adding the sites of S as vertices to N'. N' has $|V| + n$ vertices and $n + m$ edges. We refer to a vertex of N' that originated from N or S as *network vertex* or *site vertex*, respectively.

The general approach is as follows. We assume that we are given a nice tree decomposition (T, X) of N' of width $tw(N)$. With each bag i of T, we associate a table containing certain information. These tables represent partial solutions for the subnetwork N'_i of N' that corresponding to the subtree of T rooted at i. More specifically, in the tables we keep track of subforests in N'_i, of their lengths and of the number of site vertices they contain. Such a subforest 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 5. *Given a graph network with m edges whose treewidth is bounded by some constant, 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 points from S.*

5 Conclusions and Open Problems

We studied a network analysis problem motivated by crime hotspot detection. Our approach focused on finding cases for which polynomial-time solutions are possible. Specifically, we showed that if the network N is a tree, efficient algorithms exist to solve the problem: we gave a linear-time algorithm for the **TPS** variant and a simpler $O(mn+n^2)$-time algorithm for **TT**. Furthermore, we gave exact polynomial-time algorithms for the realistic cases in which the maximum degree of the vertices and the minimum edge length in N are bounded, and for networks N of bounded treewidth. Although our algorithms are efficient from a theoretical point of view, the practical suitability of them could only be determined in experiments.

Various extensions of this work are possible. 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. Second: 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.

Acknowledgments

The authors thank Matya Katz for helpful discussions. This research was initiated during the GADGET Workshop on Geometric Algorithms and Spatial

Data Mining, funded by the Netherlands Organisation for Scientific Research (NWO) under BRICKS/FOCUS grant number 642.065.503.

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. 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

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

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

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.

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

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.

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.

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

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.

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.

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

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

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

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

Illinois Criminal Justice Information Authority. STAC user manual, 1996.

T. Kloks. *Treewidth*. PhD thesis, Utrecht University, 1993.

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

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

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

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

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

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

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

G. Rote. Binary trees having a given number of nodes with 0, 1, and 2 children. *Séminaire Lotharingien de Combinatoire*, B38b:6 pages, 1997.

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.

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.

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

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